Université de Reims - Champagne Ardenne

THÈSE

présentée pour l'obtention du grade de

Docteur de l'Université de Reims Champagne-Ardenne

Spécialité : Informatique

Test de robustesse des systèmes temps-réel

Antoine ROLLET

soutenue publiquement le 13 décembre 2004 devant le jury suivant :

Rapporteurs	Richard Castanet	Professeur à l'ENSEIRB
	Jean-Claude Royer	Professeur à l'Ecole des Mines de Nantes
Examinateurs	Alain Bui	Professeur à l'Université de Reims
	Ivan Lavallée	Professeur à l'Université de Paris 8
	Alexander A. Shvartsman	Professeur à l'Université du Connecticut (USA)
Directeur	Hacène Fouchal	Professeur à l'Université d'Antilles Guyane

Laboratoire CReSTIC

[...] *quand on tourne en rond, à quoi ça sert d'aller vite ? Si Carl Lewis me disait : "tiens, Albert, on court un 100 mètres ensemble". Je mettrais vite mes chaussures et je courrais avec lui ; lui n'aura aucun souvenir, mais moi j'aurai un souvenir, c'est ça l'émulation et c'est le contraire de la compétition.*

Albert Jacquard

Remerciements

Tout d'abord, je tiens à remercier toutes les personnes qui, sciemment ou non, m'ont permis d'arriver à ce point.

Je remercie ma famille, et particulièrement mes parents pour leur soutien aussi bien matériel que moral dans les périodes difficiles.

Un grand merci aussi aux (ex) DESS RSI, Cyril, Sébastien et Ludovic pour leur aide précieuse dans la mise au point de notre outil logiciel.

Je remercie aussi les rapporteurs, Richard Castanet et Jean-Claude Royer pour avoir consacré une partie de leur précieux temps à la lecture de ce document, ainsi que tous les autres membres du jury qui ont pu se déplacer pour cette soutenance.

La liste serait incomplète si je ne remerciais pas mes collègues de bureau et amis, qui me supportent (dans tous les sens du terme) depuis tout ce temps, et qui animent le vie de ce bureau. Je pense plus particulièrement à Arnaud, Abbas, Christophe, Cyril, Devan, Florent et Thibault. Je remercie aussi tous mes collègues de l'équipe LICA et mes amis, qui se reconnaîtront...

Mais la personne que je tiens à remercier par dessus tout, sans laquelle rien de tout cela n'aurait été possible, c'est bien entendu Hacène, mon directeur de thèse, qui a toujours su trouver les mots pour me motiver ou m'encourager, et me secouer lorsque cela était nécessaire.

Résumé

En quelques décennies, l'informatique est arrivée au stade d'outil indispensable dans la vie de tous les jours. En lien avec cette évolution, il n'est pas rare de voir des systèmes informatiques responsables de vies humaines ou d'appareils très coûteux, comme dans les centrales nucléaires, le contrôle aérien ou dans les avions. Une défaillance de tels systèmes peut s'avérer catastrophique, ce qui explique que certaines garanties soient obligatoires. Ainsi, des techniques de test doivent permettre de certifier un système et de prévenir un éventuel dysfonctionnement. Cette thèse va se positionner dans le cadre du test de robustesse, plus particulièrement pour le systèmes temps-réel et à base de composants.

Nous allons présenter une méthode pour tester la robustesse d'un système temps-réel. Un système est considéré robuste s'il est capable d'adopter un comportement acceptable en présence d'environnement stressant. Nous avons comme point de départ deux spécifications du système, sous forme d'automates temporisés, une dite nominale qui donne le comportement normal du système, dans des conditions prévues, et une spécification dégradée qui va contenir le comportement minimum autorisé en cas de situation inattendue. Ce sont en fait les actions vitales. Ensuite, nous générons des séquences de test à partir d'une des deux spécifications, et nous y ajoutons des aléas pour provoquer une situation inattendue. Finalement, on évalue de façon postérieure à l'exécution du test, si l'implantation a réagi de façon suffisamment robuste, et ce à l'aide d'une relation d'acceptation.

Par ailleurs, nous proposons une approche et une architecture pour tester la robustesse d'un système temps-réel à base de composants. Chaque composant est relié à son propre testeur, et toutes les interactions entre composants passent par les testeurs correspondants. Chaque testeur contient une file pour stocker les entrées. Dans un premier temps, chaque composant est testé de façon autonome. Ensuite, on applique en parallèle les séquences de test sur chaque composant, et on teste ainsi la robustesse des interactions du système. Un algorithme décrivant l'exécution du test est fourni. Enfin, nous proposons une stratégie simple pour prendre en compte les délais supplémentaires des interactions provoqués par les testeurs.

Finalement, nous présentons un outil de génération de séquences que nous avons développé. Il permet de traiter divers formats logiciels d'automates temporisés, et génère des séquences de test utilisables pour le test de conformité ou pour le test de robustesse. La méthode utilisée cherche à identifier dans l'implantation des états dits contrôlables, à l'aide dans un premier temps d'une méthode inspirée de la méthode UIO, puis pour les états non identifiés après cette étape, avec une méthode inspirée de la méthode Wp.

Mots-clés : test de robustesse, test de systèmes à base de composants, automates temporisés, séquences de test, système temps-réel, outil de génération de séquences.

Abstract

Nowadays, computer science is very important in our lifes. It is very frequent to notice that computers take care of human lives or sensitive systems like nuclear plants, air traffic control or in aicrafts. A failure of such systems may be catastrophic. To prevent such events, some garanties about the system are needed. Thus, testing techniques should help to certify a system and to prevent a potential failure. This document is going to deal with robustness testing, especially for real-time and component based systems.

We present a method to test robustness of a real-time system. A system is considered as robust if it is able to have an acceptable behavior in a stressful environment. We start with two specifications of the system, as timed automata : a nominal specification showing the normal behavior of the system, in an expected situation, and a degraded specification containing the minimal authorized behavior in case of unexpected situations. This represents in fact the vital actions. Then we generate test sequences from a specification, and we add some hazards in order to provoque an unexpected situation. Finally, after the full test execution, we check if the implementation reacted in a sufficient robust way, using an acceptance relation.

In addition, we propose an approach and an architecture to test robustness of a real-time component based system. Each component is linked with its own tester, and all interactions between components are treated by the corresponding testers. Each tester contains a fifo queue to store the inputs. In a first step, each component is tested in an autonomous way. Then test sequences are simultaneously applied on each component, and thus, robustness of interactions of the system is tested. An algorithm describing the test execution is given. Finally, we propose a way to consider the additional delays implied by the use of testers.

Finally, we present a tool for sequence generation. It permits to handle different software formats of timed automata and generates test sequences usefull for conformance or robustness testing. The aim of the method is to identify particular states, called controllable states, in the implementation, using a method inspired by the UIO one in a first step, and using a method inspired by the Wp one for the remaining not identified states in the other steps.

Keywords : robustness testing, real-time component based systems testing, timed automata, test sequences, real-time system, sequences generation tool.

Table des matières

Table des figures

Chapitre 1

Introduction

1.1 Pourquoi valider ?

“ [...] Le lanceur a commencé à se désintégrer à environ H0 + 39 secondes [...] le braquage des tuyères a été commandé par le logiciel du calculateur de bord (OBC) agissant sur la base des données transmises par le système de référence inertielle actif (SRI2). A cet instant, une partie de ces données ne contenait pas des données de vol proprement dites mais affichait un profil de bit spécifique de la panne du calculateur du SRI2 qui a été interprété comme étant des données de vol [...]. “

Rapport de la commission d'enquête Ariane 501 ([Lio96]).

Mars Pathfinder engineers reported a day of flawless operations of the lander and Sojourner Rover on Mars with the end of the mission's 13th day on Mars this morning, and also noted that they have found and are in the process of fixing a software bug that had caused the lander's computer to reset itself four times in recent days.

“ The resets on the lander computer were caused by a software task that was unable to complete the task in the allotted time," said Flight Director Brian Muirhead. "We found that the task was being cut short because it had not been given high enough priority to run through to completion. Basically, we just need to add one instruction to the computer software to raise the priority of that task. “

Mars Pathfinder Mission Status July 17, 1997 11 a.m. Pacific Daylight Time ([NAS97])

“ [...] après sept mois de croisière de la Terre à Mars, des milliers de fichiers se sont accumulés dans la mémoire Flash du robot” (Spirit) “, *causant son débordement et, conséquence, un redémarrage de l'ordinateur. Problème : au reboot , le système de fichiers recharge en mémoire l'image de ses données, reproduisant, comme dans une boucle infinie, l'erreur de saturation de la RAM... et provoquant à nouveau la relance du système.*

Phénomène sans fin, qui aurait sonné le glas de l'engin si ses batteries n'avaient flanché, déclenchant l'entrée du système dans un mode dégradé. Et permettant à l équipe au sol de reprendre le robot en main et de vider sa mémoire. [...] "

01 Informatique no 1757 - 20 février 2004 ([Par04]).

Ces différents extraits montrent des situations où un système a été victime d'une erreur dite "informatique", que l'on appelle aussi couramment un "bug", et dont les conséquences ont été plus ou moins désastreuses mais dans tous les cas très coûteuses. La fusée Ariane 501 a explosé en vol à cause de cette panne. Le robot Pathfinder a été corrigé grâce à un patch envoyé depuis la terre (technique assez couramment utilisée dans le domaine spatial), mais cela a nécessité beaucoup de ressources intellectuelles, matérielles et financières. Enfin le robot Spirit a pu être repris en main grâce à la présence d'un mode dégradé providentiel ...

Ces exemples sont connus car ils ont été fortement médiatisés. Mais il est possible de trouver un nombre incalculable de rapports et articles de ce genre, faisant état de défaillance d'un système informatique, que ce soit dans le domaine spatial, ou dans la vie de tous les jours. A titre indicatif, une liste de pannes répertoriées dans le domaine spatial est fournie en annexe, (cf Annexe A) dans laquelle on peut constater le nombre impressionnant de problèmes dus à l'informatique.

La raison, c'est que maintenant l'informatique fait partie intégrante de notre vie. Personne n'oserait en douter. On trouve des systèmes informatiques partout, et notamment dans des domaines où des dysfonctionnements peuvent coûter très cher, que ce soit en termes financiers (on évalue le coût de l'erreur du processeur pentium sur les calculs de flottants à plus de 470 Millions de US$), ou en termes de vies humaines (avions, voitures, contrôle aérien, centrales nucléaires, etc ...). On peut citer par exemple la défaillance du robot médical Therac-25, qui entre 1985 et 1987 a tué plusieurs patients en les exposant à des radiations trop intenses, et ce à cause d'une erreur logicielle ([LT93]).

Dans une société où l'on veut toujours aller plus loin et plus vite, et avec les progrès des matériels utilisés, il n'est pas étonnant que les systèmes deviennent de plus en plus complexes. Les systèmes sont utilisés en parallèle, ou de façon distribuée... D'autres part, la tendance est aussi à la conception et la réutilisation de composants. Tout ceci forme un berceau fabuleux pour toutes sortes de dysfonctionnements et de pannes dus à l'informatique.

Dans le domaine des systèmes embarqués, les difficultés décrites ci-dessus sont encore accrues. En effet, grâce à la miniaturisation, à la baisse des prix des composants, et surtout à l'évolution de nos modes de vie, ces types de systèmes sont en très forte expansion en ce moment, et la tendance n'est pas prête de s'inverser. Par ailleurs, la responsabilité de vies humaines leur est plus souvent confiée que pour les systèmes "classiques", et ils sont parfois soumis en parallèle à des contraintes dues au manque de place, et à des conditions d'utilisation plus difficiles (avionique ou domaine spatial). Ajoutons à cela que dans ces systèmes, souvent conçus pour évoluer en milieu hostile, la possibilité d'intervenir pour

réparer est souvent difficile à mettre en place (voire impossible). En effet, il est difficile d'envisager de couper le système de commande d'un avion en plein vol pour le réparer !

C'est pour toutes ces raisons que, lorsqu'on utilise ce genre de systèmes, il est nécessaire d'avoir quelques garanties de sécurité, de fiabilité et de robustesse... et ainsi on a mis en place ce qu'on appelle la validation.

1.2 Développement d'un système

1.2.1 Le cycle de vie

Aux premiers pas de l'informatique, le programmeur avait une idée plus ou moins claire de son application, et il partait directement dans la réalisation. Cette approche était suffisante lorsqu'il s'agissait de développer un petit jeu ou de faire des calculs simples. Mais il faut reconnaître que la fiabilité de ce genre de systèmes pouvait parfois laisser à désirer, surtout dans le cas d'applications plus complexes.

Ensuite on a essayé de mettre en place des méthodes de développement plus efficaces et plus complètes, allant des premières idées sur le système à l'implantation finale validée. C'est ce qu'on appelle le **cycle de développement du logiciel** ou encore **cycle de vie**. L'idée, c'est de partir d'une description formelle de haut niveau du système, afin d'éviter les ambiguïtés et les imprécisions causées par le langage humain, et d'arriver à un produit fini ayant subi toutes les validations nécessaires. Ce cycle découpe le développement en différentes phases clé, généralement successives, qui s'appuient chacunes sur le résultat de la phase précédente. A chaque moment du cycle, on contrôle la qualité du produit en cours, et en cas d'échec à un de ces contrôles, des retours en arrière sont prévus. Il va de soit que plus une erreur est détectée tard, plus sa correction est coûteuse (le pire des cas étant après la mise en service...), et c'est pourquoi il ne faut pas négliger les phases de validation, même si elles entraînent des frais supplémentaires. Mieux vaut prévenir que guérir... La technique de conception la plus éprouvée est la méthode V (développée par Barry Boehm en 1981). Nous proposons par la suite un cycle de vie assez général et détaillons les différentes étapes.

1.2.2 Les étapes du cycle de vie

La figure 1.1 décrit les différentes phases du cycle de vie.

Il est constitué des étapes suivantes :

1. **Phase d'analyse des besoins et exigences**
   Durant cette phase préliminaire, on va écrire de la façon la plus claire possible les besoins et les fonctionnalités du logiciel. Cette étape est la seule qui se fasse en général en langage humain, mais toutefois avec une certaine rigueur. Tous ces aspects vont alors être écrits dans le cahier des charges. Aucun aspect matériel n'est abordé. Elle doit néanmoins être effectuée avec soin, car c'est le point départ de

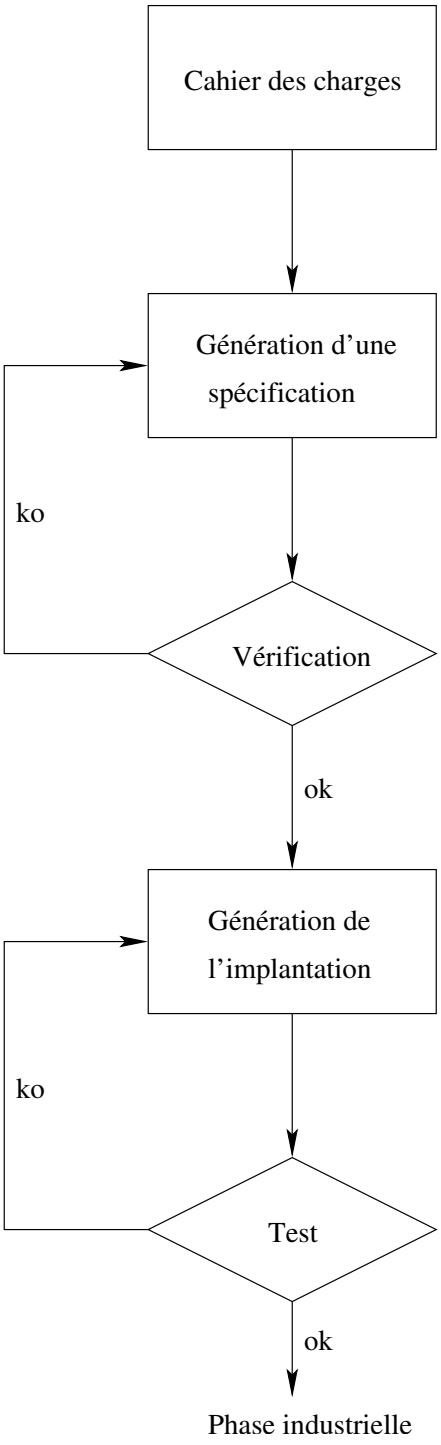


FIG. 1.1 – Le cycle de vie

tout le système, et ce document sert en général de contrat ou de preuve en cas de contestation de la part du client. Lors de cette phase, toutes les exigences du système doivent être clairement stipulées. Par exemple, dans le cas d'un système temps-réel, les contraintes de temps doivent y figurer.

2. **Phase de conception**
   Lors de cette phase, on utilise le cahier des charges comme base pour traduire précisément toutes les propriétés "haut niveau" du système. Pour cela, on choisit un formalisme rigoureux qui permette une description concise et complète, tout en évitant les imprécisions. C'est ce qu'on appelle la spécification. Le choix dépendra aussi du type d'applications que l'on souhaite mettre en place. Il est possible d'utiliser un grand nombre de formalismes de spécification, comme les logiques (temporelles) ([ACE87]), les machines à états finis ([Gil62]), ou encore les automates temporisés ([AD94]), etc... Il est aussi envisageable de décrire le système avec plusieurs spécifications, selon le niveau d'abstraction et de détail que l'on souhaite. Pour choisir le langage de description, de nombreuses questions se posent : quelles sont les conditions d'utilisation ? Y a-t-il des contraintes de temps ? Peut-on utiliser un modèle synchrone ? Nous reverrons ces aspects plus en détail par la suite.
3. **Phase de validation de la spécification**
   On l'appelle aussi phase de vérification. L'intérêt d'avoir choisi un langage formel dans la phase de conception, est certes d'avoir une description plus rigoureuse du système, mais aussi de pouvoir appliquer des techniques d'analyse de propriétés de toute sorte ou encore de faire des démonstrations. Cette phase est cruciale, car il faut bien se rendre compte qu'un système peut entraîner une erreur alors qu'il respecte parfaitement sa spécification... seulement l'erreur, en fait, se trouvait dans la spécification même ! Ainsi, on va vérifier des propriétés logiques (ou autres) telles que des propriétés de sécurité ou de vivacité. C'est dans cette phase qu'on regarderait par exemple si dans le système de contrôle d'un passage à niveau, une barrière n'est jamais ouverte pendant le passage d'un train... Une des techniques les plus connues de vérification est le model checking ([ACD90], [HNSY92], [LPY95]).
4. **Phase d'implantation**
   A ce stade, on commence à s'intéresser aux aspects matériels du système. C'est ici que l'application réelle (que l'on espère) finale est développée sous forme de code exécutable. On utilise la spécification pour écrire le programme de façon précise. Actuellement, la tendance est à la génération automatique de code à partir de la spécification, et ce pour éviter une intervention humaine, souvent synonyme d'ajout de fautes.
5. **Phase de validation de l'implantation**
   Cette étape est aussi appelée test. Elle consiste à soumettre le système considéré comme final à toute une série d'épreuves pour voir si son comportement est bien celui attendu. C'est dans cette phase du développement que se situe notre travail. Il est possible de tester toutes sortes de situations, que ce soit la conformité avec la spécification, l'interopérabilité, les performances, ou la robustesse. Ces aspects sont détaillés par la suite. Cette phase est en quelques sortes le dernier rempart avant la mise en service du produit, et elle ne doit surtout pas être négligée. Elle sert à mettre en évidence des éventuels dysfonctionnements du système.

1.3 Différents types de test

1.3.1 Vocabulaire

" Le test permet de prouver l'existence de faute, et non l'absence de faute" (Dijkstra). Cette phrase résume en quelques mots les limites du test. On peut essayer de l'améliorer tant que l'on voudra ou que l'on pourra, il restera toujours une certaine probabilité de dysfonctionnement d'un système non détecté durant la phase de test.

En ce qui concerne le vocabulaire, il est important d'insister sur la différence que nous faisons entre les termes validation, vérification et test. La **vérification** consiste à s'assurer que la spécification du système possède bien les propriétés requises. En revanche, le **test** c'est l'activation du système réel final (ie l'implantation). Ces deux étapes sont complémentaires. En effet, les langages de spécification ne permettent pas toujours de représenter tout ce qui peut se passer dans la réalité, car ils sont influencés par un environnement non contrôlé. En revanche, la phase de test peut ne pas mettre en évidence une erreur de spécification. Le terme de **validation** est un terme général qui englobe les deux autres.

Généralités sur la vérification

Il existe trois principales méthodes pour vérifier une spécification :

- le *Theorem proving (ou déduction automatique)* : elle permet, à partir d'un système et d'une propriété exprimée dans un même langage de spécification, de prouver que la propriété est vérifiée ou non par le système. Ceci est réalisé en utilisant des règles de déduction, comme on pourrait le faire pour montrer un théorème de mathématiques. Un démonstrateur permettant de faire une telle preuve pour chaque système et chaque propriété est bien entendu impossible à construire, mais actuellement des "assistants à la preuve" permettent d'écrire des preuves, l'utilisateur devant donner des indications à l'outil (voir par exemple [HKPM97] et [Rus01]).
- le *Model checking* : c'est une procédure automatique qui permet de vérifier qu'un modèle d'un système vérifie une spécification. Cette procédure de vérification ne se fait pas "par déductions" comme dans le cas de la déduction automatique mais grâce à des algorithmes tirant profit des modèles utilisés pour le système et pour la propriété, par exemple un étiquetage d'états d'un graphe avec des sous-formules ([CGP99]). Un algorithme de *Model checking*, appelé *"model checker"*, à partir d'une spécification sous forme d'automate et de propriétés sous forme de logique *CTL* est donné dans [CES86].
- l'*Equivalence checking* : permet de vérifier si deux systèmes sont "identiques" selon une relation d'équivalence donnée. Une étude sur les relations d'équivalence est proposée dans [Gla90]. Pour pouvoir utiliser ces relations, il faut qu'elles soient décidables, ce qui n'est pas un problème simple à résoudre.

Une étude appronfondie sur les techniques de vérification de logiciel se trouve dans [Sch99].

Test

Il existe différentes sortes de test selon les aspects que l'on cherche à mettre à l'épreuve. Voici les plus connues :

1. **test de l'usager**
   Ce type est le plus ancien et le plus intuitif. On l'appelle aussi beta-test. Il y a peu, c'était la technique la plus utilisée... L'idée est de sélectionner des utilisateurs pour mettre à l'épreuve de façon intensive le logiciel. Plus on choisit de personnes, et plus le test est long, plus les chances de trouver des fautes sont grandes... L'avantage, c'est que cette technique coûte très peu cher, mais bien entendu, les garanties sur le produit sont assez faibles. En effet, on ne peut pas être certains que toutes les parties du système, ou au moins les parties critiques, sont testées. De même, il est difficile de réitérer plusieurs fois le même jeu de tests. Ce test peut être appliqué à tout type de système.
2. **test d'interopérabilité**
   Permet de vérifier si les interactions entre plusieurs systèmes se font correctement. Pour cela, on analyse les communications entre les systèmes testés. Cette technique est utilisée lorsque des systèmes sont amenés à communiquer entre eux, et notamment en cas de systèmes complexes.
3. **test de performance**
   Permet d'évaluer et d'analyser les performances du système avec des critères comme la vitesse de calcul, le débit d'une information, le temps de réponse, etc... Ce test est applicable à tout type de système.
4. **test de conformité**
   Permet de s'assurer que l'implantation testée réagit bien comme décrit dans sa spécification. Ce genre de test a été élaboré au départ pour être appliqué aux protocoles de communication. Il a fait l'objet d'une normalisation en 1991 par la norme ISO9646 ([ISO91]). En général, on applique des séquences de test sur l'implantation, et on vérifie la conformité des réponses par rapport à la spécification.
5. **test de robustesse**
   Le but ici est de s'assurer que le système réagit correctement en milieu hostile, dans des situations non prévues dans la spécification. En effet, dans ce genre de milieux, le système va être soumis à toutes sortes d'aléas qu'il est malheureusement impossible de prédire, mais qu'il faut malgré tout anticiper.

1.3.2 Boîte blanche vs boîte noire

Boîte noire

Les différents types de test cités précédemment sont généralement considérés comme des tests en boîte noire, ce qui signifie en fait qu'on ne peut rien observer ni supposer sur le système que l'on est en train de tester. Les seuls renseignements disponibles sont ceux fournis par les interfaces reliant le système à son environnement. On les appelle aussi les tests fonctionnels.

Boîte blanche

En revanche, dans les tests en boîte blanche, appelés aussi tests structurels dans le cas d'un logiciel (voir [Pha94]), on considère que la structure interne du système est observable, et ainsi que l'on peut atteindre toute donnée nécessaire du système pour contrôler sa validité. En général, de ce genre de tests, on considère que le code de l'application testée est connu.

Boîte grise

En fait, cette catégorie est tout simplement intermédiaire entre les deux précédentes. On ne considère pas que la structure interne complète du système est connue, mais on a quand même la possibilité d'accéder à certaines informations précises appartenant à cette structure interne. Ces informations peuvent être nécessaires dans certains cas pour contrôler la validité du système.

1.4 Contenu de cette thèse

Dans ce document, nous allons aborder le test de robustesse de systèmes temps-réel et de systèmes temps-réel à base de composants. Une principale contribution sera l'intérêt que nous allons porter aux aspects comportementaux du système, avec prise en compte de temps, lorsqu'il est soumis à une situation inattendue.

Nous ferons volontairement une distinction entre le *test de tolérance aux fautes* et le *test de robustesse*. Le premier sert à mettre en évidence des mécanismes tels que la redondance, dans le cas où une partie du système se trouve hors-service. Le deuxième s'assure que le système adopte un comportement "acceptable" en cas de situation critique. C'est ce dernier qui nous intéressera.

Après avoir rappelé les différents modèles utilisés dans les méthodes de validation (chapitre 2) ainsi que les principaux travaux sur le test (chapitre 3), nous allons présenter dans le chapitre 4 une méthode complète pour tester la robustesse d'un système temps-réel. Nous avons choisi une approche en boîte noire.

Le concepteur doit fournir deux spécifications sous forme d'automates temporisés : une dite nominale et l'autre dite dégradée. La première exprime le comportement du système dans des conditions normales et prévues. La seconde exprime le comportement minimum que l'on autorise en cas de situation critique. On appelle cela les actions vitales. A l'aide d'une de ces spécifications, on génère des séquences de test que l'on applique à l'implantation. A la fin, on analyse les traces d'exécution pour décider, à l'aide d'une relation, si le système est assez robuste. La particularité de cette méthode, c'est qu'on ne sait pas à l'avance quel sera exactement le comportement du système. On doit donc évaluer que ce comportement se trouve dans les limites "acceptables".

Ensuite, nous abordons dans le chapitre 5 le test de systèmes à base de composants. Nous proposons une architecture de test, dans laquelle chaque composant est relié à son propre testeur. Toutes les communications entre composants passent par les testeurs.

Ainsi, après avoir testé chaque composant de façon autonome, nous proposons de tester la robustesse des communications du système global. Pour cela, chaque testeur contient une file d'attente, dans laquelle les actions venant d'un autre testeur sont stockées. Un algorithme complet de l'exécution du test est fourni.

Par la suite, nous présentons dans le chapitre 6 un outil de génération de séquences de test que nous avons développé. Celui-ci se base sur un algorithme de génération inspiré des méthodes UIO et Wp pour les systèmes non temporisés. Cet outil accepte en entrée divers formats de fichier répandus pour décrire les automates temporisés, et génère des séquences temporisées, utilisables pour tester la conformité. Ensuite, il permet d'insérer des aléas dans les séquences pour pouvoir tester la robustesse du système.

Finalement, le chapitre 7 permet de conclure et de donner les diverses perspectives de ce travail.

Chapitre 2

Modélisation des systèmes

Il existe une quantité impressionnante de modèles mathématiques permettant de décrire le comportement d'un système. Nous n'allons pas tous les citer ici, mais nous allons plutôt essayer de voir les plus couramment utilisés dans les méthodes de vérification et de test. Rappelons que pour qu'un modèle soit utilisable, il doit avoir des fondements mathématiques, et par conséquent une sémantique correcte. Le choix du modèle dépend ensuite du type de système que l'on souhaite développer, et aussi des conditions dans lesquelles ce système est censé évoluer. C'est ce modèle qui servira à écrire la spécification du système. Pour plus de clarté, nous avons divisé ces modèles selon des grandes catégories.

Une différence cruciale entre différents modèles, c'est la prise en compte des aspects temporels. Cette notion est arrivée bien plus tard, mais a toute son importance dans les systèmes soumis à des contraintes de temps. Encore une fois, on préfèrera s'assurer que la barrière du passage à niveau est bien abaissée avant que le train ne soit arrivé...

Un autre notion importante concerne la représentation que l'on se fait du temps. On peut considérer des modèles asynchrones, c'est à dire que le temps est traité de façon continue et que l'on ne fait aucune hypothèse sur les dates des événements. C'est l'hypothèse la plus proche de la réalité mais elle est difficile à mettre en place, notamment pour la spécification de systèmes concurrents. En revanche, on peut considérer dans un souci de simplification que les réactions prennent un temps nul, et que tous les événements ont lieu à des instants précis (des tops d'horloge) de durée nulle : c'est l'hypothèse synchrone.

2.1 Systèmes non temporisés

Dans cette section, nous allons voir des modèles mis au point pour spécifier des systèmes dans lesquels le temps ne fait pas partie des contraintes exprimées par le cahier des charges. Dans ce genre de systèmes, le concepteur et le client ne posent aucune condition particulière quant aux délais de traitement et de calcul.

2.1.1 Machines à états finis

Une machine à états finis (FSM) est un graphe décrivant les actions de la spécification. Les états de cet automate décrivent les états internes du système. Un ensemble de symboles d'entrée et un ensemble de symboles de sortie sont définis pour modéliser respectivement les signaux émis vers la spécification et reçus à partir de cette dernière. Ce modèle est très utilisé pour spécifier des protocoles de communication. La figure FIG 2.1 montre un exemple de FSM.

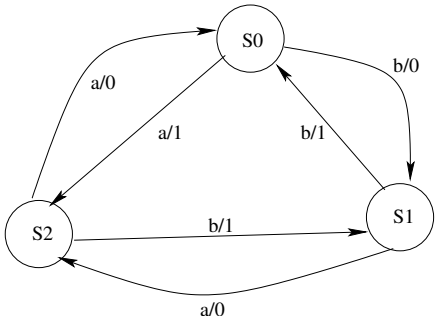


FIG. 2.1 – Un exemple de FSM

Une théorie complète sur les machines à états finis (FSM) peut être trouvée dans [Gil62]. Nous rappelons juste les notions essentielles, et qui interviennent souvent dans les méthodes de test. Il s'agit d'un graphe définit ainsi :

Définition 2.1.1 (Machine à états finis (FSM)) *Une FSM M est un quintuplet $< s_0, S, I, O, T >$ où :*

1. *s_0 est l'état initial,*
2. *S est un ensemble fini d'états, non vide,*
3. *I est l'ensemble non vide des actions d'entrée,*
4. *O l'ensemble non vide des actions de sortie,*
5. *T est un ensemble de transitions. Une transition est un quadruplet $< s_d, i, o, s_a >$ avec $s_d \in S$ correspondant à l'état de départ, $s_a \in S$ l'état d'arrivée, $i \in I$ et $o \in O$ respectivement l'action d'entrée qui déclenche le passage de la transition, et l'action émise au passage de cette transition. De façon plus brève, on note : $s_d \xrightarrow{i/o} s_a$.*

Définition 2.1.2 (Déterminisme) *Une FSM $M =< s_0, S, I, O, T >$ est dite déterministe ssi*
$\forall s, s', s'' \in S$, $\forall i, i' \in I$, $\forall o, o' \in O$ tels que $s \xrightarrow{i/o} s'$ et $s \xrightarrow{i'/o'} s''$, $s' \neq s''$ implique $i \neq i'$.

Définition 2.1.3 (Complétude) *Une FSM $M =< s_0, S, I, O, T >$ est dite complète ou complètement spécifiée si pour chaque état de $s \in S$, $\forall i \in I$, il existe une transition partant de s déclenchée par l'entrée i.*

Dans notre exemple FIG 2.1, si le système se trouve en S0, la réception d'un signal d'entrée b déclenchera l'émission du signal 0, ainsi que le passage du système à l'état S1. Le modèle FSM est prévu pour l'émission et la réception sur la même transition. Beaucoup de méthodes de test ont été développées à partir de ce type de modèles, comme la méthode du tour de transition ([NT81]), la méthode UIO ([SD88]) ou la W ([Cho78]).

2.1.2 Systèmes de transitions étiquetées

Le système de transitions étiquetées (LTS) est un modèle simple bas niveau qui permet de décrire la plupart des systèmes. Il s'agit en fait d'un ensemble d'états reliés par des transitions étiquetées par des actions.

Définition 2.1.4 (Système de transitions étiquetées (LTS)) *Un LTS est un quadruplet* $< S, L, T, s_0 >$ *où :*

1. S *est un ensemble non vide d'états*
2. L *est un ensemble non vide d'interactions ;*
3. $T \subseteq S \times (L \cup \{\tau\}) \times S$ *est la relation de transition. (*τ *représente un action interne)*
4. s_0 *est l'état initial*

Il est utilisé aussi utilisé pour décrire des sémantiques de langages tels que CCS ([Mil89]), CSP ([Hoa78]) ou LOTOS ([ISO87]). Il est aussi utilisé dans les travaux théoriques sur le test comme dans [Bri87] ou [Pha94]. La figure FIG 2.2 montre un exemple très simple de LTS.

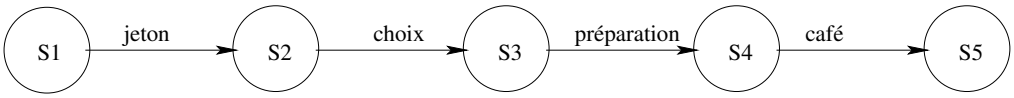


FIG. 2.2 – Un exemple de LTS : une machine à café

Souvent, pour modéliser les spécifications et les implantations, on utilise une variante des LTS, en considérant l'ensemble des états comme fini, et en divisant les interactions en deux sous-ensembles : les actions d'entrée (symbole précédé d'un " ?") et les actions de sortie (symbole précédé d'un " !"). On appelle ce modèle un *automate à entrées et sorties (IOSM).*

Définition 2.1.5 (Automate à entrées et sorties (IOSM)) *Un IOSM est un quadruplet* $< S, L, T, s_0 >$ *où :*

1. S *est un ensemble fini non vide d'états*
2. L *est un ensemble non vide d'interactions ;*
3. $T \subseteq S \times ((\{?, !\} \times L) \cup \{\tau\}) \times S$ *est la relation de transition*
4. s_0 *est l'état initial*

Ce formalisme peut aussi être vu comme une variante des FSM, et ce à cause du nombre fini d'états, et la possibilité de manipuler des entrées/sorties.

2.1.3 Réseaux de Petri

Les réseaux de Petri (RdP) sont un outil à la fois graphique et mathématique. Cette théorie a été présentée par Carl Adam Petri en 1962 ([Pet62]). Ce modèle est très utilisé, notamment pour spécifier des systèmes à événements discrets. Il a d'ailleurs donné naissance à une célèbre forme altérée en France : le *Grafcet*, pour la programmation des automates. Le RdP est très pratique pour exprimer le parallélisme et le partage de ressources. Il est défini comme suit :

Définition 2.1.6 (Réseau de Petri (RdP)) *Un réseau de Petri est un quadruplet* $R =< P, T, Pre, Post >$ *où :*

- P *est un ensemble fini de places*
- T *est un ensemble fini de transitions*
- $Pre : P \times T \longrightarrow \mathbb{N}$ *est l'application incidence avant (places précédentes)*
- $Post : P \times T \longrightarrow \mathbb{N}$ *est l'application incidence arrière (places suivantes)*

Un réseau marqué est le couple $N =< R, M >$ *où :*

- R *est un réseau de Petri*
- M *est le marquage initial, c'est une application* $M : P \longrightarrow \mathbb{N}$.
  $M(p)$ *est le nombre de marques (jetons) contenus dans la place* p.

Contrairement aux automates, une transition est segmentée en arcs entrants et sortants. Chaque arc sortant est valué par un nombre entier représentant le nombre de jetons que la place de départ doit avoir pour que la transition puisse être franchie. Dans ce cas, la place de départ perd le nombre de jetons correspondant. De même, chaque place d'arrivée reçoit un nombre de jetons égal à la valuation de l'arc entrant. La figure FIG. 2.3 montre un exemple de réseau de Petri.

Même si le niveau d'abstraction est plus bas que LOTOS ou SDL, les réseaux de Petri sont très appréciés notamment dans les domaines de l'évaluation de performance ou de la vérification, grâce en partie à la théorie mathématique sous-jacente ainsi qu'aux nombreux outils déjà existants. De nombreuses extensions des RdP ont été mises au point, pour la prise en compte du temps entre autres. On pourra citer les RdP temporisés ([Ram74]), les RdP temporels ([MF76]) ou encore les RdP stochastiques temporisés ([GJ95]).

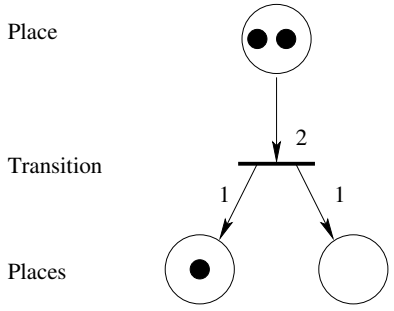


FIG. 2.3 – Un exemple de réseau de Petri

2.1.4 La méthode B

L'objectif de la méthode B ([Hab01]) est de fournir des systèmes conformes à leur spécification, et ce de façon prouvée. En fait cette méthode couvre une partie du cycle de vie du système, allant de la spécification (dite technique) à la génération de l'implantation. Le principe de cette méthode est de partir de la spécification, et de la raffiner progressivement, en prouvant à chaque étape que le raffinement est correct, pour arriver finalement à du code exécutable.

Un composant en B est appelé *machine abstraite*. Il se présente sous forme de texte ASCII, avec un langage de programmation spécial pour le B, contenant entre autres des notations logiques et ensemblistes. On doit toujours pouvoir prouver sa validité. Elle contient des *données* et offre des *opérations*. Ensuite on peut la raffiner en une autre machine abstraite. De même, toute machine abstraite peut être implantée par une autre machine abstraite, ce qui permet d'obtenir un code exécutable équivalent (en fait le dernier raffinement est l'implantation). L'idée centrale de la méthode B, c'est que toute action du programmeur s'accompagne d'une obligation de preuve.

L'intérêt de cette technique est, bien entendu, de fournir à la fin du développement un système vérifié et dont la validité est prouvée. C'est pourquoi cette technique est de plus en plus utilisée, en particulier dans le développement d'applications sensibles. On remarque par ailleurs que la méthode B (de base) ne prend pas en compte les étapes de test.

2.1.5 Formalismes de plus haut niveau

Dans cette partie, nous exposons des langages qui ont un degré d'abstraction plus élevé. Ils ont pour avantage de pouvoir décrire des comportements de façon brève (selon le degré d'abstraction), mais en contrepartie, ils manquent de capacité à exprimer des notions proches des implantations, ce qui est souvent utile pour les techniques de test. Parmi ces langages, nous pouvons citer les logiques temporelles, Estelle, LOTOS, SDL ou encore les réseaux de Petri. Notons qu'Estelle, LOTOS et SDL ont été normalisés et qu'ils font l'objet d'une étude complète dans [Tur93].

Logiques temporelles

Une étude complète de ces types de logiques se trouve dans [ACE87]. Rappelons qu'une logique modale peut être vue comme un langage muni d'un mécanisme déductif. Ce langage est constitué de *formules* construites à partir de *variables propositionnelles* et de *constantes logiques*, permettant de former des expressions complexes à partir des formules élémentaires et de règles de formation. Ensuite, le mécanisme déductif se fait grâce à la composition d'axiomes et de règles d'inférence. D'autre part, on définit une sémantique caractérisée par les notions de *modèle* et de *satisfaisabilité*. Par rapport à une logique classique, la logique modale prend en compte une nouvelle définition "d'implication stricte", faisant intervenir les notions de possibilité et de nécessité. Les logiques modales les plus utilisées sont les systèmes S4 et S5.

Pour palier à un manque quant à la datation des événements, la logique temporelle a été introduite. L'idée est de pouvoir exprimer la consécution entre dates. C'est un langage qui permet d'exprimer l'ordonnancement, mais en aucun cas de travailler sur les aspects contraintes de temps (contrairement à ce que le nom pourrait sous-entendre). Par rapport à la logique modale, des opérateurs pour le passé et le futur ont été ajoutés, permettant de définir une syntaxe complète ainsi qu'une nouvelle sémantique. On peut citer CTL ([CE81]) comme exemple de logique temporelle classique. Des extensions dont le but est d'exprimer les contraintes de temps ont été écrites, comme TCTL ([ACD90], [Yov93]), MTL ([AH90]), ou RTTL ([Ost92]).

Estelle

Dans les années 80, le CCITT[1] et l'ISO[2] initient le développement du langage Estelle pour modéliser les protocoles de communication. Comme LOTOS et SDL, il est normalisé par une norme ISO : ISO9074 ([ISO86]). Il s'agit en fait d'une technique de description formelle pour la spécification des systèmes concurrents distribués, prévue surtout pour spécifier les protocoles et les services de télécommunication. Une description détaillée se trouve dans [BD88].

On peut décrire Estelle comme une technique basée sur un modèle d'états-transitions étendu (avec des timers). En fait, il s'agit d'une structure hiérarchique d'automates non déterministes (appelés modules), étendus par des expressions en langage Pascal (pour la description des données), qui peuvent s'exécuter en parallèle (de manière asynchrone) et communiquer entre eux en échangeant des messages ou en partageant certaines variables. Estelle permet de séparer la description des interfaces de communication entre composants et la description du comportement interne de chacun d'eux.

L'interface de communication entre composants est définie à l'aide de trois concepts :
- les points d'interaction (internes ou externes)
- les canaux
- les interactions proprement dites

Un module est représenté graphiquement comme un rectangle, avec des points sur la bordure (points d'interaction externes) ou à l'intérieur (points d'interaction internes). Des flèches sont utilisées pour exprimer le sens d'interaction.

Finalement, grâce à sa hiérarchisation, et à la séparation interface/comportement, Estelle permet de définir bon nombre de systèmes réels, pas seulement dans le domaine des télécommunications. A une période d'émergence de systèmes à base de composants, cette idée novatrice à l'époque reste encore un très bonne approche dans la spécification de tels systèmes. On notera toutefois que son niveau d'abstraction est plus faible que LOTOS, car il est très proche du formalisme de l'automate.

Dans le domaine du test, Paul D. Amer a étudié ([FUDA00]) une faiblesse d'Estelle : les conflits de timers. En effet, il se peut que dans un modèle de protocole sous forme d'état-transitions, il existe des chemins infaisables, ce qui peut ensuite induire des séquences de

[1]Comité Consultatif International Téléphonique et Télégraphique, remplacé maintenant par l'International Union of Telecoms

[2]International Standardization Organisation

test irréalisables. En fait, il s'agit d'un cas particulier du problème de la faisabilité de séquences de test. Des solutions sont étudiées dans [UD99] et [FUDA00].

LOTOS

Le langage LOTOS[3], a été prévu pour spécifier l'architecture et le fonctionnement de systèmes distribués. C'est un langage asynchrone de haut niveau. Il est défini intégralement dans la norme ISO8807 ([ISO87]). Une présentation complète peut être trouvée dans [Gar90]. A la base, il était prévu pour spécifier des services et des protocoles de télécommunication mais son domaine d'utilisation s'est généralisé. Il fait partie des langages normalisés par l'ISO et le CCITT. Comme Estelle et SDL, il possède un grand pouvoir d'expression, une syntaxe et une sémantique rigoureuses.

Une spécification LOTOS se présente comme du code, sous forme de texte. On y définit des processus pour décrire le contrôle et des types pour les données. En fait, une spécification LOTOS se décompose en deux parties bien distinctes : les structures de contrôle et les structures de données. On utilise aussi des identificateurs, répartis en six classes (processus, portes, variables, types, sortes, opérations). Grâce à la possibilité de décrire des comportements qui s'exécutent en parallèle, et qui se synchronisent par un mécanisme de rendez-vous, LOTOS permet d'exprimer une grande quantité de comportements réels, notamment dans les systèmes répartis.

Les structures de contrôle de LOTOS sont inspirées des algèbres de processus, notamment de CSP ([Hoa78]) et CCS ([Mil80]). Ainsi, sa syntaxe est écrite en algèbre, à l'aide d'expressions mathématiques appelées *comportements*, alors que du point de vue sémantique, chaque comportement représente un automate d'états finis ou infinis.

En résumé, malgré une syntaxe considérée parfois comme "lourde" comparée par exemple à la syntaxe graphique de SDL, sa rigueur permet d'utiliser des outils d'analyse et des outils de compilation performants. Par exemple, il est possible actuellement de générer du code C d'assez bonne qualité. Toutefois, il faut reconnaître que LOTOS ne permet pas de décrire les aspects quantitatifs ou les problèmes de contraintes de temps, ce qui en fait un modèle difficile à utiliser pour spécifier des systèmes temps-réel. Des approches adaptées pour le temps réel ont été élaborées, comme ET-LOTOS ([LL97]) ou RT-LOTOS ([CO95]).

SDL

Normalisé par l'ISO et le CCITT, SDL (Specification and Description Language) est aussi un langage de spécification asynchrone, de haut niveau. La première version date de 1972, mais de nombreuses améliorations ont été apportées depuis, jusqu'à la dernière version connue, décrite dans [Mam01] (2001). Il se fonde sur les principes orienté objet. Ce langage est prévu pour spécifier des comportement de systèmes ayant des activités parallèles et communicantes, ce qui explique que son utilisation se fasse majoritairement dans le domaine des télécommunications.

[3]Language Of Temporal Ordering of Specification

SDL est composé de blocs, connectés par des canaux. Chaque bloc peut contenir un ou plusieurs processus. Il offre le choix entre deux formes syntaxiques différentes pour représenter un système : une forme graphique (SDL/GR "SDL Graphical Representation") et une forme textuelle (SDL/PR "SDL Phrase Representation"). Ce langage est fondé sur la notion de machines à états finis étendues. Un processus est une machine états finis étendue autonome, qui s'exécute de manière concurrente avec les autres processus. Ils communiquent de manière asynchrone à l'aide de canaux (ou de routes si la communication se fait entre processus d'un même bloc). Contrairement à LOTOS, il n'y a pas de synchronisation entre processus : les signaux (correspondant à un flot d'informations) envoyés sont stockés dans un file d'attente (supposée infinie). Afin de gérer quelques aspects temporels, SDL permet de lire le temps courant et de définir, armer, annuler des timers. Cela offre des meilleures possibilités pour spécifier des systèmes avec contraintes de temps, et permet ainsi de décrire certains systèmes temps-réel.

2.2 Systèmes synchrones

A partir de cette section, nous allons voir des formalismes adaptés pour prendre en compte les aspects temporels de systèmes, tels que les systèmes temps-réel. Contrairement à certaines idées reçues, un système temps-réel n'est pas jugé par rapport à sa rapidité de réaction. Par définition, c'est un système dans lequel les contraintes de temps doivent être obligatoirement respectées. Ces contraintes peuvent être plus où moins larges selon les situations. Voici une définition plus générale, écrite par Jean-Pierre ELLOY (1991) :

Définition 2.2.1 (temps-réel) *On qualifie de* **temps-réel** *une application mettant en oeuvre un système informatique dont le comportement est conditionné par l'évolution dynamique de l'état du procédé qui lui est connecté.*
Ce système informatique est alors chargé de suivre ou de piloter ce procédé en respectant des contraintes temporelles définies dans le cahier des charges de l'application.

2.2.1 Aperçu historique des modèles synchrones

Une étude complète sur les langages synchrones peut être trouvée dans [PRBP94]. Les langages synchrones comme Lustre, Esterel et Signal ont été créés vers le milieu des années 80. En parallèle, Harel et Pnueli travaillaient sur un langage presque synchrone : Statecharts, utilisé pour les spécifications de systèmes temps-réel embarqués (projet StateMate).

Au début des années 90, F. Maraninchi propose une nouvelle sémantique pour une nouvelle version des StateCharts, appelée Argos. Ensuite, certains aspects sont apparus clairement : Argos et StateCharts ne permettaient pas de spécifier qu'une implantation doit satisfaire des contraintes temps-réel; la notion de raffinement n'y était pas envisagée, et enfin l'aspect objet n'y existait pas Par ailleurs, on s'est rendu compte que Esterel/Lustre (E/L) d'un côté, StateMate de l'autre, avaient des utilités différentes. E/L servent pour spécifier un automate temporisé minimal pour contrôler un composant dans un système embarqué temps-réel, alors que StateMate sert plutôt à spécifier un système

distribué en entier, avec un certain nombre de composants répartis géographiquement. Ainsi, leur rôle complémentaire était accepté. Ensuite, l'équipe de J. Sifakis (Verimag) a proposé de donner la future spécification d'un système pour Airbus à l'aide de Argos/StateMate pour la spécification haut niveau, et avec E/L pour la spécification bas niveau de chaque composant. Après cela, la description des combinaisons entre les deux formalismes est devenue un enjeu de recherche. En parallèle, des systèmes de preuve ont été développés, et des approches ont été faites pour spécifier des systèmes temps-réel (Huizing, Pnueli).

2.2.2 Evolution des modèles synchrones

F. Maraninchi étudie dans [PRBP94] les mécanismes de communication synchrone (rendez-vous, réactions en chaînes dans les StateCharts) en exprimant leur sémantique dans un framework général où des *modèles d'information maximale* sont associés aux programmes. Les modèles sont des LTS (Labeled Transition System) avec des labels structurés.

Charles André propose un nouveau modèle de représentation graphique synchrone, les SyncCharts ([And96]), qui est proche des StateCharts, mais qui possède l'avantage d'être totalement compatible avec les hypothèses synchrones, et qui peut donc être traduit systématiquement en Esterel. Finalement, [ABP+97] introduit la notion d'objet synchrone.

2.2.3 Esterel vs Lustre

Esterel

Esterel est un langage impératif synchrone modulaire qui possède une sémantique mathématique rigoureuse. Il a été développé à l'école des Mines à Sophia Antipolis. Une description complète peut être trouvée dans [Ber97]. Son utilité principale est de spécifier ou programmer des systèmes réactifs, c'est à dire "des systèmes qui sont en interaction permanente avec leur environnement et dont le rythme est imposé par cet environnement" ([HP85]). Esterel est surtout adapté pour les systèmes réactifs à contrôle prépondérant.

Une spécification Esterel s'écrit sous forme textuelle. On y définit des *modules* qui vont communiquer entre eux à l'aide de *signaux* valués instantanés. La particularité d'Esterel c'est que l'on peut écrire facilement différentes tâches ou modules, qui ont la possibilité de s'exécuter en parallèle. En général, il sert à décrire la partie contrôle d'une application, le reste étant écrit directement en C ou ADA.

Ainsi, une application est un ensemble de modules, chacun d'eux étant constitué d'une interface (signaux d'entrée et de sortie) et d'un corps (suite d'instructions exécutées de façon séquentielle). Il existe deux types d'instructions :

- instructions logiques : temps d'exécution nul (if/then/else, boucles, affectations, etc ...)
- instructions temporelles : temps d'exécution non nul

L'avantage d'Esterel, c'est qu'il permet de spécifier bon nombre de systèmes temps-réel de façon concise, et ce grâce à l'hypothèse synchrone et au parallélisme. Sa sémantique permet la mise en place d'outils d'analyse, de vérification ou de simulation performants. On notera qu'il faut un certain temps d'adaptation au langage, notamment en ce qui concerne les "paradoxes" induits par l'hypothèse synchrone ou encore par rapport à la gestion concurrente des ressources.

Par ailleurs, ce sont les machines de Mealy qui servent de support à la spécification des comportements de programmes Esterel pur. Ainsi, il est possible de décrire les programmes Esterel à l'aide d'une algèbre de processus. Jusqu'à la version 3 du programme, le compilateur générait systématiquement cet automate. A partir de la version 4, le compilateur génère un système d'équations booléennes. [AD01] propose d'utiliser une sémantique constructive pour éviter certaines hypothèses sur les signaux, hypothèses obligatoires si l'on utilise une sémantique comportementale logique.

Lustre

Lustre est un langage déclaratif synchrone à flots de données, défini à l'IMAG (Grenoble). Une description complète se trouve dans [CPHP87]. Encore une fois, il est utilisé pour la spécification et la programmation de systèmes réactifs, comme le contrôle automatique ou encore la description d'aspects matériels (circuits, ...). Il permet aussi de décrire des propriétés d'un programme. Il est particulièrement performant pour spécifier les systèmes réactifs à traitement dominant. L'aspect "flot de données" de Lustre le rend assez proche des outils de description connus (diagrammes-blocs, réseaux d'opérateurs, etc...), et son interprétation synchrone facilite la gestion du temps et permet de le compiler en programme séquentiel.

En Lustre, chaque variable ou expression concerne un *flot*, qui est une paire contenant :
- une séquence (éventuellement infinie) de valeurs avec un type donné
- une horloge, représentant une séquence d'instants

Un flot prend la n-ième valeur de sa séquence de valeurs au n-ème instant de son horloge. C'est le cycle d'exécution d'une horloge dite de base qui sert de référence à toutes les autres horloges. Par ailleurs, on utilise des variables, définies sous forme d'équations, en sachant que Lustre applique le principe de substitution.

Lustre est un outil utilisable pour la spécification où la programmation de systèmes réactifs. Grâce encore une fois aux fondements théoriques de sa sémantique, il est directement utilisable aussi bien pour la vérification, la preuve de programme, la simulation ou encore pour la génération de code.

2.3 Systèmes temporisés asynchrones

Les modèles présentés dans cette section considèrent le temps comme un élément continu. Cette hypothèse est certes plus proche de la réalité mais cela implique en général des raisonnements plus difficiles et des algorithmes avec des complexités bien plus élevées.

2.3.1 ACSR - GCSR

L'algèbre ACSR

L'algèbre ACSR [4] ([CL97]) est une algèbre de processus temporelle, basée sur le modèle de synchronisation CCS ([Mil80]). Elle offre des possibilités pour représenter la synchronisation, le temps, les portées temporelles, les ressources et les priorités. Elle est fondée sur le principe qu'un système temps réel consiste en un ensemble de *processus* qui communiquent de manière asynchrone, et qui s'exécutent sur un ensemble fini de ressources réutilisables, et qui se synchronisent avec d'autres par des événements à travers des canaux de communication. L'utilisation des ressources partagées est représentée par des *actions* qui consomment du temps et de la ressource, et enfin la synchronisation est gérée par des *événements* instantanés.

Plus formellement, la syntaxe ACSR est définie par la grammaire suivante :

$$P ::= NIL | A^t : P | (e,p).P | P_1 + P_2 | P_1 \parallel P_2 | P\Delta_t{}^b(P_1, P_2, P_3) | [P]_I | P\backslash\backslash I | P\backslash F | C$$

Pour pouvoir spécifier efficacement des systèmes temps réel, l'algèbre ACSR supporte des *priorités statiques* permettant de faire l'arbitrage entre des actions en concurrence sur des ressources partagées, ou entre des événements prêts pour la synchronisation. Elle offre aussi différentes notions d'équivalence utiles en particulier pour vérifier que deux spécifications se comportent de la même manière.

Le détail complet de cette grammaire est donné dans [CL97], c'est pourquoi nous allons voir uniquement les opérateurs les plus fréquents. NIL est un processus qui n'exécute aucune action. Il y a deux opérateurs à préfixe qui correspondent à deux types d'actions. La première, $A^t : P$, exécute une action temporisée et qui consomme de la ressource, A, pour $t \in \mathbb{N}$ unités de temps, et passe ensuite au processus P. Le second opérateur préfixé, $(e,p).P$ exécute l'événement instantané e avec le niveau de priorité p, et passe ensuite à P. L'opérateur $P_1 + P_2$ représente la possibilité de choisir P_1 ou P_2. L'opérateur $P_1 \parallel P_2$ est l'exécution concurrente de P_1 et P_2. Le scope $P\Delta_t{}^b(P_1, P_2, P_3)$ lie le processus P à un scope temporel, et incorpore les notions de timeout et d'interruption. P peut exécuter au maximum $t \in \mathbb{N} \cup \{\infty\}$. On peut sortir de ce scope de trois façons. La première, si P se termine avec succès avant t en exécutant un événement $\bar{b}$, alors le contrôle procède à l'exception handler P_1. La deuxième, si P ne peut se terminer avant t, alors le contrôle procède au "timeout-handler" P_2. Enfin, à n'importe quel moment pendant son exécution, P peut être interrompu par P_3.

ACSR a été implémenté dans les outils VERSA ([CLX95]) et PARAGON ([DBALS97]). VERSA a trois principales fonctions : réécriture, test d'équivalence, et exécution interactive. PARAGON fournit un environnement graphique ACSR pour spécifier les systèmes temps-réel.

[4] Algebra of Communicating Shared Resources

GCSR

Dans le même esprit, GCSR[5] est un langage formel pour la spécification, le raffinement, et l'analyse de systèmes temps réel ([BALC95]). GCSR permet d'intégrer les besoins fonctionnels (exécution concurrente, communication, contraintes de temps, etc...) ainsi que les besoins au niveau de ressources (cpu, mémoire, capteurs, etc...). La sémantique de GCSR est définie soit directement par des LTS, soit indirectement par une translation vers ACSR.

2.3.2 Automates Temporisés

Alur et Dill ont proposé dans [AD94] une modélisation des systèmes temporisés qui décrit le temps de manière continue grâce à un ensemble d'*horloges*. Elle permet de décrire la plupart des systèmes temporisés. Ce modèle a inspiré beaucoup d'études, notamment sur la vérification et le test.

Un *automate temporisé* (timed automaton (TA)), c'est un LTS auquel est associé un ensemble d'horloges. Chaque horloge est évaluée par un réel (représentation dense du temps) et évolue avec le temps. Elles évoluent toutes à la même vitesse. A chaque instant, la lecture d'une horloge donne le temps écoulé depuis la dernière fois qu'elle a été réinitialisée. Elles sont initialisées à 0 dans l'état initial de l'automate, mais peuvent être réinitialisées avec chaque transition. Ces transitions peuvent par ailleurs contenir des contraintes sur ces horloges. Ainsi, pour qu'une transition soit franchie, toutes les horloges du système doivent satisfaire ces contraintes. Nous rappelons ici les définitions essentielles et en particulier celle d'automate temporisé, extraites de [AD94].

Définition 2.3.1 (Contraintes d'horloge) *Pour un ensemble X d'horloges, l'ensemble $\Phi(X)$ de contraintes d'horloge δ est défini de façon inductive par :*

$$\delta := x \leq c \,|\, c \leq x \,|\, \neg\delta \,|\, \delta_1 \wedge \delta_2,$$

où x est un horloge de X et c une constante de $\mathbb{Q}$.

Définition 2.3.2 (Interprétation d'horloge) *Une interprétation d'horloge ν sur un ensemble X d'horloges donne une valeur réelle pour chaque horloge, c'est une fonction de X dans $\mathbb{R}$. On dit qu'une interprétation ν pour X satisfait une contrainte d'horloge δ ssi δ donne la valeur vrai à partir des valeurs données par ν.*
Pour $t \in \mathbb{R}$, $\nu + t$ dénote l'interprétation d'horloge correspondant à la valeur $\nu(x) + t$, et l'interprétation $t.\nu$ correspond à la valeur $t.\nu(x)$.
Pour $Y \subseteq X$, $[Y \mapsto t]\nu$ dénote l'interprétation d'horloge sur X qui assigne t pour chaque $x \in Y$, et donne la valeur de ν pour les horloges restantes.

Définition 2.3.3 (Automate temporisé (TA)) *Un automate temporisé est un quintuplet $< \Sigma, S, S^0, C, E >$ où :*

[5] Graphical Communicating Shared Resources

- Σ *est un alphabet fini,*
- S *est un ensemble fini d'états,*
- $S^0 \subseteq S$ *est l'ensemble des états initiaux,*
- C *est un ensemble fini d'horloges,*
- $E \subseteq S \times S \times \Sigma \times 2^C \times \Phi(C)$ *est l'ensemble des transitions. Une arête* $< s, s', a, \lambda, \delta >$ *représente une transition de l'état* s *vers l'état* s' *sur l'occurrence du symbole* a*. L'ensemble* $\lambda \subseteq C$ *donne les horloges qui doivent être réinitialisées avec la transition, et* δ *est une contrainte d'horloge sur* C*.*

Notons que par la suite, nous n'utiliserons que des automates temporisés avec un seul sommet initial. Dans certains modèles, il est possible d'avoir ce qu'on appelle des ϵ-transitions. Une action ϵ signifie qu'aucune information n'est émise ou reçue lors du passage de la transition : c'est une *action interne*. Il est assez fréquent aussi de voir un ensemble d'états finaux dans la définition d'automate temporisé, ce qui permet ensuite de travailler sur l'acceptation de langages temporisés.

La figure 2.4 donne un exemple d'automate temporisé :

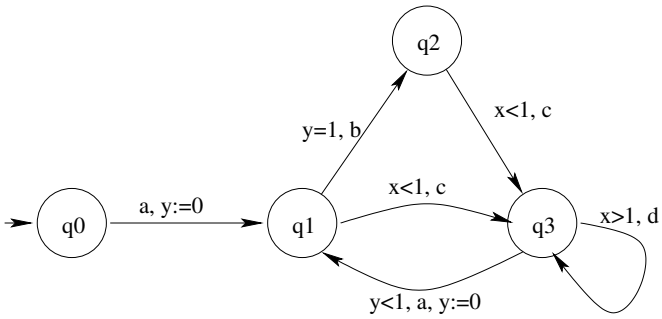


FIG. 2.4 – Un exemple d'automate temporisé

Toutefois, ce modèle ne permet pas de faire de différence entre l'émission ou la réception d'une action. Or cette distinction s'avère parfois nécessaire, en particulier il peut être utile de savoir si la passage d'une transition se fera à l'initiative de l'environnement du système (c'est une entrée), ou si au contraire c'est le système lui-même qui déclenche le passage (c'est une sortie). Pour exprimer cette information, une extension des automates temporisés a été définie : le TIOA (Timed Input Output Automaton ou automate temporisé à entrées / sorties).

Définition 2.3.4 (TIOA) *Un TIOA est un automate temporisé* $A =< \Sigma_A, S_A, S^0_A, C_A, E_A >$ *où :*

- S^0_A *est un singleton*
- Σ_A *se partitionne en deux sous-ensembles distincts :*
  - $\mathcal{I}_\mathcal{A}$ *l'ensemble des symboles d'entrée (on les écrit précédés d'un " ?")*
  - $\mathcal{O}_\mathcal{A}$ *l'ensemble des symboles de sortie (on les écrit précédés d'un " !")*

La définition du déterminisme des automates temporisés sera vue ultérieurement. La figure FIG. 2.5 nous donne cette fois un exemple de TIOA.

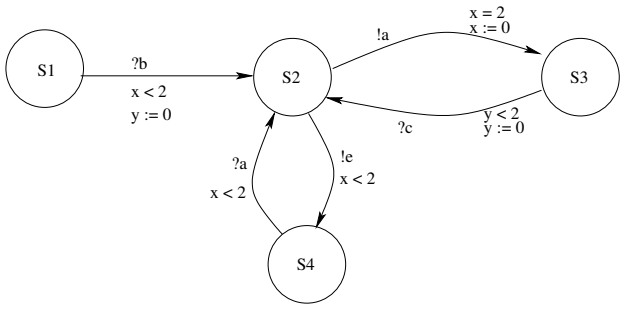


FIG. 2.5 – Un exemple de TIOA

Notre système (FIG. 2.5) comporte deux horloges : x et y. Supposons que le système soit en S1. A un moment donné, il reçoit le signal b. Pour pouvoir passer à l'état S2, il faut absolument que toutes les contraintes d'horloges soient vérifiées, soit ici x<2. Si c'est le cas, on passe à l'état S2, et on réinitialise l'horloge y, comme l'impose la transition.

Ainsi, le comportement d'un système temporisé dépend de ses états et des valeurs d'horloges appelées les valuations (concrètement, la valuation des horloges, c'est l'ensemble des coordonnées dans un espace de dimension n de chacune des n horloges du système). Par la suite, nous serons amenés à manipuler essentiellement des TIOA.

Patricia Bouyer étudie dans [Bou02] une extension des automates temporisés permettant la mise à jour des horloges avec n'importe quelle valeur. Ce sont les automates temporisés avec mise à jour. Dans les travaux sur la vérification, cet ajout entraîne des problèmes d'indécidabilité.

Une étude de différentes variantes d'automates temporisés a été réalisée lors du projet CALIFE ([Cal00]). A la suite de cette étude, une généralisation des automates temporisés, appelés *p-automates* a été proposée, permettant de modéliser des systèmes réalistes à différents niveaux d'abstraction au sein d'un même cadre formel. Ils permettent aussi les mises à jour d'horloges. Une motivation majeure pour l'élaboration de ces automates était de régler le problème d'explosion combinatoire dans les problèmes de vérification (model-checking).

2.3.3 les DBM : une représentation matricielle des contraintes

Pour représenter les contraintes d'horloge, il est possible de définir des zones, qui sont en fait des sous-ensembles de $\mathbb{Q}^n$. Ensuite, ces zones peuvent être représentées dans des structures de données, appelées *DBM* ([Dil90]). Une *matrice de différences bornées (DBM)* pour n horloges est une matrice carrée $(m_{i,j}, \prec_{i,j})_{i,j=0\ldots n}$ de taille $n+1$ de paires

$$(m; \prec) \in \mathbb{V} = (\mathbb{Z} \times \{<, \leq\}) \cup \{\infty; <\}.$$

Une DBM $M = (m_{i,j}, \prec_{i,j})_{i,j=0\ldots n}$ définit l'ensemble suivant de $\mathbb{Q}^n$ (l'horloge x_0 est supposée être constamment égale à zéro, ie pour toute valuation ν, $\nu(x_0) = 0$) :

$$\{\nu : \{x_1, ...x_n\} \to \mathbb{Q} \,|\, \forall\, 0 \leq i, j \leq n, \nu(x_i) - \nu(x_j) \prec_{i,j} m_{i,j}\}$$

où $\gamma < \infty$ signifie que γ est un nombre quelconque. Ce sous-ensemble de $\mathbb{Q}^n$ est une zone et est noté $[M]$. Chaque DBM sur n horloges représente une zone de $\mathbb{Q}^n$. Remarquons que deux DBM (écrites différemment) peuvent représenter la même zone. Des outils de vérification comme UPPAAL ([LPY97]) et KRONOS ([Yov97]) utilisent cette représentation.

Le zone définie par les équations $x_1 > 3 \wedge x_2 \leq 5 \wedge x_1 - x_2 < 4$ peut être représentée par la DBM suivante :

$$\begin{pmatrix} (0;\leq) & (-3;<) & (\infty;<) \\ (\infty;<) & (0;\leq) & (4;<) \\ (5;\leq) & (\infty;<) & (0;\leq) \end{pmatrix}.$$

2.3.4 Graphes des régions

Certaines manipulations sur les aspects temporels sont assez difficiles lorsqu'on travaille sur des automates temporisés. Le problème vient du fait que l'ensemble des valeurs d'horloges est infini. De même, si l'on veut connaître la valuation d'une horloge à un moment donné de l'exécution, il est nécessaire de connaître précisément tout le parcours effectué dans l'automate temporisé. Pour offrir une solution à ces problèmes, [AD94] propose de regrouper les valeurs des horloges dans des espaces de dimension n, n étant le nombre d'horloges, appelés régions. L'idée, c'est que toutes les informations concernant le temps soient disponibles directement sur l'état atteint, grâce à sa région. C'est une sorte d'intervalle de temps pendant lequel une action peut être exécutée. Ainsi, les régions prennent en compte à la fois les contraintes d'horloges, les remises à zéro mais aussi l'écoulement du temps. Pour cela, Alur et Dill définissent une région "successeur" d'une autre si l'écoulement du temps permet de passer de la première à la deuxième. Nous n'allons pas ici redonner toutes les définitions, disponibles dans [AD94], nous allons juste donner un exemple de graphe des régions. La figure FIG. 2.6 donne le graphe des régions équivalent à l'automate temporisé de la figure FIG. 2.4.

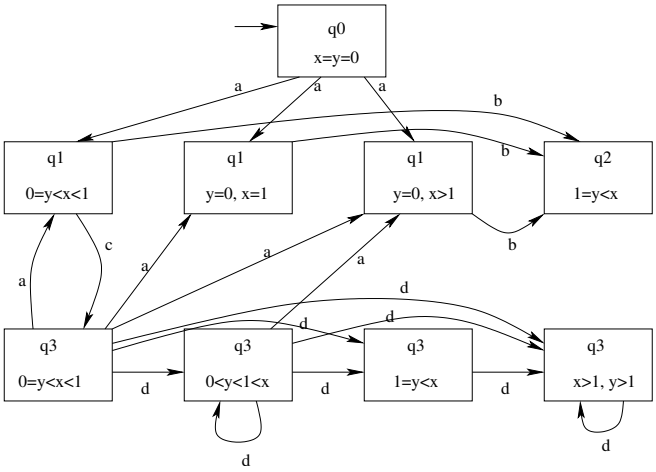


FIG. 2.6 – Un exemple de graphe des régions

Par exemple, si on étudie les états $q1, y = 0, x > 1$ et $q2, 1 = y < x$, en les nommant respectivement $Q1$ et $Q2$ dans le graphe des régions, on remarque que le passage de $Q1$ à $Q2$ ne peut se faire que sur l'action b. Si on regarde sur l'automate correspondant FIG. 2.4, le passage de $q1$ à $q2$ n'est possible que si y vaut 1, ce qui explique que $y = 1$ dans $Q2$. On constate aussi que dans toutes les transitions menant à $q1$, l'horloge y est réinitialisée, ce qui implique qu'au passage de $q1$ à $q2$ on ait obligatoirement $y < x$. Cette inégalité est donc retranscrite dans $Q2$.

C'est dans le domaine de la vérification que ce modèle est le plus répandu. On en trouve aussi beaucoup dans le test. Il est vrai que les régions d'horloges en font un modèle confortable et performant pour les théories en rapport avec le temps, mais le passage de l'automate temporisé vers le graphe des régions est une opération coûteuse (un algorithme de transformation se trouve dans [AD94]), et il n'est pas envisageable d'un point de vue pratique de demander au concepteur d'un système de donner directement une spécification sous forme de graphe des régions.

2.3.5 Automates à événements d'horloge

La théorie des automates temporisés permet de résoudre certains problèmes de vérification pour des systèmes temps-réel avec contrôle fini, ainsi que certains problèmes avec délai. Cependant, le problème général de la vérification est indécidable pour les automates temporisés. Le problème vient du fait que ces automates peuvent être non-déterministes, et qu'ils permettent dans ce cas d'exprimer beaucoup plus de situations que les déterministes, contrairement aux automates non-temporisés. Ainsi, il est difficile d'envisager les automates temporisés comme modèle canonique pour les calculs sur modèles temps-réel. C'est pourquoi [AFH94] propose de restreindre l'utilisation des horloges, afin d'obtenir une classe déterminisable d'automates temporisés. Les horloges d'un *automate à événements d'horloge* sont associées de façon fixe et prédéfinie avec un symbole de l'alphabet d'entrée (qui représente les événements). [AFH94] définit ainsi l'horloge *event-recording* du symbole d'entrée a, notée x_a, qui est une variable à mémoire, ie dont la valeur est toujours égale au temps écoulé depuis la dernière occurrence de a par rapport au temps courant, et l'horloge *event-predicting*, notée y_a, qui est une variable "prophétie", ie dont la valeur est toujours égale au temps séparant l'instant courant de la prochaine occurrence de a. Ainsi, contrairement aux automates temporisés, un automates à événements d'horloge ne contrôle pas les réaffectations de ses horloges, et à chaque symbole d'entrée, les valeurs des horloges sont déterminées uniquement grâce au mot entré (ie la séquence de symboles entrée). Cette propriété permet leur déterminisation.

Concrètement, un automate à événements d'horloge est une FSM (non déterministe) dont les transitions sont étiquetées à la fois avec un symbole de l'alphabet, et avec les contraintes d'horloges, concernant les horloges *event-recording* et *event-predicting*.

Définition 2.3.5 (Automate à événements d'horloge) *Un automate à événements d'horloge est un 6-uplet* $< \Sigma, L, L^0, L^f, E >$ *où :*

- Σ *est l'alphabet d'entrée*
- L *est un ensemble d'états*
- $L^0 \subseteq L$ *est l'ensemble des états initiaux*

- $L^f \subseteq L$ *est l'ensemble des états acceptants*
- E *est l'ensemble des arêtes. Chaque arête est un quadruplet* $< l, l', a, \varphi >$ *avec un état de départ* $l \in L$*, un état d'arrivée* $l' \in L$*, un symbole d'entrée* $a \in \Sigma$ *et une contrainte d'horloge* φ *sur l'ensemble des horloges* C*.*

La figure 2.7 donne un exemple d'automate à événements d'horloge.

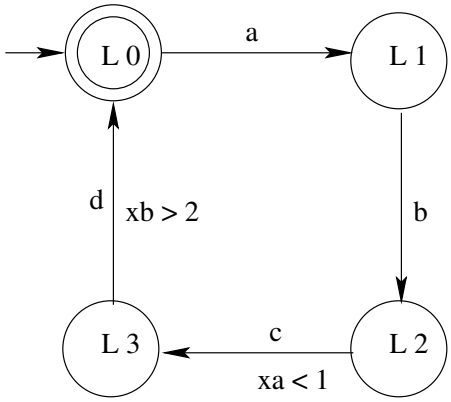


FIG. 2.7 – Automate à événements d'horloge $A1$

Sur cette figure, l'automate $A1$ utilise deux horloges *event-recording*, x_a et x_b. Le sommet l_0 est le sommet de départ de $A1$, mais aussi le seul sommet "acceptant". L'automate accepte en entrée les mots temporisés de la forme $(\bar{a}, \bar{t})$ avec $\bar{a} = (abcd)^k$, pour $k \geq 0$. Toutes les transitions non-étiquetées par une horloge ont par défaut la contrainte d'horloges vraie. La contrainte $x_a < 1$ associée à la transition $l2$ à $l3$ assure que chaque c apparaît moins de une unité de temps après le a précédent. De même, $x_b > 2$ en lisant d assure que chaque d après un b a lieu à plus de deux unités d'écart.

La définition du déterminisme d'un automate à événements d'horloge est analogue à celle des automates temporisés. Par ailleurs, [AFH94] montre que les automates à événements d'horloge sont aussi expressifs qu'ils soient déterministes ou pas. Dans la même idée, [AFH94] montre que pour tout automate à événements d'horloge A, il existe un automate à événements d'horloge déterministe définissant le même langage temporisé que A. Toutefois, cette déterminisation provoque une augmentation exponentielle du nombre de sommets, mais ne change en rien le nombre d'horloges ou bien les valeurs des constantes dans les contraintes d'horloges. Par ailleurs, il existe une construction de régions analogues à celles vues précédemment, à partir des automates à événements d'horloge. L'algorithme est donné dans [AFH94].

2.3.6 Automates hybrides

Les automates hybrides permettent de décrire des changements d'état qui peuvent se faire soit à des instants déterminés, de façon discrète, soit de façon continue (avec l'évolution des variables). Les systèmes décrits par ce type de modèle ont des paramètres qui évoluent de façon continue, et des événements discrets. En fait, l'exécution d'un système

hybride est une séquence d'étapes. Tout au long d'une étape, le système évolue de façon continue en suivant une loi, jusqu'à ce qu'une transition soit tirée. Les transitions sont des changements instantanés d'état qui séparent les évolutions continues de l'état.

On modélise donc un système hybride comme un automate fini, avec un ensemble de variables. Les états de contrôle de l'automate sont étiquetés avec des lois d'évolution. Sur un état, les valeurs des variables évoluent de façon continue avec le temps, en accord avec la loi associée. Les transitions de l'automate sont étiquetées avec une garde et des affectations. Une transition est autorisée uniquement si sa garde associée est vraie, et son exécution modifie les valeurs des variables en accord avec les affectations. Chaque état est aussi étiqueté par une condition d'invariant, qui doit toujours être vérifiée lorsque le contrôle reste sur cet état. La définition formelle d'un automate hybride se trouve dans [ACH+95]. En ce qui nous concerne, nous allons l'illustrer à travers un exemple, celui de la figure 2.8.

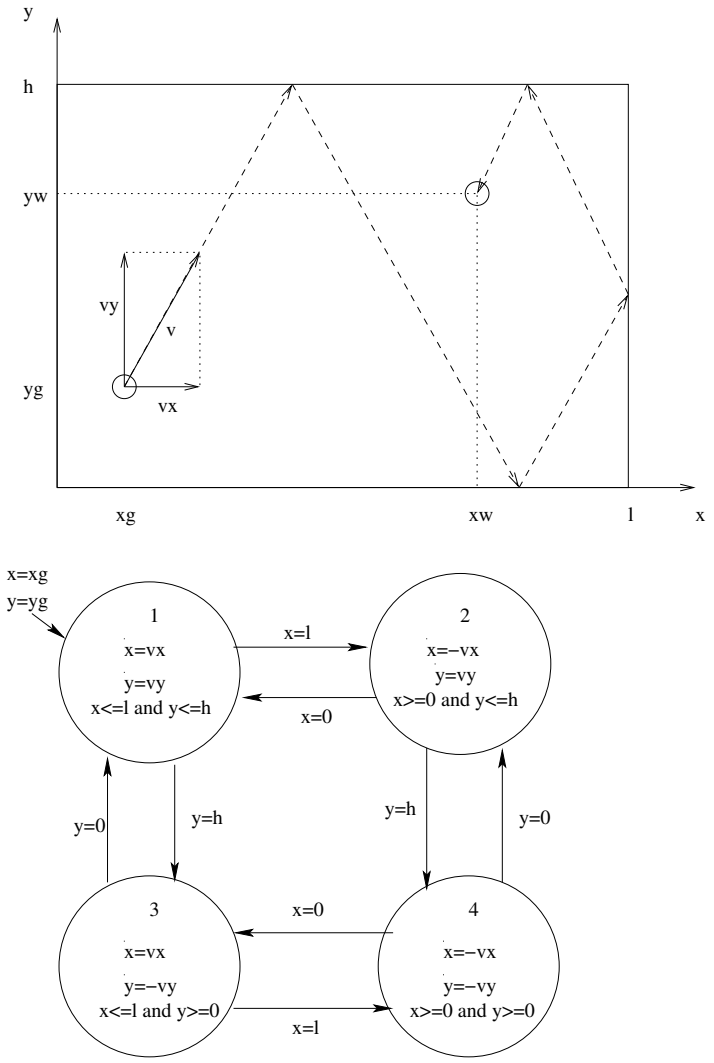


FIG. 2.8 – Jeu de billard et sa modélisation en automate hybride

Au départ, la bille se trouve en coordonnées $x = x_g$ et $y = y_g$. A l'état 1, la vitesse de la bille est (v_x, v_y). Etant donné la direction prise, on vérifie uniquement que $x \leq l$ et $y \leq h$. Lorsqu'on arrive au bord, on a $x = l$. Dans ce cas, on peut passer la transition et arriver à l'état 2. Etant donné les lois de la mécanique concernant les interactions entre objets, la vitesse de la bille devient $(-v_x, v_y)$ tout au long de cet état, c'est à dire tout au long de ce segment de chemin. Dans ce trajet, on doit toujours avoir $x \geq 0$ et $y \leq h$. Et ainsi de suite si $y = h$ ...

On remarque que les automates temporisés sont un sous-ensemble des automates hybrides. Par ailleurs, actuellement, des travaux ont été réalisés sur la vérification des systèmes hybrides, notamment dans [ACH+95]. Pour ce qui est du test, c'est un domaine encore peu exploité. Ce modèle permet de décrire une quantité impressionnante de comportements réels. Toutefois, les problèmes théoriques introduits par l'ajout de lois de commande sur les états en font un modèle difficile à mettre en oeuvre.

2.3.7 UML - UML-RT

Depuis sa normalisation en 1997 par l'Object Management Group (OMG), UML (1.0) est devenu un des langages de spécification les plus utilisés. Toutefois, il s'est vite avéré que ce genre de modélisation était insuffisante dans le cadre du développement d'applications temps-réel. En effet, il manque une façon de définir les interdépendances entre hardware et software, la description des contraintes de temps, des structures de communication, et des politiques de gestion de tâches. A cause de ces lacunes, UML a été utilisé dans le domaine temps-réel de façon combinée avec d'autres techniques comme SDL ou ROOM ([Gom01]). La version 1.4, publiée en 2001, avait pour but principal de corriger certaines erreurs typographiques ou grammaticales, ou de clarifier certains aspects.

ROOM[6] est un langage de description d'architecture utilisé principalement dans l'industrie des télécommunications pour développer des systèmes embarqués. Les concepts de ROOM ont été incorporés dans Rational Rose Real-Time (RoseRT), sous la forme de "profil UML", appelé en général UML-RT. Les concepts clé de UML-RT ([Sel98]) sont les suivants :

- Une *capsule* est un objet actif, avec son propre thread logique de contrôle. Une capsule contient une description du comportement sous la forme d'une version objet des Statechart, appelée *ROOMCharts*. Les machines concurrentes sont modélisées par des capsules séparées, qui communiquent par message de façon asynchrone.
- Une représentation explicite des *ports, protocoles* et *connecteurs* permet des constructions de capsules sous forme d'architecture.
- Un *framework d'exécution* appelé *TargetRTS*

Pour chaque capsule, le modèle d'exécution suit la sémantique *Run-To-Completion* (RTC). Une fois qu'elle est activée par un message sur un port d'entrée, la capsule doit exécuter l'action complètement avant d'exécuter le message suivant. Les messages sont stockés dans une file, et peuvent avoir différents niveaux de priorité.

Ainsi, UML-RT est une extension d'UML basée sur les concepts de ROOM. En utilisant les mécanismes d'extension intégrés dans UML, les notions de capsules, ports, connec-

[6] Real-Time Object-Oriented Modeling

teurs, protocoles, et rôles de protocoles ont été introduites dans UML-RT pour permettre la modélisation de systèmes complexes temps-réel. Pour représenter l'architecture de ces systèmes, un diagramme de collaboration entre capsules (appelé diagramme d'architecture) est défini. Il permet de représenter les composants du système et comment ils sont connectés. Un exemple de diagramme d'architecture ([KS01]) est donné figure FIG. 2.9.

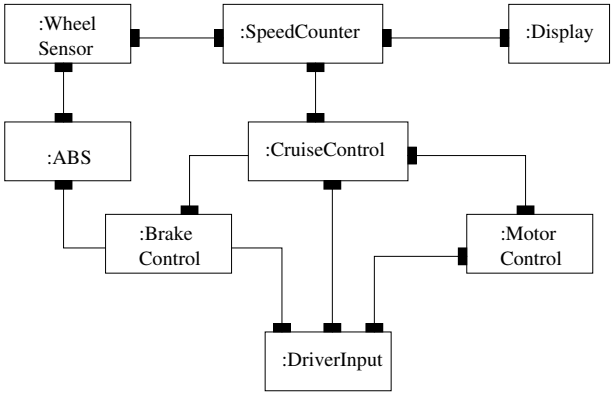


FIG. 2.9 – Un exemple de diagramme d'architecture en UML-RT

Une *capsule* est une classe spécialisée active et est utilisée pour modéliser un composant autonome. Contrairement aux classes ordinaires, les opérations d'une capsule ne peuvent être appelées que depuis l'intérieur de cette capsule. Afin de pouvoir communiquer avec les autres capsules, une capsule peut avoir un ou plusieurs ports, à travers lesquels elle est interconnectée à d'autres capsules par des *connecteurs*. Ces derniers représentent des connexions hardware via lesquelles les capsules communiquent en envoyant et recevant de messages. Le comportment d'une capsule est décrit sous forme de diagramme statechart :

Définition 2.3.6 (Diagramme statechart) *Un diagramme statechart UML est un 4-uple* $StatD = (S, E, A, t)$ *tel que :*

- S *est un ensemble fini, non vide, d'états.*
- E *est un ensemble d'événements.*
- A *est ensemble fini d'actions.*
- t *est la relation de transition avec* $t : S \times E \times A \rightarrow S$*, noté* $(s_i, l_i, s_j) \in t$ *où* $l_i \in E \times A$ *dénote l'événement et l'action d'une transition.*

La nouvelle version UML 2.0 ([OMG03a], [OMG03b]), publiée en 2003, tend à corriger les insuffisances des versions précédentes, et permet aussi le développement de systèmes à base de composants. Elle prend en compte les contributions de UML-RT, proposant ainsi :

- des profils : ils sont utilisés pour étendre UML avec des éléments spécifiques au domaine d'utilisation. Ainsi, des profils d'ordonnancement, de performance et de temps ont été créés.
- une modélisation structurelle, qui offre de nouveaux concepts comme la composition hiérarchique de modèles, des structures de communication et des composants.

– une modélisation comportementale : gestion des actions grâce aux diagrammes d'activité, gestion des interactions.

Même si la plupart des lacunes de UML 1.4 ont été corrigées dans UML 2.0, et notamment le composition hiérarchique qui permet de construire des modèles consistants, il reste encore des améliorations à faire comme donner une spécification formelle et non informelle de UML, avec une sémantique, et trouver une solution pour réduire le nombre impressionnant d'éléments qui compliquent l'utilisation. Le manque de spécification formelle en fait un modèle difficile à utiliser pour le vérification et le test formel.

Les lacunes des versions antérieures d'UML ont incité en parallèle divers développements de profils UML pour le temps-réel. On peut citer TURTLE ([ACLdSS04]) qui étend les diagrammes de classes et les diagrammes d'activité par des opérateurs de composition et des opérateurs temporels. Sa sémantique formelle est exprimée en RT-LOTOS. En fait, TURTLE fournit un outil de conception de système temps-réel, qui offre en plus des possibilités de validation.

Chapitre 3

Méthodes de test

Comme nous l'avons vu précédemment, il existe une grande quantité de méthodes de validation d'un système. Nous avons volontairement séparé la validation en deux catégories distinctes : la vérification et le test. Toutefois, il existe de nombreux points communs entre ces deux techniques. En effet, que ce soit pour vérifier ou pour tester, il est nécessaire de s'appuyer sur ce que l'on connaît du système, à savoir son comportement attendu. Ainsi, dans ces deux étapes, c'est généralement la spécification du système qui sert de base au travail. Par conséquent, le choix du modèle pour décrire la spécification aura une incidence à la fois sur la phase de vérification, mais aussi sur la phase de test.

Tester la conformité d'un système, c'est regarder si le comportement observé de l'Implantation sous Test (par la suite appelée *IUT* pour *Implementation Under Test*) est conforme à celui décrit dans la spécification. Le principe, c'est d'utiliser la spécification du système pour générer des séquences de test, ensuite ces séquences vont être appliquées sur l'IUT à l'aide d'un testeur, et finalement, en fonction des réponses de l'IUT, on décide si elle est conforme à sa spécification. La figure 3.1 résume ce principe. Cela entraîne deux problèmes distincts :

- créer des séquences permettant d'estimer que toutes les parties du système ont été mises à l'épreuve (ou une partie ciblée à l'avance),
- être capable d'interpréter les réponses de l'IUT pour évaluer si elle est conforme ou non à la spécification.

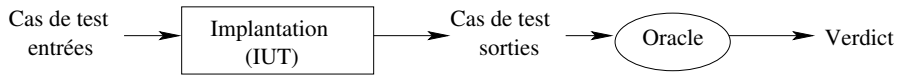


FIG. 3.1 – Schéma simplifié du test de conformité

La qualité d'une méthode de test de conformité est évaluée généralement par ce qu'on appelle la *couverture de test*. Mais avant d'introduire cette notion, il est indispensable de déterminer quelles sont les fautes de l'implantation qui doivent être couvertes. Une façon courante d'introduire un modèle de fautes est de définir un ensemble d'implantations avec un comportement erroné qui devrait être détecté par la suite de test. Si cet ensemble est fini, alors il existe une suite de test finie qui couvre toutes les fautes, sinon, il se peut

que celle-ci n'existe pas. Pour définir un modèle de fautes, on utilise l'approche dite de "mutation", dans laquelle un *mutant* d'une spécification A représente n'importe quelle implantation satisfaisant des hypothèses prédéfinies. Dans le cas des méthodes de test de conformité à base de FSM, les *mutants* de A correspondent à la classe de toutes les FSM (non déterministes) complètement spécifiées avec le même alphabet d'entrée. Un modèle de fautes $\mathcal{F}$ est un sous-ensemble de cette classe.

Pour distinguer parmi les mutants lesquels expriment un comportement erroné, on a besoin du concept de relation de conformité. On applique en général ce genre de méthode si on considère que l'IUT peut se représenter par le même formalisme que la spécification. Dans le cas des FSM, on utilise en général l'équivalence, la quasi-équivalence ou la relation de réduction ([Gil62]) comme relation de conformité. Ainsi, une FSM donnée A et une relation de conformité $conf$ partitionnent l'ensemble de mutants $\mathcal{F}$ en deux sous-ensembles :

- les implantations conformes :

$$C(A) = \{I | I \in \mathcal{F} \text{ et } IconfA\}$$

- les implantations non conformes :

$$N(A) = \{I | I \in \mathcal{F} \text{ et } InotconfA\}$$

Il y a plusieurs façons de caractériser un mutant ou un ensemble de mutants par rapport à un modèle de fautes donné. Dans l'approche par mutation traditionnelle, on définit un ensemble de *types de mutation*, chacun représentant un type spécifique de faute, ie une modification qui pourrait être introduite dans une spécification pour obtenir un comportement différent. Dans le cas des FSM, les types de mutations considérées sont en général de trois sortes :

- *les erreurs de transfert :* une implantation produit une erreur de transfert s'il existe une transition qui ne conduit pas au même état dans la spécification et l'implantation.
- *les erreurs de sortie :* une implantation produit une erreur de sortie s'il existe une transition qui ne porte pas le même symbole dans la spécification et l'implantation.
- *extra état (état manquant) :* une implantation est dite avoir un état supplémentaire si son nombre d'états doit être diminué (augmenté) pour obtenir le même nombre d'états que la spécification.

Mais ces types peuvent varier dans le cadre du test de logiciel.

Ainsi, les machines mutantes d'une classe vont contenir un certain nombre d'erreurs de sortie, de transfert, et éventuellement d'états additionnels (selon les méthodes choisies). Ensuite, pour chaque classe, un nombre limité de machines mutantes est généré aléatoirement, puis la suite de test est appliquée dessus. Finalement, la couverture de test est évaluée grâce à la proportion d'erreurs détectées durant ce processus. Il faut noter que plus le nombre de machines générées est grand, plus le résultat de couverture est fiable.

Dans ce chapitre, nous allons nous intéresser aux méthodes de génération de séquences de test connues, principalement dans le cadre du test de conformité. Ensuite nous présenterons un état de l'art sur le test de robustesse.

3.1 Génération de séquences pour le test de conformité de systèmes non temporisés

3.1.1 Point de vue général

Dans cette section, nous allons voir les différentes méthodes de test non temporisé. Elles se basent sur des spécifications sous forme de FSM et LTS. La plupart de ces méthodes de test proviennent du domaine des télécommunications, et en particulier de la validation de protocoles. Dans toutes ces méthodes de test, l'implantation est considérée comme une boîte noire, ie que les actions internes ne sont pas visibles. Les seules observations possibles sont les interactions de sortie produites par l'implantation lors de la consommation des entrées de messages émis par son environnement. Ce sont donc les interactions entre l'implantation et l'environnement qui vont nous permettre de juger de la conformité de l'IUT. Notons que ces méthodes ont pour but de tester toute l'implantation. Il existe par ailleurs des méthodes dites basées sur un objectif de test, dont le but est de tester simplement certaines parties ou propriétés du système.

3.1.2 Tour de Transition (TT)

Cette méthode ([NT81]) consiste à calculer, pour une FSM fortement connexe, une séquence d'entrées/sorties appelée Tour de transitions. L'application de cette séquence permet de visiter toutes les transitions de la machine au moins une fois. Un tour minimal peut être obtenu en résolvant le problème classique de graphes : le Tour du postier chinois rural.

Sur la figure 3.2, voici une exemple de tour de transition : $b/0.a/0.b/1.b/1.a/1.a/0$

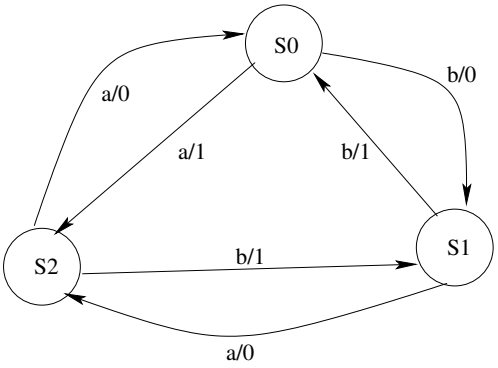


FIG. 3.2 – Une FSM quelconque

Le problème de cette séquence, c'est qu'aucun traitement n'est effectué pour vérifier que l'état d'arrivée de chaque transition est bien celui prévu dans la spécification. En d'autres termes, TT ne permet pas de détecter les erreurs de transfert. Elle est toutefois très utile pour détecter les erreurs flagrantes de l'IUT, car sa complexité est faible.

3.1.3 Notion de signature

Les méthodes de test suivantes fonctionnent sur le schéma décrit ci-après. Le principe est de tester les transitions de la FSM une à une, de manière à avoir une séquence de test pour chaque transition. Chaque transition $t = [S_i, i/o, S_p]$ est testée en trois étapes :

1. Positionner l'implantation à l'état de départ S_i de la transition : c'est le préambule.
2. Tester la transition en appliquant l'entrée i à l'implantation , puis vérifier si la sortie produite est bien celle prévue dans la spécification, ie o.
3. Vérifier que l'état d'arrivée est bien celui prévu dans la spécification, ie S_p : c'est le postambule.

Le préambule est une séquence d'entrées qui déclenche une séquence de transitions, et positionne l'implantation sur l'état de départ S_i depuis l'état initial. Le postambule est une séquence d'entrées à partir de S_p qui a pour but d'identifier S_p des autres états : c'est une *signature*.

Ainsi, pour une FSM comportant n transitions, on calcule n séquences de test, chacune divisée en trois parties. Pour le calcul du préambule, on suppose que la machine est fortement connexe, et réinitialisable. Pour le calcul du postambule, le but est de trouver une séquence d'entrées/sorties qui distingue chaque état de la machine. Pour cela, on suppose que les machines sont minimales. Nous allons maintenant aborder différentes méthodes pour identifier les états.

3.1.4 Méthode de la Séquence de Distinction (DS)

Cette méthode ([Gon70]) consiste à calculer une seule et unique séquence d'entrées appelée *séquence de distinction (DS)*. L'application de cette séquence à partir de chacun des états devrait engendrer une séquence de sorties distincte. Cela permet d'identifier de manière unique un état. Ainsi, on identifie l'état de départ et d'arrivée d'une transition testée pour s'assurer qu'il n'y a pas d'erreur de transfert sur cette transition. Sur la figure 3.2, la séquence d'entrées $a.b$ est une séquence de distinction. Le tableau 3.1 donne les signatures engendrées.

Etat	Signature
S0	a/1.b/1
S1	a/0.b/1
S2	a/0.b/0

TAB. 3.1 – Signatures engendrées par la figure 3.2

Dans le cas général, il n'est pas toujours possible de calculer, pour une machine donnée, une seule et unique séquence d'entrées qui, appliquée à partir des différents états, engendre des signatures distinctes. La figure 3.3 donne par exemple une machine qui ne possède pas de séquence de distinction. Notons que l'existence de cette séquence est favorisée lorsque la machine est complètement spécifiée, et si le nombre d'états est important. Cette méthode permet de détecter les erreurs de sortie et de transfert.

3.1.5 Méthode des séquences uniques d'entrée/sortie (UIO)

La méthode UIO (*Unique Input Output*) ([SD88]) est une généralisation de la méthode DS. L'idée est de déterminer pour chaque état une séquence d'entrées spécifiques pour cet état, telle que la séquence d'entrées/sorties engendrée soit unique pour cet état dans la spécification. On l'appelle *séquence UIO*. Ainsi, ce n'est plus la séquence de sorties qui caractérise les états, mais la séquence d'entrées/sorties. La figure 3.3 ne possède pas de séquence de distinction, mais possède les signatures sous forme de séquences UIO du tableau 3.2.

Etat	Séquence UIO
S0	c/1.b/1
S1	a/0.c/1
S2	b/0.c/1
S3	c/1.a/0

TAB. 3.2 – Signatures engendrées par la figure 3.3

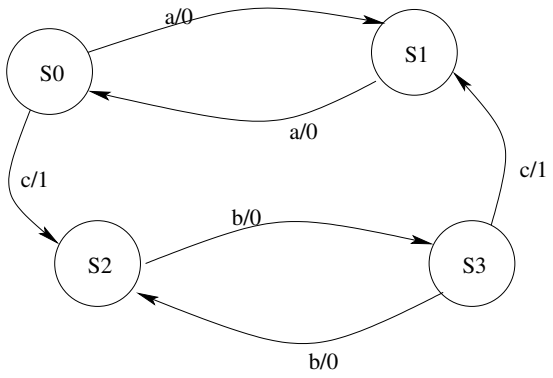


FIG. 3.3 – Une FSM sans séquence de distinction

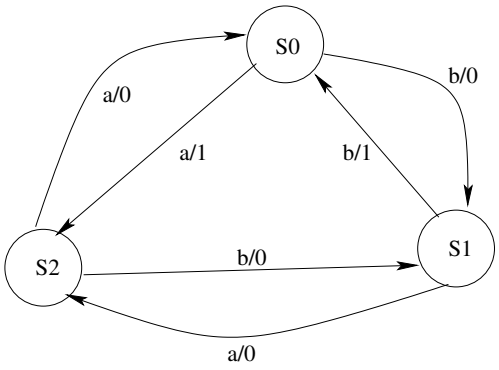
Cette méthode détecte les erreurs de sortie et de transfert. Toutefois, il se peut qu'il n'existe pas de séquence UIO pour chaque état.

3.1.6 La méthode des séquences de caractérisation (W)

Pour les machines qui ne possèdent pas de séquence de distinction ni de séquences UIO, la méthode W ([Cho78]) résout ce problème en calculant plusieurs séquences de distinctions partielles, appelées séquences de caractérisation. On considère encore que les machines sont minimales. Le principe, c'est qu'un état donné a forcément un ensemble de traces différent des ensembles de traces des autres états. Il est donc toujours possible de trouver pour chaque état une séquence d'entrées/sorties qui le distingue au moins d'un autre état. La méthode W consiste à calculer pour une machine un ensemble de séquences

d'entrées qui permet de distinguer n'importe quelle paire d'états d'une machine donnée. Cet ensemble est appelé *ensemble de caractérisation* et est noté $\mathcal{W}$. L'algorithme de calcul de cet ensemble se trouve dans [Gil62].

La figure 3.4 n'a ni séquence DS ni UIO.



états	séquences W a	b
S0	0	0
S1	0	1
S2	1	0

FIG. 3.4 – Machine sans séquence DS ou UIO

Une des solutions possibles pour la machine de la figure 3.4 est l'ensemble des séquences de caractérisation $\mathcal{W}$= $\{\{a\},\{b\}\}$. Comme le montre le tableau, la première séquence $W_1 = \{a\}$ permet de distinguer l'état S2 des états S0 et S1. On obtient ainsi deux partitions distinguées S2 et S0, S1. Les états S0 et S1 sont ensuite distingués par l'application de la deuxième séquence de caractérisation $W_2 = \{b\}$ depuis ces états. On remarque que cet ensemble de caractérisation est minimal, et l'application de la méthode Wp, vue ci-dessous, nous donnerait le même ensemble.

Comme les méthodes UIO et DS, la méthode W détecte les erreurs de transfert et de sortie. L'une des particularités de la méthode W est que pour tester l'absence d'erreur de transfert sur une transition donnée, la transition doit être expérimentée autant de fois qu'il y a de séquences de caractérisation. Ainsi, ce type de méthode engendre des procédures de test très longues. Finalement, nous sommes contraints de choisir entre l'utilisation des séquences W qui existent toujours, mais engendrent procédures de test très longues, et l'utilisation des séquences UIO, qui engendrent des tests plus courts, mais qui n'existent pas toujours. La complexité pour calculer les séquences de la méthode W est de l'ordre de $n^2 * k$, n étant le nombre d'états, et k le nombre de symboles d'entrée.

3.1.7 Les diverses améliorations (UIOv, Wp, UIOp, ...)

Il existe de nombreuses autres méthodes de test, qui souvent s'appuient sur ces modèles là. Nous citons dans un premier temps la méthode UIO avec vérification (UIOv) ([VCI89]). Il s'agit en quelques sortes d'une version corrigée de la méthode UIO. En effet, dans cette dernière, on a omis le fait que les signatures vont être appliquées à l'implantation, et non pas à la spécification. L'implantation étant une boîte noire, rien ne garantit que les

signatures uniques dans la spécification le restent au niveau de l'implantation. Cela remet en cause la détection des erreurs de transfert.

Afin de remédier à ce problème, la méthode UIOv propose de rajouter à la méthodologie vue au dessus une phase de vérification de l'existence et de l'unicité des signatures au niveau de l'implantation avant de tester une à une les transitions.

On peut citer aussi la méthode Wp ([FBK+91]), qui est une sorte de compromis entre la méthode W et la méthode UIO. L'idée est d'essayer d'optimiser les signatures. Plus précisément, on calcule pour chaque état le plus petit ensemble des plus courtes séquences de caractérisation, suffisant à l'identification de l'état. Une autre méthode propose d'extrapoler la méthode UIO et de calculer uniquement pour les états qui n'ont pas de séquence UIO, plusieurs séquences UIO partielles (UIOp) permettant d'identifier ces états. C'est la méthode dite UIOp ([CA92]).

Le tableau 3.3 donne les séquences UIOp et Wp de la figure 3.4.

Etats	Wp	UIOp
S0	{a,b}	{a}∪{b}
S1	{b}	{b}
S2	{a}	{a}

TAB. 3.3 – Séquences UIOp et Wp de la figure 3.4

3.1.8 Test basé sur les LTS

Avec le modèle des LTS, les approches sont assez similaires que pour les FSM. Toutefois, des relations de conformité différentes sont utilisées. On peut citer la bisimulation, ou les preordres (failure equivalence and preorder, testing equivalence and preorder, refusal preorder, etc...). Tretmans ([Tre96a], [Tre97]) généralise ces relations en introduisant la notion d'*observeur* : un LTS p est équivalent à un LTS q si n'importe quel observeur peut faire exactement les mêmes observations avec p qu'avec q. Il (re)définit ainsi différentes relations sur des LTS (relations testing preorder, **conf**, refusal preorder), ou des IOLTS[1] (une extension des LTS qui distingue les actions d'entrée des actions de sortie) à l'aide pour ces derniers d'une nouvelle notion d'état *quiescent* modélisant l'absence de sortie, et grâce à un opérateur de communication parallèle. Les relations étudiées sont les suivantes : relations input-output testing, **ioconf**, input-output refusal, **ioco**, **ioconf**, **ioco**$_{\mathcal{F}}$ (qui est une généralisation des deux précédentes).

Tretmans définit un cas de test aussi comme un IOLTS, et donne un algorithme (récursif) pour le générer. Le principe est de tester à chaque fois toutes les entrées possibles, et de prévoir le résultat pour chaque réponse éventuelle du système (pass ou fail). Ensuite, selon le choix de la relation, une machine sera considérée comme conforme ou non. La figure FIG. 3.5 montre un exemple de spécification et de cas de test correspondant (remarquons que c'est un graphe orienté, mais ici la flèche est sous-entendue).

[1] Input Output Labeled Transition System

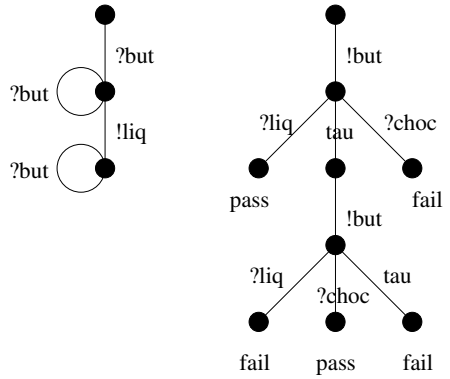


FIG. 3.5 – Spécification LTS (gauche) et un cas de test (droite)

3.2 Génération de séquences pour le test de conformité de systèmes temporisés

Depuis un certain nombre d'années, le temps fait souvent partie des paramètres à prendre en compte dans un système. On ne peut plus se contenter de méthodes qui vérifient uniquement les événements. L'instant d'émission et/ou de réception d'une action a aussi une grande importance, et il est donc normal que l'on cherche à savoir si un système respecte des contraintes de temps, et c'est pourquoi des méthodes de test prenant en compte le temps ont été mises au point.

3.2.1 Modèle de fautes pour systèmes temporisés

Le fait de prendre en compte les aspects temporels implique obligatoirement que l'on peut envisager de nouveaux types d'erreurs du système. Ainsi, un modèle de fautes des systèmes temporisés a été proposé dans [FP01], [Pet00] pour s'adapter aux systèmes spécifiés par des automates temporisés. Ce modèle est donc le suivant :

- *erreur de sortie, erreur de transfert, état supplémentaire* (voir systèmes non temporisés) ; ces erreurs sont toujours prises en compte.
- *erreur de dépassement de délai* : lorsque l'IUT n'envoie pas d'action de sortie alors qu'elle est attendue dans la spécification avec un certain délai.
- *erreur d'entrée temporelle* : lorsque l'IUT refuse un symbole d'entrée à un moment considéré comme valide par la spécification. Ce genre d'erreur peut parfois mettre en évidence un blocage du système.
- *erreur de sortie temporelle* : lorsque l'IUT envoie une action de sortie à un moment reconnu comme non valide par la spécification. Cette erreur est très proche du dépassement de délai, et possède souvent les mêmes causes. Parfois, on ne fait pas la différence entre les deux.

[EKD00] propose dans la même idée un modèle de fautes pour des TIOA communicants (une extension des TIOA ayant une file). Il définit deux types de fautes : les fautes de transfert et les fautes temporelles. Dans la dernière catégorie, il ajoute les notions de

fautes temporelles internes d'entrée ou de sortie, qui sont des fautes qui se produisent entre les différents CTIOA du système, et qui dépendent ainsi du contexte.

3.2.2 Méthodes avec objectif de test

Le principe de la technique par objectif de test est de ne tester qu'une partie prédéfinie du système. L'avantage c'est que le nombre de séquences de test est réduit, par contre, on ne garantit pas la couverture de la totalité du système. L'idée de [Lau99] et [Kon01] est de rechercher dans la spécification un chemin qui correspond à l'Objectif de Test, ce qui revient en fait à résoudre un problème d'accessibilité. Ils proposent un parcours en profondeur, avec une construction *à la volée* du graphe "synchronisé". Le but est de construire un chemin ayant des informations temporisées sur les transitions, pour pouvoir ensuite en déduire une séquence de test sous forme TTCN. Cette méthode est basée sur l'ETIOSM (*Extended Timed Input Output State Machine*) qui est en fait une extension des TIOA avec en plus un ensemble de variables (qui contrairement aux horloges n'évoluent pas avec le temps). Dans ce cas, nous obtenons trois verdicts différents :

- **Fail** : l'événement, le temps ou la variable ne correspond pas à la spécification
- **Pass** : l'événement, le temps et la variable vérifient la spécification et l'Objectif de Test
- **Inconclusive** : l'événement, le temps et la variable vérifient la spécification mais pas l'Objectif de Test

Pour parvenir à ces verdicts, il a fallu construire un *produit synchronisé* qui est en fait un automate (ETIOSM) produit à partir de deux automates distincts : la spécification et l'objectif de test. Pour cela, ils utilisent une méthode dite *énumérative*, en opposition à une méthode à base de *graphes de régions* ([Sal01]). Cette synchronisation se réalise premièrement sur les actions, puis sur les contraintes d'horloge. Des intervalles de temps synchronisés, satisfaisant une action de la spécification et une action de l'objectif de test sont alors construits. Ainsi, lors de la phase de test, si chaque action d'une séquence de test s'exécute dans son intervalle respectif alors l'implantation satisfait l'objectif de test.

[FPS00] et [Sal01] proposent de partir aussi de deux automates temporisés, la spécification et l'objectif de test, puis de transformer les deux en graphes des régions, et ensuite de les utiliser pour générer un produit synchronisé sous forme de graphe des régions.

Ces techniques permettent d'avoir en général moins de séquences de test, mais elles peuvent mener à des cas où on ne peut conclure quant à la conformité de l'IUT (Inconclusive).

3.2.3 Test basé sur la caractérisation d'états

Dans bon nombre de méthodes de test temporisé, le but est de réutiliser les techniques qui ont déjà fait leurs preuves dans d'autres types de systèmes. La tendance est de partir d'un modèle temporisé (automate temporisé par exemple), et d'appliquer dessus une transformation de telle sorte que le temps ne soit plus vu comme une notion propre, mais comme une donnée quelconque. Ce n'est que lors de l'exécution du test que le temps sera pris en compte et testé. Sur l'automate ainsi obtenu, il ne reste plus qu'à adapter une des

méthodes classiques de test. L'avantage, c'est que d'office, les erreurs de transfert et de sortie seront détectées. La difficulté réside dans le détection des autres types d'erreur.

[Pet00] propose de transformer l'automate temporisé de la spécification en graphe des régions, grâce à l'algorithme décrit dans [ACH+92], puis de le transformer en *automate plat*. L'algorithme de cette transformation se trouve dans [Pet00]. Cet automate est un modèle très proche du LTS. Enfin, une dernière modification consiste à réunir les symboles d'entrée et de sortie en une seule transition pour obtenir finalement une FSM, et pouvoir lui appliquer les méthodes de test basées sur les FSM et vues auparavant (UIO, W, Wp, ...).

Dans la même idée, [EDKE98] propose une approche en partant d'un TIOA pour spécifier les systèmes temporisés. La première étape consiste de façon classique à transformer le TIOA en graphe des régions. A partir de ce graphe des régions, et avec une certaine granularité, on obtient un automate à grille (*Grid Automaton (GA)*). Un GA est défini comme un sous-automate d'automate de régions. L'idée derrière la construction du GA est de représenter chaque région d'horloges avec un ensemble fini de valuations d'horloges, que les auteurs de [EDKE98] considèrent comme *représentatives* de la région. Pour cette raison, ils définissent les points de la grille avec une *granularité* $1/k$. Enfin ils le transforment en *Nondeterministic Timed Finite State Machine (NTFSM)*. Ainsi, ils peuvent adapter les méthodes connues pour générer les séquences de test, en l'occurrence ici la méthode Wp.

[KEDA00] utilise encore une fois le même schéma. La différence c'est qu'ils utilisent le modèle de temps discret. Certes, il ne correspond pas directement au comportement physique du système (contrairement au modèle de temps continu), mais il est suffisant dans de nombreux cas, comme pour des systèmes qui échantillonnent leurs entrées. Ils considèrent une horloge globale numérique, qui génère des événements *ticks* à une fréquence constante. Ils utilisent ainsi des timers, qui sont des variables entières automatiquement incrémentées après chaque tick, et qui peuvent être remises à zéro à l'occurrence de n'importe quel événement. La spécification est décrite en automate temporisé, puis transformée en un *tick-FSA* qui est une forme de LTS, avec une transition par *tick*. Ils proposent pour éviter l'explosion des état une tranformation directement du TA vers un *se-FSA*, qui regroupe les états du *tick-FSA* selon des ensembles. Ensuite, ils appliquent la méthode Wp, sur le *se-FSA*.

3.2.4 Test de système synchrone

[OP94] propose de tester une spécification sous forme de langage synchrone à flot de données : Lustre ([CPHP87]). Rappelons que ce langage est particulièrement adapté pour spécifier et développer des systèmes réactifs. Les auteurs présentent trois approches différentes.

La première consiste à transformer automatiquement un ensemble de propriétés d'invariant caractérisant l'environnement du système, exprimées en Lustre, en un générateur aléatoire de séquences de test. Ainsi, c'est la connaissance de l'environnement d'évolution du système qui permet de guider les tests. Il faut remarquer qu'il n'est pas toujours simple de connaître parfaitement les propriétés de l'environnement.

La deuxième approche requiert tout d'abord d'analyser et d'exprimer en Lustre les propriétés de sûreté du système. Ensuite, ces propriétés peuvent être utilisées (dans certains cas spécifiques) pour générer des séquences de test. Il est évident que cette approche a pour but de tester les propriétés de sécurité du système.

La dernière approche considère que Lustre est en fait directement utilisé pour l'implantation du système, et elle définit des critères de test basés sur la structure du programme. Le choix de ces critères dépend du modèle utilisé pour "capturer" la structure du programme Lustre.

3.2.5 Autres techniques...

Il est très difficile de faire un état des lieux exhaustif des méthodes de test temporisées. Dans cette partie, nous allons essayer de résumer quelques autres techniques.

On peut citer la méthode de [LC97a] qui se base sur une approche par *testeur canonique*, à partir des ETIOSM. Cette méthode utilise une relation de conformité fondée sur les traces temporisées d'automates : $R(I,S) \Longleftrightarrow Traces(I) = Traces(S)$. Pour gérer les aspects temporels, [LC97a] propose de découper la spécification en sous-automates chacun avec une seule horloge, puis de générer des intervalles de temps satisfaisant les contraintes de chaque transition obtenue. Cette méthode est plus rapide que la W, par exemple, mais en revanche son taux de couverture est plus faible.

Une approche utilisée dans [CL97] utilise comme langage l'algèbre ACSR. Le point départ est un langage graphique, le *graphe de contraintes* qui permet de décrire les contraintes de comportement et de performance. Ensuite, ce graphe est traduit sous forme de processus ACSR. Les tests sont encodés aussi en processus ACSR, à partir de l'étape précédente, en représentant une entrée (e,p) reçue par le système testé comme un événement de sortie $(\bar{e},p)$, et inversement. Ensuite, les auteurs décrivent comment ils utilisent ACSR à la fois comme modèle de description du système et comme représentation des séquences de test. Dans [DBALS97], les auteurs définissent et utilisent concrètement l'exécution du test comme un processus $R = (P \parallel T)\backslash E$, où P est le processus testé, T la séquence d'événements et d'actions élaborant le test, E est l'ensemble de labels d'événements sur lesquels P et T interagissent, et R est le résultat du test ($\bar{succ}ess$ ou $\bar{fail}ure$). Ainsi, il est possible d'utiliser les outils ACSR pour développer cette technique de test.

[NS01] propose de générer des séquences de test à partir d'automates à événements d'horloge. Ensuite, une relation de conformité est définie sur la théorie des tests de Hennessy ([NH84]). Cette théorie suppose que l'implantation I (et la spécification S) peut être observée par un ensemble fini de tests T via une séquence de communications synchrones de type CCS[2]. Ainsi, l'exécution d'un test est composée d'une séquence finie de communications formant une *Computation*, notée $Comp(T||I)$ (ou $Comp(T||S)$). Un verdict est attribué à une exécution de test, et une computation est satisfaite si elle se termine après une observation donnant le verdict Pass.

[2]Calculus of Communicating Systems - c'est une algèbre de processus

3.3 Quelques outils

Dans cette section, nous allons présenter brièvement quelques outils connus de vérification et test de conformité. Ces outils ont été choisis car ils permettent de prendre en compte les aspects temporels des systèmes. Ainsi, il est envisageable de les utiliser dans le domaine du temps-réel.

3.3.1 HyTech

Hytech ([HHWT95]) est un outil de vérification pour les systèmes hybrides. Il prend en entrée un système distribué temporisé décrit par des automates hybrides. A partir de ces derniers, il forme une composition parallèle ne donnant qu'un seul automate. Hytech utilise aussi un langage de description d'analyse qui permet d'écrire simplement des programmes itératifs pour produire des tâches telles que l'analyse d'accessibilité de composants ou la génération de traces d'erreur.

3.3.2 L'outil Kronos

Kronos ([Yov97]) est un outil de vérification des systèmes temps-réel. Il permet notamment (version 2.4.4) de tester qu'une spécification sous forme d'automate temporisé vérifie bien une propriété logique. Il permet deux approches différentes de la vérification : l'approche logique, et l'approche comportementale. Les critères à vérifier peuvent être spécifiés de deux façons : en formules au format TCTL (approche logique) et au format d'automate temporisé (approche comportementale).

Pour l'approche logique, Kronos implémente l'algorithme du model-checking qui s'assure qu'un système vérifie une propriété donnée au format TCTL. Il est possible de faire la distinction entre le model-checking avec *analyse en arrière (backward analysis)* qui effectue une recherche en arrière sur le graphe d'accessibilité (*reachability graph*), ie en calculant l'ensemble des prédécesseurs d'un état donné, et l'*analyse en avant (forward analysis)* qui exécute en revanche une recherche de l'ensemble des successeurs à l'aide d'un parcours en avant.

Pour l'approche comportementale, Kronos fournit un algorithme qui construit un automate dans lequel le temps est traduit *de façon abstraite (abstracted away)* de telle sorte que la relation de causalité entre les événements soit préservée. En effet, tester si deux automates temporisés sont *équivalents* revient à tester si les automates à temps abstrait (*time-abstracted automata*) ou FLTS[3] construits par Kronos sont *bisimilaires*. Cela est possible directement sur les fichiers générés (au format .aut) si on utilise l'outil ALDE-RABAN pour tester la bisimilarité.

L'environnement Kronos fournit divers outils, notamment "minim". Ainsi, en utilisant Kronos couplé avec "minim", il nous a été possible de convertir un fichier automate temporisé au format Kronos en un fichier de graphe des régions minimal toujours au format Kronos. Par ailleurs, "minim" permet de soulager le problème d'explosion des états.

[3]Flat Labeled Transition System

3.3.3 La représentation IF

IF (*Interchange Format*) ([BFG+00b]) est avant tout un langage pour modéliser les systèmes temps-réel communicant de manière asynchrone. Il est devenu ainsi le lien d'un ensemble d'outils de validation dédiés aux systèmes temps-réel, désormais appelé *IF validation environment*. La motivation pour créer IF a été d'obtenir un environnement complet, avec la possibilité de supporter différentes techniques de validation, allant de la simulation active à la vérification de propriété logique, en passant par les cas de test et la génération de code exécutable.

L'environnement de validation IF se découpe en trois niveaux de représentation de programme : le niveau de spécification, le niveau intermédiaire IF, et enfin le niveau de modèle sémantique LTS. La figure 3.6 décrit l'architecture totale et les connexions entre les différents outils de l'environnement.

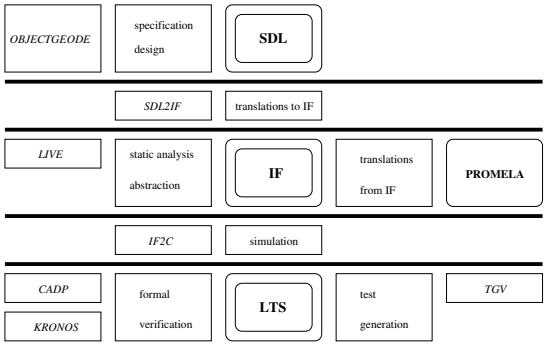


FIG. 3.6 – Un environnement de validation pour IF

Le niveau de spécification est la description initiale du programme, exprimée par exemple en utilisant un langage existant. Pour pouvoir être utilisée, cette description est traduite (automatiquement) dans sa représentation en langage IF. Le formalisme de la spécification en entrée utilisé ici est SDL, mais des connexions avec d'autres langages comme LOTOS ou PROMELA sont aussi possibles.

3.3.4 PARAGON

[DBALS97] utilise certains principes de haut niveau et décrit un outil de spécification, vérification et de test qui a été développé : PARAGON. Cet outil offre une interface graphique ou un langage de spécification textuelle avec une sémantique formelle. Les langages sont tous deux basés sur l'algèbre ACSR. La syntaxe graphique est basée sur le langage GCSR.

Dans PARAGON, l'exploration d'états, les tests d'équivalence et l'exécution interactive opèrent sur une représentation LTS du système analysé. Ce LTS est produit par un algorithme qui part du processus et cherche à créer un LTS avec toutes les exécutions possibles. Ceci génère ainsi des arcs inaccessibles. Par ailleurs, certaines fonctionnalités ont été rajoutées, comme l'exécution pas à pas.

En ce qui concerne le test, PARAGON travaille sur un LTS, et teste tout l'espace, ce qui peut parfois engendrer un temps non raisonnable. Mais il permet aussi une autre approche : le test de certains comportements spécifiques. Il utilise alors la composition parallèle pour appliquer des tests en ACSR au modèle du système.

En résumé, PARAGON est un outil de spécification, vérification, et de test pour les systèmes temps-réel, basé sur ACSR. Pour ce qui est du test, il s'agit d'un test classique en boîte noire, qui peut être exhaustif ou non. Le test exhaustif est souvent très long, et dans ce cas, il sélectionne des séquences de test avec la méthode décrite dans [CL97].

3.3.5 Lutess et Lurette : test de systèmes synchrones

Les outils LUTESS ([BORZ99]) et LURETTE ([RNHW98]), élaborés respectivement aux laboratoires LSR et Verimag, permettent de produire automatiquement des séquences de test conformes à une spécification de l'environnement d'un programme LUSTRE.

LURETTE permet la génération de séquences de deux façons :

- Des entrées "réalistes" générées à partir de la description non déterministe de l'environnement d'exécution.
- Le verdict du test est décidé à l'aide d'une description formelle des propriétés désirées, autrement dit les "comportements corrects".

LURETTE a pour objectif de concentrer les tests dans les cas où la vérification formelle est limitée (propriétés complexes, en particulier celles en rapport avec les aspects numériques). Il faut noter que la dernière version de LURETTE n'utilise plus le langage Lustre, mais le langage Lucky, créé par Verimag, qui permet de décrire et de simuler les systèmes réactifs stochastiques.

LUTESS ([BORZ99]) est un outil qui permet d'utiliser les principales techniques de test connues pour les systèmes synchrones. Il permet de générer automatiquement des séquences de test, ainsi que de donner directement le verdict du test. Par ailleurs, il permet de tracer l'exécution du test, et de mettre en évidence les situations où le programme viole certaines propriétés. Son but est de pouvoir simuler des comportements de l'environnement les plus réalistes possibles, de générer des séquences pour cibler une propriété donnée ou encore de conduire le système testé dans des situations intéressantes. Il utilise plusieurs méthodes pour générer les test :

- distribution statistique pour générer les entrées de l'environnement : les tests sont générés uniquement en respectant les contraintes de l'environnement.
- test basé sur des profils opérationnels : ces profils sont en fait des probabilités appliquées aux entrées.
- test orienté par les propriétés : le but est de vérifier qu'une propriété de sûreté n'est jamais violée.
- test basé sur un modèle comportemental : guider le système vers des états pertinents ou sensibles.

3.3.6 L'outil UPPAAL

Uppaal ([LPY97]) est un outil pour la validation (via une simulation graphique) et la vérification (via un model-checking automatique). Il consiste en deux parties distinctes : une interface graphique utilisateur, et un model-checker. L'idée, c'est de modéliser un système en utilisant un automate temporisé, de le simuler, et enfin de vérifier des propriétés dessus. Un système consiste en un ensemble de processus. L'étape de simulation permet d'exécuter le système de façon interactive pour vérifier qu'il fonctionne comme on le souhaite. Ensuite, on peut demander au vérifieur de regarder les propriétés d'accessibilité, ie si un certain état est accessible ou non. Il s'agit de model-checking.

Uppaal permet de saisir des automates temporisés, en autorisant les contraintes sur les transitions, les invariants sur les états et des synchronisations. Une variable suivie d'un " !" est une émission, alors que le " ?" signifie la réception. La saisie complète se fait de façon graphique. Le format des automates s'appuie sur le même principe que Kronos, ie qu'il est possible de définir des invariants sur les sommets et des affectations sur les transitions.

Uppaal permet aussi de définir l'ordre de recherche souhaité, comme Kronos, ou le type de réduction voulue lors des vérifications. Il est aussi possible d'exporter un graphe saisi au format Postscript. Par ailleurs, Uppaal peut aussi servir à générer certains tests.

3.3.7 Autres outils

Etant donné le nombre d'outils de génération de séquences existant, la liste exhaustive est impossible à réaliser. Nous allons ici donner quelques autres outils connus pour tester des systèmes complexes.

TVEDA

TVEDA ([Pha94]) est un outil de génération de séquences de test mis au point au CNET (centre de recherche de France Telecom). Il implante entre autres une méthodologie de génération de séquences par testeur canonique (vu précédemment). Il travaille à partir d'une spécification sous forme de machines à états finis étendues (EFSM pour Extended Finite State Machine), mais accepte aussi des spécifications Estelle et SDL. Les séquences obtenues sont décrites dans le langage TTCN (format décrit ci-après).

TGV

TGV[4] ([FJJV96]) a été développé à Verimag. Il utilise une approche par objectif de test. A partir d'une spécification et d'un objectif de test, sous forme de machines à entrées/sorties (IOSM pour Input Output State Machine), il génère des séquences de test au format TTCN. Dans un premier temps, il construit un graphe modélisant le comportement observable de la spécification dans un environnement de test. Ensuite, à

[4]Test Generation with Verification technology

l'aide d'un parcours en profondeur de la spécification, il produit une synchronisation entre celle-ci et l'objectif de test.

TestGen

TestGen ([ACM+96]) est un outil de génération de séquences de test développé à l'INT (Institut National des Télécommunications), à Evry. Il utilise SDL comme langage de spécification, dont il dérive une IOFSM comme représentation intermédiaire. A partir de cette IOFSM, trois types de séquences sont générés :

- des séquences dites "non optimisées"
- le tour du postier chinois rural
- des séquences de recouvrement

TestGen génère automatiquement les séquences en garantissant une couverture complète pour les fautes de transfert et de sortie, selon certaines hypothèses. La transformation de SDL vers IOFSM est effectuée grâce l'outil Geode, développé par Verilog (Grenoble), devenu ensuite ObjectGeode.

Samstag

Samstag[5] ([GHN93]) est une méthode et un outil (implantant cette méthode) de génération cas de test au format TTCN, en partant de spécifications de protocole écrites en SDL et en MSC (Message Sequence Charts [Z1299]). En général, le comportement du protocole à tester est écrit en SDL et l'objectif de test est écrit en MSC. Samstag permet de formaliser les objectifs de test, de définir les relations entre ces objectifs, les spécifications de protocole et les cas de test. Il contient par ailleurs différents algorithmes de génération de séquences, décrits dans [GHN93].

L'outil Samstag est constitué d'un outil de simulation MSC, d'un outil de simulation SDL, et d'un générateur de cas de test.

TestComposer

TestComposer ([Ver99]) a été ajouté à l'environnement de de développement ObjectGeode de Verilog. Il se base sur l'expérience acquise par la mise au point de TGV et TVEDA. C'est un outil de génération de séquences de test qui utilise des techniques d'exploration d'espace d'états, et qui utilise TTCN pour exprimer ce séquences. Une particularité de TestComposer est sa flexibilité pour exprimer des objectifs de test : il fournit une interface permettant à l'utilisateur d'adapter l'outil à n'importe quel langage de spécification connu dans le test. Le langage de base est SDL. Un autre point fort de TestComposer est la possibilité de générer automatiquement des postambules.

[5] Sdl And Msc baSed Test cAse Generation

3.3.8 Autres...

Comme nous l'avons dit, il est impossible de citer tous les outils de validation connus. On pourra toutefois citer l'environnement de validation Calife ([Cal00]) permettant la vérification et le test de systèmes complexes. Dans le cadre du test de robustesse, des approches ont été présentées dans les projets Vertecs et Triskell ([Jéz03]) de l'IRISA.

Les outils de validation de systèmes à base de composants sont présentés en introduction du chapitre 5.

3.4 Langage pour représenter les séquences : TTCN ... TTCN-3

TTCN (Tree and Tabular Combined Notation) est un langage normalisé ([ISO91]), qui représente les séquences de test en illustrant les symboles que le testeur doit émettre et recevoir, et qui est reconnu par la plupart des entreprises de test.

TTCN permet de décrire des séquences de test dites abstraites, c'est à dire exprimant principalement des interactions ou des primitives de services, sans en détailler le codage.

Deux formats pour TTCN ont été proposés : TTCN-GR (Graphic) qui modélise les séquences de test par tableaux, et TTCN-MP (Machine Processable) qui traduit ces formats en format accepté par les testeurs. Une expression TTCN-GR de séquence de test est composée de deux parties :

- une partie déclarative contenant la déclaration des PCO (Points de Contrôle et d'Observation) à utiliser par le test, la déclaration des horloges et les Unités de Service et de Protocole
- une partie dynamique qui décrit la séquence de test, c'est à dire les envois et les réceptions de symbole, l'interrogation et la réinitialisation des horloges et les verdicts liés à chaque observation.

TTCN a été prévu pour représenter des séquences de test depuis des systèmes non temporisés. Mais il contient plusieurs mots-clés utiles pour exprimer certains aspects temporels, notamment la gestion des horloges.

TTCN-3 (Testing and Test Control Notation) ([JGH00]) est le dernier successeur standardisé de TTCN. Le but de TTCN-3 est d'élargir le domaine d'utilisation aux autres types de test, comme le test de performance, d'interopérabilité et de robustesse, etc...

TTCN-3 possède une forme textuelle avec une syntaxe proche d'un langage de programmation, avec entre autres :

- configurations de test dynamiques et concurrentes
- des mécanismes de communication synchrones et asynchrones
- des templates pour les données
- paramétrisation pour manipuler n'importe quel type ou valeur
- écriture plus fine des verdicts
- utilisation combinée avec ASN.1

TTCN-3 possède plusieurs formats de représentation, et en particulier une forme graphique, très utile pour visualiser les systèmes complexes.

Il existe une extension de TTCN-3 pour les systèmes temps-réel, appelée TIMEDTTCN-3 ([ZRDN02]) permettant de gérer les aspects liés au temps.

3.5 Méthodes d'injection de fautes

Dans cette section, nous allons cibler les travaux qui se rapprochent plus du test de robustesse. Il faut reconnaître que la littérature dans ce domaine précis n'est pas riche. La plupart des études connues portent sur l'injection de fautes dans le système testé. Le principe général de ce genre de techniques est de faire en sorte que le système subisse des pertes d'information en mémoire, ou des dommages, et ainsi de voir s'il est capable de palier tout seul à ces fautes. Le but de ces insertions est en général de tester la *tolérance aux fautes* du système. La plupart du temps, ce genre de systèmes possède des architectures adaptées pour ce genre de situations, notamment des architectures redondantes. Dans le cas du test d'un logiciel, une autre approche consiste à modifier les entrées fournies au système, afin de pouvoir contrôler sa gestion des erreurs.

M. Kaaniche dans [Kaa99] étudie différentes approches pour évaluer la sûreté de fonctionnement d'un système : les fautes physiques, les fautes de conception ou les malveillances. Dans le même esprit, [Jar03] définit un cadre conceptuel d'étalonnage de la sûreté de fonctionnement, basé sur les techniques d'évaluation analytique et expérimentale. Le but est de permettre la quantification des mesures de sûreté de fonctionnement.

Même s'il existe de nombreux points communs, nous allons volontairement séparer par la suite deux types de test : le *test de tolérance aux fautes* et le *test de robustesse*. Tester la tolérance aux fautes, c'est vérifier que le système est capable de continuer à fonctionner correctement (ie conformément à sa spécification) malgré des dommages, des pertes ou des erreurs d'une partie du système. Comme nous venons de le voir, l'objectif de ce test est de contrôler des mécanismes de redondance, et d'élection. Tester la robustesse, c'est vérifier que le système adopte un comportement "acceptable" en cas d'aléas, qui peuvent être externes (actions inattendues venant de l'extérieur) ou internes (problème à l'intérieur du système).

3.5.1 Méthodes d'injection physique de fautes

Une première façon d'injecter des fautes dans un système est d'appliquer des perturbations électriques. [GKT89] et [Ka94] mesurent les effets des radiations sur celui-ci en les appliquant sur des répliques des CPU du système. [KFA+95] propose dans l'outil MARS une comparaison de trois méthodes d'injection physique de fautes :

- radiations avec des ions lourds
- perturbations au niveau des connexions des composants
- perturbations électromagnétiques

Quelques temps après, Samson ([SMF98]) injecte des perturbations sur les transistors de processeurs VLIW à l'aide d'un laser utilisé habituellement pour la conception de ce

genre de circuits. Le principal inconvénient de ces méthodes dites d'injection physique de faute est qu'elles coûtent relativement cher, et qu'elles peuvent provoquer des dommages irréversibles sur le système (selon la méthode choisie). De façon indirecte, il est difficile de tester de la sorte le système final. En général, les tests sont appliqués sur des répliques.

3.5.2 Méthodes d'injection logicielle de fautes

En parallèle de ces techniques, une autre approche d'injection de fautes a été développée, utilisant les aspects software plutôt que hardware. On appelle cela la méthode SWIFI (Software Implemented Fault Injection). Elle coûte beaucoup moins cher que les techniques d'injection physique, et elle permet de spécifier précisément quelle(s) faute(s) on souhaite injecter, et éventuellement de réitérer l'injection de cette faute. Parmi ces techniques, le système FIAT ([BCSS90]) modifie l'image binaire d'un processus en mémoire (injection de faute au moment de la compilation). FTAPE ([TIJ96]) combine un injecteur de fautes avec un générateur de charge de travail (workload generator) : l'injecteur de fautes mesure la charge de travail instantanée pour déterminer automatiquement l'instant et l'endroit d'injection qui provoquera la meilleure propagation de fautes dans le système. L'outil FERRARI ([KKA92]) permet de modifier directement l'image exécutable d'un programme sans le recompiler, de telle sorte que lorsque ce code modifié est exécuté, le système se comporte comme si des erreurs internes étaient présentes.

3.5.3 Approche basée sur les types de données

A l'Université Carnegie Mellon (Pittsburgh), une approche orientée objet est proposée pour tester la robustesse d'un logiciel, fondée sur les types de données des paramètres plutôt que sur les fonctionnalités du composant ([KJS98]). Elle implémentée dans l'outil *Ballista* ([DKPD99]).

C'est une approche en boîte noire, ce qui signifie dans [KJS98] que le testeur n'a besoin ni du code source du logiciel testé, ni de sa spécification, mais uniquement de l'interface, en termes de paramètres et de types de données, du composant testé. L'idée part de la constatation que la plupart des défaillances dites de "robustesse" d'un logiciel sont tout simplement provoquées par un code qui "oublie" de tester les entrées invalides d'un algorithme (ie ce sont des problèmes de préconditions).

Le but de Ballista est de générer automatiquement et d'exécuter des tests de robustesse, à l'aide d'entrées invalides. Le principe, c'est que pour chaque type de paramètre utilisé dans le composant, une liste de valeurs potentiellement invalides est construite et appliquée dans les cas de test. Ainsi, ces tests sont conçus pour détecter les "crashs" et les erreurs provoquées par des entrées invalides lors d'appels de fonctions. Les réponses du composant testé sont évaluées à l'aide de l'échelle CRASH ([JSD+97], [Car96]). Ensuite, ils proposent un modèle à plusieurs dimensions de fautes, basé sur le nombre de paramètres d'une fonction et sur les combinaisons possibles pouvant provoquer des erreurs. Ensuite, ils ont développé la possibilité de générer du code qui rattrape automatiquement les erreurs détectées ([PPD99]) et cette fonctionnalité a été ajoutée à Ballista.

Ensuite, Ballista a été expérimenté sur l'API Microsoft Win32 ([CPK00]) et sur différentes implantations de CORBA. D'autre part, [Gri99] suggère de tester la robustesse des protocoles avec l'outil PIRANHA, une extension de l'approche de Ballista, en ajoutant des actions exceptionnelles appelées *aléas*.

3.5.4 Simulation

Une autre possibilité pour mesurer la tolérance aux fautes est d'utiliser la simulation. Pour cela, on part d'un modèle mathématique du système, que l'on soumet ensuite à des scénarios de fautes. On peut citer quelques outils comme DEPEND ([GI90]), MEFISTO ([JAR+94]), FOCUS ([CI92]), REACT ([D.K93]) et ADEPT ([GJP95]). [Bou97] propose de vérifier expérimentalement la tolérance aux fautes à l'aide de l'outil MEFISTO précédemment cité.

L'avantage de la simulation c'est qu'il est possible d'injecter n'importe quel modèle de fautes. En revanche, elle ne met pas toujours en évidence le comportement réel de l'implantation.

Quelques études essaient de mélanger les différentes approches précédentes, comme Young ([YAIG93]) ou Gutoff ([GS95]). [KGJ95] propose de combiner l'injection de fautes hardware et software dans leur outil FERRARI.

3.6 Conclusion

Ce chapitre nous a permis de voir que beaucoup de méthodes de test ont été mises au point, et que beaucoup d'outils logiciels de validation ont déjà été développés. Toutefois, lorsqu'on commence à s'intéresser au test de robustesse, et en particulier pour des systèmes temporisés, la littérature sur ce sujet devient rare, alors que les besoins industriels sont réels.

Nous allons par le suite présenter une méthode pour tester la robustesse d'un système temps-réel. Tester sa robustesse, c'est vérifier qu'il adopte un comportement "acceptable" dans un environnement "stressant". Nous allons donc par la suite étudier comment provoquer un environnement "stressant" pour le système, ie une situation critique, en insérant des aléas. Nous allons d'autre part définir et évaluer ce qu'on appelle un comportement "acceptable". Nous utiliserons les techniques connues pour le test de conformité comme point de départ pour mettre au point le test de robustesse. Ainsi, notre méthode se basera sur une génération de séquences, dans laquelle nous insérerons des aléas. Ensuite, ces séquences seront appliquées à l'implantation du système. Enfin, cette suite de test sera analysée afin de donner un verdict final.

D'autre part, une architecture et un algorithme d'exécution seront exposés pour tester les systèmes temps-réels à base de composants. Enfin, nous présenterons un outil de génération de séquences que nous avons développé.

Chapitre 4

Proposition d'une méthode pour tester la robustesse d'un système temps-réel

Il y a encore quelques années, il n'était pas question de tester la robustesse d'un système. Son émergence est en fait due à l'évolution des systèmes. Avec l'omniprésence de l'informatique dans tous les domaines, et notamment les systèmes embarqués, il est devenu courant qu'un système soit soumis à des conditions d'utilisation inattendues ou difficiles. Dans ces cas, le système se doit d'être robuste, c'est à dire qu'il doit réagir de façon "acceptable" face à cette situation.

Dans ce chapitre, nous allons proposer une technique complète pour tester si un système temps-réel peut être considéré comme robuste. Pour cela, nous allons soumettre l'implantation à des conditions d'utilisation inattendues, à l'aide d'aléas, et nous allons ensuite évaluer si son comportement peut encore être considéré comme "acceptable".

Après avoir rappelé les grands principes de la sûreté de fonctionnement d'un système, ce chapitre donne dans un premier temps des pistes de réflexion pour tester la robustesse, puis étudie les diverses formes d'aléas envisageables et finalement expose notre technique de test.

4.1 Rappels sur la sûreté de fonctionnement

Dans cette section, nous allons rappeler le vocabulaire et les types d'erreurs répertoriées dans le domaine de la tolérance aux pannes. [Rif01] et [Koe] donnent une étude complète sur ce sujet.

Défaillances, erreurs, fautes

Il existe un classement entre ces différents termes. En résumé, on dira : "une *faute* provoque une *erreur* qui entraîne une *défaillance*".

- *Défaillance (ou panne).* On dit qu'un module (ou un système) est défaillant lorsque son comportement n'est plus conforme à sa spécification.

- *Erreur* : c'est un état du système tel que la poursuite de l'exécution est susceptible de conduire à une défaillance (exemple : pointeur nul dans un programme ou câble réseau débranché).
- *Faute* : il s'agit d'un événement ayant entraîné une erreur. Il peut s'agir d'un erreur de programmation ou d'un événement physique.

Remarquons qu'une erreur peut ne pas entraîner de défaillance immédiatement. Elle est alors latente tant que la défaillance ne s'est pas produite.

Degré de gravité des défaillances

En général, on exprime trois types de pannes :
- *panne franche* : soit le système fonctionne normalement (les résultats sont corrects), soit il ne fait rien. Il s'agit du type de panne le plus simple ;
- *panne par omission* : des messages sont perdus en entrée ou en sortie ou les deux ;
- *panne byzantine* : le système peut faire n'importe quoi.

Permanence des défaillances

Une défaillance peut être :
- transitoire : elle se produit de manière isolée ;
- intermittente : elle se produit aléatoirement plusieurs fois ;
- permanente : elle persiste dès qu'elle apparaît.

Ensuite, une classification précise des systèmes est faite en fonction de leur pourcentage de disponibilité dans le temps.

Ainsi, dans notre insertion d'aléas dans les séquences de test, nous allons nous inspirer de ces différents types de défaillances pour simuler le comportement inattendu d'un module.

4.2 La problématique du test de robustesse

4.2.1 Notions de robustesse

Le préliminaire à tout travail, c'est de définir ce que l'on entend par "système robuste". Il faut reconnaître que dans la littérature, de nombreuses interprétations de la robustesse existent. Nous avons décidé de prendre comme point de départ à notre travail la définition donnée par IEEE ([IEE90]) :

Définition 4.2.1 (Robustesse) *Un système est considéré comme robuste s'il est capable d'opérer correctement en présence d'entrées invalides ou d'environnement stressant.*

Il évident que cette définition reste très vague et nous la repréciserons plus tard. La suite du travail dépend notamment du sens que l'on donne à "opérer correctement" ou à "environnement stressant".

On peut toutefois voir assez bien ce genre de définition se rapporter principalement à des systèmes embarqués. En effet, on imagine finalement devoir mettre à l'épreuve la robustesse d'un système pour lequel la maintenance s'avère difficile car celui-ci se trouve dans un milieu hostile, ou bien parce que son arrêt immédiat entraînerait des conséquences désastreuses. Ainsi, le raisonnement que l'on adopte dans le domaine de la robustesse, ressemble plutôt à ceci : le système se trouve dans une situation critique, inattendue, mais il faut bien qu'il s'adapte, quitte à sacrifier certaines fonctionnalités momentanément... L'exemple le plus illustratif est une sonde spatiale, ou un robot sur une autre planète (ou dans l'océan). Celui-ci communique avec la terre, mais avec un certain retard, et il doit ainsi avoir une certaine autonomie, notamment en cas de situation inattendue. Dans ce cas, on peut par exemple préférer qu'il se concentre sur sa capacité à éviter un obstacle, quitte à accepter que le relevé de températures soit non opérationnel pendant un certain temps.

Nous avons aussi décidé de nous appuyer sur un domaine bien plus étudié, le test de conformité. Ainsi, l'idée est d'essayer de voir les points communs et les différences entre le test de conformité et le test de robustesse, ce qui permet ensuite de voir les aspects et les techniques de test de conformité qui sont adaptables au test de robustesse. Si nous avons choisi cette démarche, c'est que nous avons remarqué de nombreuses similitudes tout au long de nos travaux entre ces deux domaines du test.

Par la suite, nous appellerons environnement du système tout ce qui vient de l'extérieur de ce système, que ce soit le milieu dans lequel il évolue, ou bien les autres modules avec lesquels il communique. D'autre part, nous définissons la notion de *session de test* comme suit :

Définition 4.2.2 (Session de test) *On appelle session de test l'application d'une seule séquence de test sur l'IUT.*

L'action spécifique 23 du CNRS ([AS202]) présente le problème du test de robustesse et le situe par rapport au test de conformité. Ce travail nous a servi de point de départ pour notre méthode. Par la suite, nous avons essayé d'apporter quelques réponses aux nombreuses questions posées par cette action. Elle souligne aussi l'intérêt industriel de la mise au point de techniques de test de robustesse. Lors de cette action, les différentes équipes proposent leur ligne directrice en ce qui concerne leurs travaux à venir sur le test de robustesse. L'IRISA, le LABRI, le LRI et Verimag proposent des approches basées sur la construction d'un modèle formel spécifiant le système à étudier, et mettent l'accent sur les aspects analyse de modèle et sélection de tests à partir des modèles. Le LAAS aborde le cas des systèmes complexes pour lesquels aucune spécification formelle complète n'est disponible, en proposant notamment de guider la sélection par des heuristiques et des méthodes issues de travaux sur l'injection de fautes. En ce qui nous concerne, notre approche peut se situer à un niveau intermédiaire entre ces deux axes : nous travaillons sur une façon de spécifier notre modèle dans le cadre de la robustesse, et nous travaillons par ailleurs sur l'injection de fautes dans le système. Par ailleurs, nous proposons de nous intéresser aux aspects temporels du système testé.

4.2.2 Les nouveaux enjeux comparés au test de conformité

Pour pouvoir appréhender le test de robustesse, il faut essayer de répondre à une série de questions, qui permettront ensuite de dégager des pistes :

- Quelles sont les différences (possibles) entre le test de conformité et le test de robustesse ?
- Comment construire le domaine d'entrées du test ?
- Comment interpréter le domaine de sortie ?
- Comment modéliser le système ?
- Quel type d'architecture de test peut-on envisager ?

Dans la suite, nous n'avons pas la prétention de répondre précisément à toutes ces questions, qui d'ailleurs n'ont pas toujours la même réponse selon l'interprétation que l'on se fait de la robustesse. En revanche, nous allons être amenés à faire des choix par rapport à ces questions que nous justifierons.

Tester la conformité, c'est regarder si l'IUT du système se comporte de façon conforme à sa spécification. Autrement dit, la moindre défaillance sera pointée du doigt. L'approche pour tester la robustesse diffère nettement en ce point, puisqu'on cherche juste à détecter si un comportement peut être considéré comme "correct" ou "acceptable". Il faudra donc trouver une façon de donner une certaine tolérance au système.

D'autre part, il est assez intuitif de s'imaginer que plus les conditions de l'environnement sont difficiles et inattendues, plus la tolérance que l'on accepte dans le comportement du système devra être élevée. Autrement dit, il existe un lien entre le comportement observé du système, la définition de comportement "acceptable" et enfin le niveau de situation "critique".

Différences entre conformité et robustesse

Une première piste qui se dégage pour tester la robustesse d'un système est d'utiliser l'idée de générer des séquences de test, comme pour le test de conformité. Rappelons que ces séquences sont habituellement obtenues à partir de la spécification du système. Ensuite, ces séquences peuvent être appliquées au système, à l'aide d'une architecture de test classique. Une différence majeure entre les deux types de test, c'est que dans notre cas, nous allons essayer de soumettre le système à une situation inattendue, en utilisant ce que nous appellerons des *aléas*. Ces aléas seront vus plus en détail par la suite. Il s'agit d'événements ponctuels et inattendus qui vont être appliqués au système. Une fois que le système est soumis à ces aléas, il faut mesurer sa capacité à adopter un comportement "acceptable".

Il existe plusieurs sortes d'aléas vues par la suite. Une partie d'entre eux peut se trouver à l'intérieur des actions d'entrées envoyées au système. Si on se place d'un point de vue comparatif au test de conformité, on dira que l'on modifie le domaine d'entrée du test. De même, si on modifie le domaine d'entrée du test, et si les conditions d'utilisation sont inattendues, alors il n'est pas choquant de modifier aussi le domaine de sortie. Il est aussi possible de remettre en cause l'oracle du test de conformité. En effet, le test de robustesse peut envisager des verdicts plus subtiles que "succès" ou "échec", qui risquent d'être

insuffisants. On peut par exemple envisager une mesure quantitative de la robustesse, en rapport avec les aléas insérés au système.

Choix du modèle

Le choix du modèle est une étape cruciale pour tester la robustesse. Il existe plusieurs choix :

1. On peut décider que le modèle du système ne tient pas compte des aléas. Dans ce cas, la difficulté est de trouver des aléas significatifs. En fait, il se peut qu'une grande partie des fautes injectées au système n'aient pas d'influence sur le système, ou que les conséquences ne soient pas observables. On peut alors envisager des méthodes heuristiques pour sélectionner les aléas à injecter, dans le but d'obtenir des scénarios plus critiques et plus pertinents. On peut aussi penser à identifier le type de faute (hardware, software ou humain), et éventuellement définir une notion de niveau de faute (catastrophique, ...). On peut aussi tester le système confronté à une charge anormale de travail.
2. On peut au contraire décider que le modèle du système prend en compte les aléas, ce qui va nous rapprocher des techniques de test de conformité. Le problème, c'est qu'il impossible de prévoir ce qui par définition est imprévu. Il faut donc un modèle qui se concentre sur "l'essentiel" et qui puisse assimiler les aléas. Il est possible par exemple de séparer dans le modèle les aspects de fonctionnement "nominal" des aspects de fonctionnement "dégradé", comme le montre la figure FIG 4.1. Il faut aussi noter que souvent le différence entre ces deux aspects n'est pas évidente, et que la frontière peut être très floue. De plus, il est envisageable qu'une implantation du système à un instant précis ne se trouve ni dans un mode, ni dans l'autre, mais entre les deux... De même, il se peut que le système alterne d'un mode à l'autre, car sur certains système, une réparation peut être envisagée (par exemple, en cas d'amélioration des conditions extérieures ou en cas de changement de stratégie du système...). Dans ce cas, on peut envisager de parler d'une spécification S, d'un modèle de fautes M, et d'une propriété de robustesse P, appliquer des séquences d'entrée au système, et évaluer les sorties.
3. On peut travailler sur une approche mixte des deux précédentes.

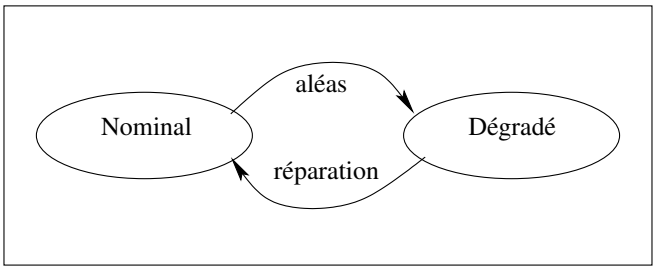


FIG. 4.1 – Modes nominal et dégradé

En ce qui nous concerne, nous avons choisi un modèle proche du deuxième : les aspects "nominaux" et "dégradés" sont séparés, mais les aléas ne sont pas explicitement pris en compte dans le modèle.

4.2.3 Schéma général de la méthode

Dans cette partie, nous allons donner les grandes lignes de notre méthode pour tester la robustesse. Il paraît en effet indispensable que le lecteur ait une idée générale de la technique avant de détailler chaque partie.

Le but essentiel du test de robustesse n'est pas encore une fois de trouver les erreurs du système, mais plutôt d'évaluer comment le système réagit face à des aléas, et par conséquent à des situations imprévues. Notre premier objectif va donc être de mettre le système dans ce genre de situation. Pour cela, nous avons travaillé sur les aléas et sur les possibilités de les appliquer sur l'IUT, et notamment en en insérant dans les séquences de test.

Ensuite, il faut un modèle pour représenter la spécification du système. Il est évident que plusieurs modèles étaient envisageables. Nous avons toutefois choisi le modèle des TIOA pour représenter la spécification du système, et ce pour les raisons suivantes :

- Ils prennent en compte les aspects temporels, ce qui est indispensable lorsque l'on parle de systèmes temps-réel.
- Ils permettent de différencier les entrées des sorties, ce qui s'avère très important lorsque l'on veut insérer des aléas. En effet, on pourra modifier les entrées appliquées au système, mais pas les sorties, qui servent à évaluer le comportement.
- Ils ont des fondements mathématiques solides.
- Ils permettent un niveau d'abstraction suffisamment précis nécessaire dans notre cas.

Nous partons donc d'une spécification $\mathcal{S} = (\Sigma, S, s^0, C, E)$, sous forme de TIOA, qui représente l'ensemble des comportements du système dans des conditions normales d'utilisation. C'est elle qui nous permettra de générer les séquences de test que l'on appliquera ensuite à l'IUT. Cette spécification est supposée vérifiée. Dans le cycle de développement, c'est cette spécification qui servira de base pour le développement de l'implantation. Par la suite, nous l'appellerons **spécification nominale**.

On demande aussi au concepteur du système d'évaluer quels sont les comportements qui sont considérés comme vitaux ou non dans le système. Une fois ce travail effectué, il est possible de formaliser ces aspects vitaux sous forme d'un autre TIOA $\mathcal{S}' = (\Sigma', S', s'^0, C', E')$. Par exemple, on peut imaginer que pour un robot, le système de guidage soit indispensable, mais qu'en revanche le relevé de température le soit moins. De façon plus simple, on peut considérer que lorsqu'on demande sa position à un robot, il répond dans un délai de une seconde en fonctionnement normal, alors qu'on peut accepter un délai de dix secondes en cas de situation critique, mais pas plus. Dans ce cas, $\mathcal{S}$ stipulera que le délai est de une seconde, alors que $\mathcal{S}'$ spécifiera un délai de dix secondes. Cette spécification est un TIOA qui peut s'exécuter, elle doit donc être vérifiée. Par la suite, ce TIOA prenant en compte uniquement les aspects vitaux du système sera appelé **spécification dégradée**. Les actions se trouvant dans la spécification dégradée seront appelées *actions vitales*.

Ainsi, nous avons comme point de départ deux spécifications, une nominale et une dégradée. Nous partons ensuite de la spécification nominale pour générer des séquences de test, en utilisant une méthode de génération à partir d'un automate temporisé. Pour cela, il est possible d'utiliser la méthode de [EDK02] ou la méthode de [SVD01].

Une fois ces séquences obtenues, on peut envisager de tester le système en situation critique. Une première étape consiste à soumettre le système à des radiations du type électromagnétiques ou ions lourds, comme dans [Ka94]. C'est un certain type d'aléa. Il est fort probable que certaines fonctionnalités du système ou certains modules soient atteints, mais nous cherchons à évaluer si le comportement vital est assuré. L'intérêt des ondes électromagnétiques, c'est que normalement elles ne doivent pas créer de dommages physiques du système, elle sont juste censées perturber momentanément les circuits.

Dans ces conditions d'exposition, on peut appliquer les séquences de test à l'IUT. Toutes les sorties du système sont enregistrées, les actions ainsi que leurs instants. Ces enregistrements serviront par la suite à évaluer le comportement, et affirmer si les critères de robustesse sont remplis. Quelles que soient les réponses de l'IUT, le testeur continue d'envoyer les entrées jusqu'à la fin de la séquence, sans se préoccuper des réponses du système. Nous avons fait ce choix car on considère qu'en milieu hostile ou éloigné, les systèmes ou modules qui communiquent avec l'IUT ne sont a priori pas au courant de la situation critique de celle-ci, et encore moins de son niveau de criticité.

Par ailleurs, si nous avons choisi de nous baser sur la spécification nominale plutôt que la dégradée pour générer les séquences, c'est parce que de façon concrète, il ne nous est pas possible de savoir exactement dans quel état se trouve l'IUT : elle doit se trouver dans les limites fixées par les deux spécifications. Et par défaut, un système qui communique avec l'IUT supposera qu'elle se trouve en mode nominal. Ce point sera vu en détails ultérieurement.

Une fois les séquences appliquées sur l'IUT, on définit une relation pour évaluer si les résultats enregistrés répondent aux critères exprimés dans la spécification dégradée. A la fin de cette étape, il nous est possible d'avoir une première évaluation sur la robustesse du système.

Dans un deuxième temps, on intègre dans les séquences de test précédentes des aléas, en jouant sur les entrées de ces séquences. On les appelle alors *séquences mutantes*. Ainsi, nous allons voir comment l'IUT réagit face à ces actions qui sont inattendues. Une fois ces aléas insérés, on procède de la même façon que précédemment : application des séquences à l'IUT, enregistrement des sorties, puis évaluation des résultats.

Cette méthode possède toute son efficacité lorsqu'on teste une IUT qui se trouvera ensuite dans un système à base de modules communicants. Une panne de l'un d'eux entraînera ainsi des actions inattendues transmises aux autres modules.

La méthode globale du test peut se résumer par le cheminement suivant :

1. `Génération de séquences de test à partir de` $\mathcal{S}$
2. `Début d'application de radiations sur l'IUT`
3. `Application des séquences sur l'IUT`
4. `Fin d'application de radiations sur l'IUT`

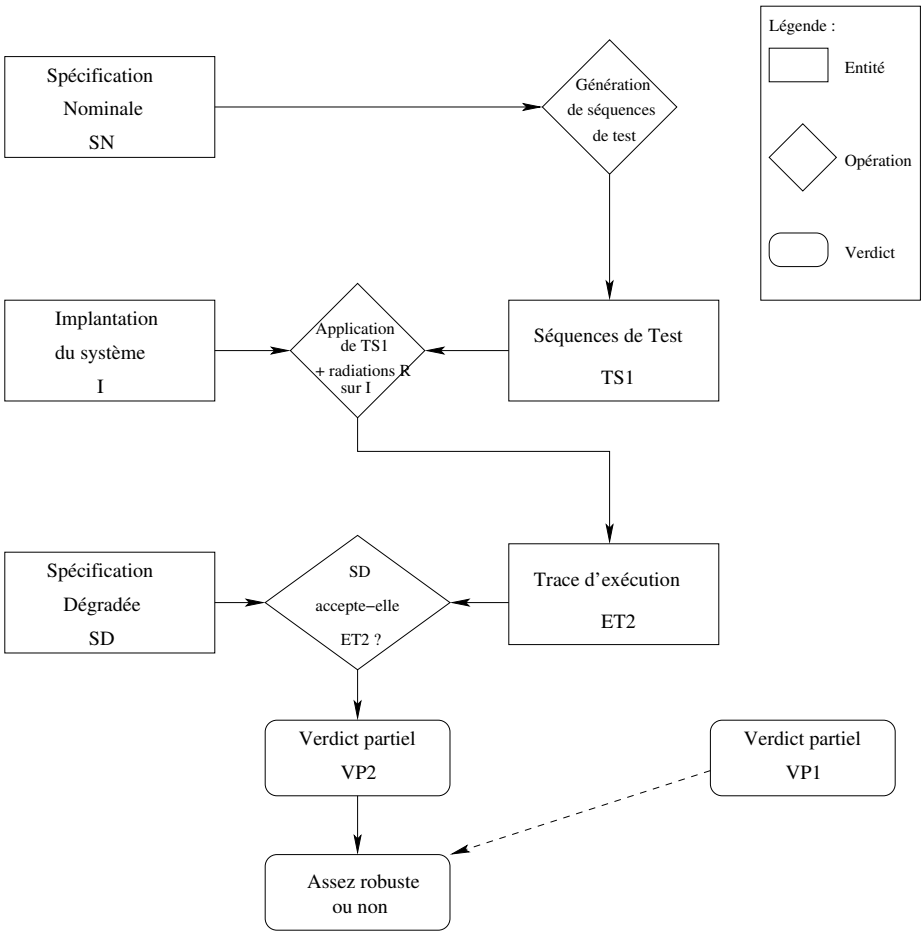


FIG. 4.2 – Schéma général de la méthode de robustesse : étape 1

5. Analyse des résultats et verdict partiel
6. Ajout d'aléas aux séquences de test
7. Application des séquences mutantes sur l'IUT
8. Analyse des résultats et verdict final

Un diagramme récapitulatif global se trouve dans les figures FIG 4.2 et 4.3, qui représentent les deux étapes majeures de la méthode.

4.2.4 Architecture de test

L'architecture de test que nous utilisons est assez simple. L'IUT est reliée par ses *Points de Contrôle et d'Observation (PCO)* à un exécuteur de test, qui contient les séquences non modifiées puis mutantes, et qui se charge de les appliquer au système. Dans le même temps, un moniteur de test est chargé d'enregistrer toutes les entrées et sorties du système.

Le schéma récapitulant l'architecture de test est donné figure FIG. 4.4.

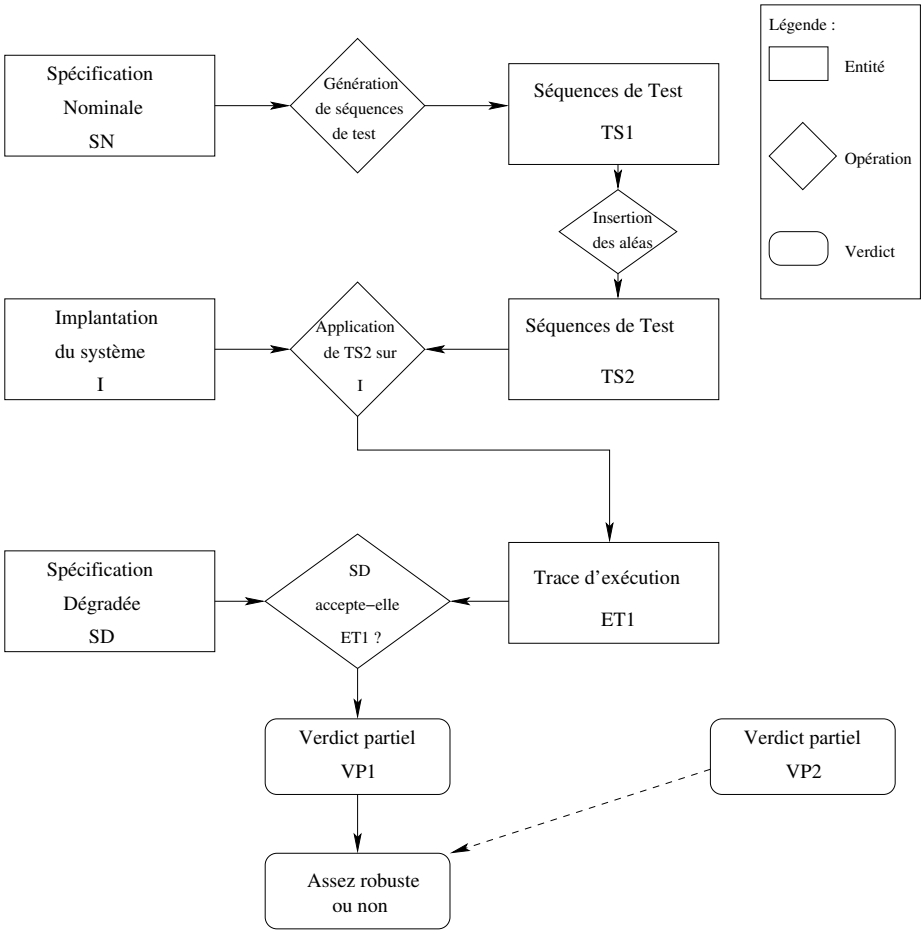


FIG. 4.3 – Schéma général de la méthode de robustesse : étape 2

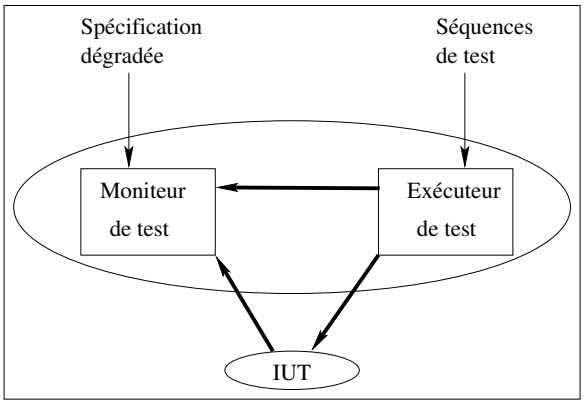


FIG. 4.4 – Architecture de test

4.2.5 Définitions et hypothèses

Nous allons aborder dans cette partie les définitions nécessaires pour pouvoir mettre en place notre méthode de test de robustesse. Elles nous serviront entre autres pour exprimer des hypothèse nécessaires pour pouvoir travailler sur les spécifications du système.

Définitions sur les TIOA

Nous allons commencer par rappeler quelques définitions classiques sur les TIOA qui sont utilisés pour représenter la spécification nominale et la spécification dégradée du système. La définition de TIOA ainsi que les définitions préliminaires se trouvent dans la section 2.3.

Nous utilisons la notion de déterminisme pour les TIOA, car nous considérons que si un état possède des transitions sortantes où certaines sont des entrées et d'autres des sorties, alors le TIOA n'est pas déterministe. En effet, si un système se trouve dans un tel état, il faut qu'on puisse savoir si le système attend une entrée, ou si au contraire on va attendre une sortie venant de lui. Si cette distinction est difficile, alors c'est un cas de non déterminisme. De même, si un état possède deux (ou plusieurs) transitions étiquetées avec la même action, alors il n'y a pas d'ambiguïté si les instants de tirage des deux transitions sont bien distincts. Ainsi, nous obtenons la définition suivante :

Définition 4.2.3 (Déterminisme) *Un TIOA* $A =< \Sigma_A, S_A, S_A^0, C_A, E_A >$ *est dit déterministe si et seulement si :*

- *A ne possède qu'un seul état initial*
- $\forall s, s', s'' \in S_A, \forall t \in E_A$ *:*
  - $t =< s, s', a, \lambda, \delta >$ *et* $a \in \mathcal{I}_\mathcal{A} \Longrightarrow \forall t' =< s, s'', b >\in E_A,\ b \in \mathcal{I}_\mathcal{A}$
  - $t =< s, s', a, \lambda, \delta >$ *et* $a \in \mathcal{O}_\mathcal{A} \Longrightarrow \forall t' =< s, s'', b >\in E_A,\ b \in \mathcal{O}_\mathcal{A}$
  - *Si plusieurs transitions sortantes de s ont pour étiquette la même action* $a \in \Sigma_A$*, alors l'intersection de leurs contraintes d'horloge est vide.*

Notons que cette définition se rapproche du déterminisme des FSM, car aucune confusion n'est possible entre une entrée et une sortie. La figure 4.5 donne quelques exemples de déterminisme (ou non) de TIOA selon cette définition.

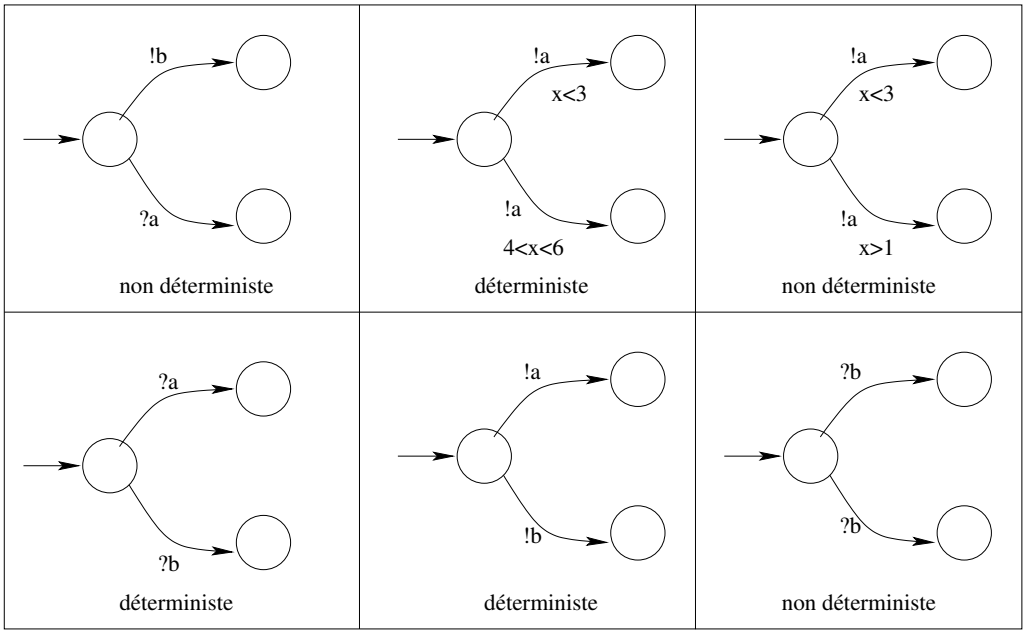


FIG. 4.5 – Exemples de déterminisme et de non-déterminisme

Définition 4.2.4 (Minimalité) *Un TIOA* $A =< \Sigma_A, S_A, S_A^0, C_A, E_A >$ *est dit minimal si pour toute paire* s_i, s_j *de* S_A, *il n'existe aucune séquence temporisée* x *qui, lorsqu'on l'applique à* s_i *et* s_j, *amène* A *dans le même état* s_r.

Définition 4.2.5 (Fortement connexe) *Un TIOA* $A =< \Sigma_A, S_A, S_A^0, C_A, E_A >$ *est dit fortement connexe si pour chaque paire d'états* s_i, s_j *de* S_A, *il existe une séquence temporisée d'actions* x *telle que* s_j *est accessible depuis* s_i *si* x *est appliquée.*

Définition 4.2.6 (Complètement spécifié (ou Complet)) *Un TIOA* $A =< \Sigma_A, S_A, S_A^0, C_A, E_A >$ *est dit complètement spécifié (ou complet) si tous les états de* A *acceptent n'importe quelle action* $a \in \Sigma_A$, *à n'importe quel instant.*

Définition 4.2.7 (Complet au niveau des entrées) *Un TIOA* $A =< \Sigma_A, S_A, S_A^0, C_A, E_A >$ *est dit complet au niveau des entrées si tous les états de* A *acceptent n'importe quelle entrée* $a \in \mathcal{I}_\mathcal{A}$, *à n'importe quel instant.*

Certains états d'un TIOA possèdent des propriétés intéressantes. Ce sont des états dans lesquels le système restera en attente, tant que l'environnement (ou le testeur) n'applique pas une entrée. Ce sont ces états qui nous permettent de mener le système dans les parties qui nous intéressent. On appelle ces états des *états contrôlables.*

Définition 4.2.8 (Etat contrôlable) *Dans un TIOA* $A =< \Sigma_A, S_A, S_A^0, C_A, E_A >$, *un état* $s \in S_A$ *est dit contrôlable si et seulement si :*
$\forall s' \in S_A - \{s\}$ *et* $\forall e \in E_A$, $e =< s, s', a, \lambda, \delta > \Longrightarrow a \in \mathcal{I}_\mathcal{A}$.

Cette notion d'état contrôlable est assez similaire à le notion d'état quiescent défini pour les LTS dans [Tre96a]. Par exemple, dans la figure FIG. 4.6 ci-après, les états $S0$, $S4$, et $S'0$ sont des états contrôlables.

Hypothèses

Nous supposons pour la suite que les spécifications nominale $\mathcal{S}$ et dégradée $\mathcal{S}'$ sont :
- déterministes
- vérifiées
- fortement connexes
- minimales.

En effet, il faut que la spécification soit vérifiée avant de pouvoir aborder les problèmes inhérents au test, pour qu'il n'y ait pas de problème d'états puit par exemple. La spécification dégradée doit aussi être vérifiée, car elle doit pouvoir être vue comme un comportement complet et indépendant du système. Les spécifications doivent être déterministes car lorsque l'on applique une séquence de test sur un de ces systèmes, il est dispensable que les sorties obtenues de l'IUT soient toujours les mêmes pour une séquences d'entrées donnée. Dans les techniques de test, la minimalité permet d'assurer que l'on peut trouver deux séquences différentes pour distinguer un état d'un autre. C'est donc une hypothèse nécessaire pour nos deux spécifications. Enfin, il faut que les automates soient fortement connexes pour que l'on puisse explorer, si on le souhaite, tous les états.

Par ailleurs, la spécification dégradée $\mathcal{S}'$ ne doit pas être choisie complètement par hasard. En effet, il paraît assez intuitif qu'il y ait un certain lien entre la spécification nominale $\mathcal{S}$ et $\mathcal{S}'$, qui sont à priori des représentations du même système. Comme nous l'avons vu précédemment, la spécification dégradée va représenter uniquement les actions vitales du système, mais doit pouvoir être vue comme un système autonome. C'est pour cette raison que ses horloges lui sont propres, et n'ont aucun lien avec celles de la spécification nominale.

Ainsi, le comportement de $\mathcal{S}'$ doit être en quelques sortes "inclus" dans celui de la spécification nominale $\mathcal{S}$. Pour exprimer cela, nous nous basons sur les actions et sur leur ordre : on va considérer que le comportement d'une spécification dégradée $\mathcal{S}'$ est "inclus" dans celui de la nominale $\mathcal{S}$ si on peut retrouver pour chaque chemin x' de $\mathcal{S}'$ un chemin x dans $\mathcal{S}$ dans lequel toutes les actions de x' se retrouvent dans x, et avec le même ordonnancement. Ce genre de raisonnement se retrouve dans les inclusions de trace. Pour cela, on définit une relation de précédence et une inclusion de comportement :

Définition 4.2.9 (Relation de précédence ($\prec_T$)) *a_i précède a_j dans un chemin T, noté $a_i \prec_T a_j$ si l'action a_i a lieu avant l'action a_j dans T.*

Définition 4.2.10 (Inclusion comportementale ($\subset_{\mathcal{D}}$)) *Soit $A = (\Sigma_A, S_A, s^0_A, C_A, E_A)$ and $A' = (\Sigma_{A'}, S_{A'}, s^0_{A'}, C_{A'}, E_{A'})$ deux TIOA. Le comportement de A' est inclus dans celui de A, noté $A' \subset_{\mathcal{D}} A$ si et seulement si :*
- $s^0_A = s^0_{A'}$
- $\Sigma_{A'} \subset \Sigma_A$

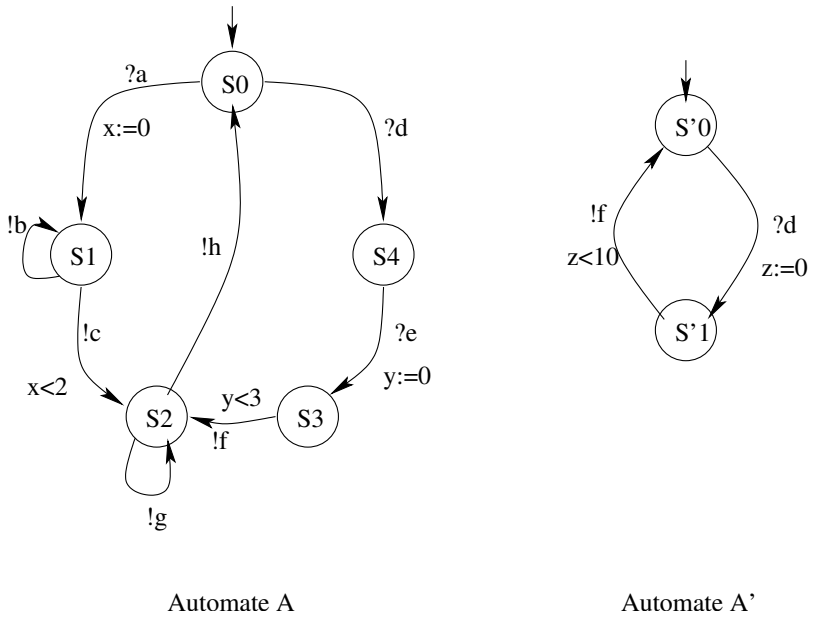


FIG. 4.6 – Exemple de TIOA respectant l'inclusion comportementale

– *pour chaque chemin T' de A', il existe un chemin T de A tel que* $\forall (a,b) \in (\Sigma'_A)^2$, $a \prec_{T'} b \Longrightarrow a \prec_T b$.

Il faut remarquer que cette relation tient compte du comportement, c'est à dire des actions, mais elle ne prend absolument pas en compte les aspects temporels. Ensuite le concepteur doit faire en sorte que les contraintes de temps de la spécification dégradée soient plus "légères" (ou au moins équivalentes) que celles de la spécification nominale.

Nous allons illustrer cette relation à l'aide d'un exemple. La figure 4.6 montre deux TIOA, A et A'. Si on regarde les actions, on peut voir que pour tout chemin dans A', on peut trouver un chemin dans A contenant toutes les actions de A' dans le même ordre.

Par exemple, le chemin $?d.!f$ de A' se retrouve dans le chemin $?d.?e.!f$ de A. De même, le chemin $?d.!f.?d.!f$ de A' se retrouve dans le chemin $?d.?e.!f.!h.?d.?e.!f$ de A, et ainsi de suite pour tous les chemins de A'. Ainsi, on peut affirmer ici que $A' \subset_{\mathcal{D}} A$.

Cette relation doit être respectée par les spécifications du système. Par conséquent, en plus du fait que les automates sont déterministes, vérifiés, fortement connexes et minimaux, ils doivent aussi respecter la relation $\mathcal{S}' \subset_{\mathcal{D}} \mathcal{S}$

Pour des raisons pratiques évidentes, on considère que toute transition des TIOA $\mathcal{S}$ et $\mathcal{S}'$ qui a pour étiquette une action de sortie doit être passée en un temps fini. Autrement dit, si aucune borne de temps n'est fixée, nous imposons alors une limite pour chaque transition avec sortie. La motivation de ce choix, c'est que par la suite, nous serons capable de décider que si aucune sortie n'a été renvoyée par le système avant la limite, alors l'IUT comporte une erreur.

L'implantation

Le système considéré est censé être robuste, du moins c'est ce qu'on cherche à vérifier. Pour cela, nous allons lui soumettre des entrées qui ne sont à priori pas attendues par la spécification. On suppose toutefois que le système ne devrait pas se bloquer si une entrée "parasite" lui est appliquée (ce qui serait curieux pour un système robuste). C'est pour cela que nous supposerons que l'implantation du système est un TIOA complet au niveau des entrées.

Les séquences de test

De façon générale, une séquence de test est un succession de symboles de l'alphabet. Elle contient à la fois les entrées que le testeur doit fournir au système, mais aussi les sorties que l'on attend du système, ce qui permet ainsi à celui-ci de vérifier si la réponse attendue est correcte. Puisque nous utilisons des modèles temporisés, nous allons donc devoir générer des séquences temporisées, c'est à dire que les instants auxquels les actions sont tirées ont de l'importance. Ces séquences sont obligatoirement extraites de la spécification nominale $\mathcal{S}$ du système.

Il est possible de représenter les séquences de test temporisées de plusieurs façons différentes. En général, on ne donne pas dans la séquence directement l'instant d'occurrence d'une action, mais plutôt l'intervalle de temps pendant lequel cette occurrence peut avoir lieu. S'il s'agit d'une action de sortie, le testeur contrôle que l'occurrence a bien lieu dans l'intervalle autorisé. Si c'est une entrée, le testeur peut choisir à quel instant émettre l'action dans l'intervalle requis. Les possibilités de représentation sont les suivantes :

- La séquence est une suite de paires formées d'une action et d'un intervalle de temps faisant référence à une horloge globale (initialisée en général au début). L'inconvénient majeur de cette représentation, c'est qu'il n'est pas possible de donner l'instant d'une action en fonction de l'occurrence d'une action précédente. Autrement dit, cette représentation interdit les réinitialisations d'horloge.
- La séquence est une suite de paires formées d'une action et d'un intervalle de temps faisant référence à l'occurrence précédente de la même action. Le problème cette fois, c'est que l'automate utilisé doit avoir une horloge par action : c'est un automate à événements d'horloge ([AFH94]).
- La séquence est une suite de triplets formés d'une action et d'une région d'horloges, qui représente l'intervalle de temps autorisé pour cette action. L'inconvénient de cette représentation, c'est qu'il faut générer le graphe des régions, ce qui est très coûteux.
- La séquence est une suite de triplets formés d'une action, de contraintes d'horloges, et des horloges réinitialisées. En fait, cette suite contient toutes les informations que l'on trouve dans le chemin correspondant dans la spécification. Cette représentation permet d'exprimer le plus large éventail de séquences, permettant notamment le lien entre l'occurrence d'une action et l'instant d'occurrence d'une action passée. L'inconvénient, c'est que le testeur doit avoir les mêmes horloges que le TIOA, et que la représentation est peut-être moins intuitive.

Dans notre étude, nous avons choisi la dernière représentation, qui nous permet d'écrire facilement n'importe quelle séquence tirée de n'importe quel TIOA. Prenons l'exemple de la figure 4.7.

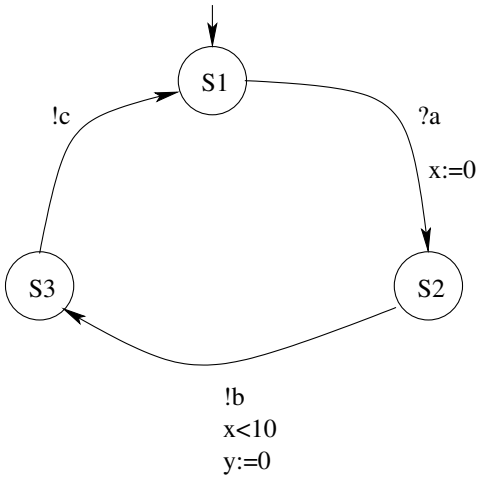


FIG. 4.7 – Exemple de TIOA pour extraire des séquences

Voici deux séquences différentes qu'il est possible d'extraire de cet automate :
- $(?a, x := 0, -), (!b, y := 0, x < 10)$.
- $(?a, x := 0, -), (!b, y := 0, x < 10), (!c, -, -)$.

Ces informations sont suffisantes pour que le testeur sache exactement dans quel intervalle de temps les actions doivent avoir lieu, à condition que celui-ci fasse évoluer les mêmes horloges que la spécification, et qu'il les réinitialise lorsque la séquence le prévoit.

Hypothèses sur les séquences de test

Nous proposons d'utiliser a priori n'importe quelle méthode connue de génération de séquences temporisées de test, valable à partir d'un TIOA. Il est possible d'utiliser une méthode inspirée d'une variante de méthode UIO ou W, qui permette un parcours assez large du système. Dans ce cas, la couverture de fautes devrait être plus élevée. En ce qui nous concerne, nous avons développé notre propre méthode de génération de séquences de test, inspirée de la méthode UIO et de la méthode Wp, que nous aborderons par la suite. Il est aussi possible d'utiliser la méthode de [EDK02] ou de [SVD01].

Toutefois, parmi toutes les séquences générées par une de ces méthodes, nous garderons que celles qui respectent les hypothèses suivantes : une séquence doit être extraite à partir d'un chemin de la spécification nominale qui :
- commence par une action d'entrée,
- finit sur un état contrôlable.

La séquence doit commencer par une entrée pour des raisons de contrôlabilité. Il paraît évident que ce soit le testeur qui décide à quel moment la séquence de test doit commencer, et non l'IUT. Par ailleurs, il faut finir sur un état contrôlable pour que le système termine

la séquence sur un état d'attente, et que l'on soit sûr qu'une action de sortie n'est pas sur le point d'arriver.

Résumé sur les hypothèses

Nous résumons brièvement les hypothèses que nous faisons sur le système et les séquences :

- les spécifications nominale $\mathcal{S}$ et dégradée $\mathcal{S}'$ sont déterministes, vérifiées, fortement connexes, minimales,
- l'implantation du système est un TIOA complet au niveau des entrées,
- une séquence de test doit être extraite à partir d'un chemin de la spécification nominale qui commence par une action d'entrée et finit sur un état contrôlable,
- toute transition de $\mathcal{S}$ et $\mathcal{S}'$ étiquetée par une action de sortie est bornée dans le temps,
- les horloges de $\mathcal{S}$ et de $\mathcal{S}'$ sont indépendantes,
- $\mathcal{S}' \subset_{\mathcal{D}} \mathcal{S}$.

4.3 Les aléas

Dans cette section, nous allons étudier en détails ce qu'on appelle les aléas, et comment on compte les appliquer au système. Lorsqu'un système est mis dans un milieu hostile, il va alors être soumis à toutes sortes d'événements imprévus, comme par exemple une élévation de température ou une pression anormalement élevée. Ce genre d'événement risque de perturber le comportement du système, et est par définition aléatoire et imprévisible, c'est pourquoi on le nomme *aléa*.

Nous allons définir le terme aléa comme suit :

Définition 4.3.1 *On désigne par aléa tout événement qui n'est pas prévu dans le fonctionnement normal du système.*

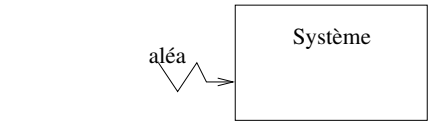
Ainsi, il peut s'agir aussi bien d'un problème matériel à l'intérieur du système, de perturbations électromagnétiques, ou encore d'actions inattendues venant de l'environnement du système. Le point commun de tous ces événements, c'est qu'en général ils ne figurent pas dans le cahier des charges ou la spécification du système. De toutes façons, il est difficile voire impossible de trouver un modèle de spécification qui prenne en compte des événements qui sont par définition imprévus. Toutefois, il est possible d'utiliser les connaissances des aléas, avec l'expérience acquise sur d'autres systèmes évoluant dans des conditions similaires, et de créer un modèle plus adapté. C'est une méthode plus "expérimentale" qui peut s'avérer efficace à la longue.

En ce qui nous concerne, la spécification ne prend pas en compte les aléas, mais plutôt le comportement vital qui doit être assuré en cas de situation difficile (décrit dans la spécification dégradée). Nous cherchons donc à reproduire ce genre de situation difficile pour évaluer si le système assure le comportement vital.

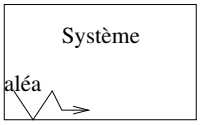
4.3.1 Différents types d'aléas

On peut classer les aléas en fonction du lieu où ils arrivent par rapport aux frontières du système. [CW02] propose la classification suivante, résumée dans la figure FIG. 4.8.

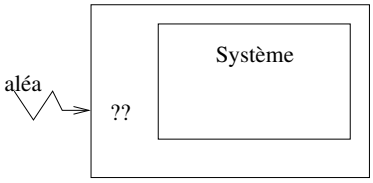
- Les *aléas internes* sont des événements inattendus qui ont lieu à l'intérieur du système. Par exemple, la rupture d'un transistor sur une carte de l'IUT sera considérée comme un aléa interne. En général, ce genre d'aléas concerne les pannes matérielles, ou l'apparition d'un élément anormal dans le système (eau, sable, ...).
- Les *aléas externes* sont des événements inattendus qui ont lieu au niveau de la frontière même du système. Par exemple, des actions non attendues par le système feront partie de cette catégorie d'aléas.
- Les *aléas hors frontière* sont des événements inattendus qui ont lieu au delà de la frontière du système. L'origine de ces aléas peut être lointaine, mais ils ont des conséquences immédiates sur le système. Par exemple, des radiations électromagnétiques ou solaires, ou encore une température extérieure anormalement élevée seront rangés dans cette catégorie.



(a) Aléa externe



(b) Aléa interne



(c) Aléa hors frontière

FIG. 4.8 – Différentes sortes d'aléas

On peut voir que ces différentes catégories sont assez souvent liées. Il est fréquent qu'un aléa hors frontières ait pour conséquence la rupture d'un composant électronique, ce qui peut être vu comme un aléa interne. Par exemple, une température trop élevée de l'environnement (aléa hors frontière) entraîne fréquemment la rupture d'un composant électronique ou une surchauffe du processeur (aléa interne).

Pour ces raisons, et pour faciliter la simulation des aléas, nous avons décidé de tout regrouper en deux catégories :

- Les *aléas externes* sont des actions inattendues appliquées au système. Il peut s'agir de l'action proprement dite, ou d'un mauvais timing. Ce genre de situation peut arriver lorsque le système chargé de communiquer avec l'IUT est en dysfonctionnement, ou en cas de problème de transmission de l'information.
- Les *aléas internes* regroupent tous les événements inattendus qui affectent l'intérieur du système. Il s'agit donc à la fois des pannes matérielles, ou de tout autre événement qui peut être du à une cause extérieure au système, comme une température anormale ou des radiations sur le système.

4.3.2 Insertion manuelle d'aléas

Le problème qui se présente dans cette partie est le suivant : nous avons une implantation à tester et nous avons un ensemble de séquences de test à notre disposition. Nous voulons maintenant soumettre cette IUT à des aléas pour évaluer son comportement.

Aléas internes

Dans un premier temps, nous essayons de nous intéresser aux aléas internes. Nous souhaitons créer une situation critique pour les composants physiques du système, mais sans causer des dommages irréversibles (ce qui entraînerait un surcoût de production). Un bon compromis consiste à exposer le système à des radiations électromagnétiques, comme dans [Ka94]. Ces radiations perturbent le fonctionnement du système, mais n'altèrent en principe pas ses composants électroniques. On peut aussi envisager de soumettre l'IUT à une augmentation anormale de température. Normalement si le système est bien conçu, et si l'augmentation de température reste "tolérable", il ne devrait pas y avoir de dommage irréversible. Selon les conditions d'utilisation, on pourra aussi soumettre l'IUT à une pression anormalement élevée, à l'aide par exemple d'un caisson de pression.

L'insertion d'aléas internes de façon mesurée et ciblée est un problème difficile voire impossible à réaliser. Nous l'avons vu dans la chapitre 3. A moins de détériorer physiquement le système, il est difficile de provoquer un aléa interne précis, ou de recréer plusieurs fois un aléa interne donné.

Ainsi, une fois que l'IUT est soumise à ces conditions "extrêmes", on commence à appliquer les séquences de test grâce au testeur, et on évalue le comportement à l'aide des sorties de l'IUT.

Aléas externes

L'insertion d'aléas externes est un problème nettement plus contrôlable. Nous allons en effet travailler directement sur les séquences de test générées auparavant. Nous allons exercer dessus des *mutations*. La séquence résultat ainsi obtenue sera appelée *séquence mutante*.

Dans un premier temps, on laisse la liberté au concepteur de choisir les aléas qu'il souhaite insérer aux séquences, en donnant toutefois un cadre formel précis, pour ne

pas autoriser n'importe quelle modification. Nous allons donc collecter un ensemble de fautes qu'il est possible d'insérer dans les séquences de test. Ensuite, il sera possible d'utiliser cet ensemble pour simuler des scénarii particuliers de défaillance de composants de l'environnement.

Tout d'abord, nous n'autorisons que les modifications sur les entrées des séquences de test. En effet, il n'est pas envisageable de modifier un sortie, puisqu'il s'agit par définition d'une réponse du système (en revanche, l'interprétation de cette réponse sera différente), et que c'est un événement dont nous n'avons pas le contrôle. Par ailleurs, étant donné que nous cherchons à voir si le système est capable de différencier une action vitale d'une autre, et de réagir correctement à cette action vitale (par exemple en renvoyant la bonne action, vitale elle aussi), nous interdisons de modifier toute action d'entrée de la séquence qui se trouve dans l'ensemble des actions d'entrée de la spécification dégradée.

Supposons encore une fois que $\mathcal{S} = (\Sigma,\ S, s^0, C, E)$ est la spécification nominale, et que $\mathcal{S}' = (\Sigma',\ S', s'^0, C', E')$ est la dégradée, toutes deux sous forme de TIOA. Nous avons à notre disposition une séquence de test T sous une forme décrite ci-dessous.

Définition 4.3.2 (Séquence de test temporisée) *Une séquence de test temporisée T extraite d'un TIOA $\mathcal{S} = (\Sigma,\ S, s^0, C, E)$ est un ensemble ordonné de triplets $Tr_i = (a_i, \lambda_i, \delta_i)$, $1 \leq i \leq n$ tels que $a_i \in \Sigma$, λ_i est un ensemble d'horloges à réinitialiser, et δ_i est un ensemble de contraintes d'horloges. On note $T = \{Tr_1.Tr_2...Tr_n\}$ cette séquence. Un triplet peut être vide, dans ce cas on le note $(-,-,-)$. On appelle longueur de T le nombre de triplets non vides dans T.*

Nous nommerons T' la séquence mutante résultat des mutations apportées.

Ainsi, nous proposons les possibilités de fautes suivantes :

1. **Remplacement d'une action d'entrée**
   On propose ici de simuler le fait qu'un autre module envoie une action inattendue à la place de celle que l'IUT attendait. Le concepteur choisit un triplet Tr_i de T tel que $a_i \in \mathcal{I}_\mathcal{S} - \mathcal{I}_{\mathcal{S}'}$. Ensuite, il change l'action a_i avec une action $a' \in \mathcal{I}_\mathcal{S} - \mathcal{I}_{\mathcal{S}'}$. Ainsi, la séquence mutante T' sera :
   $T' = \{Tr_1.Tr_2...\ Tr_{i-1}.Tr'_i.Tr_{i+1}...Tr_n\}$ avec $Tr'_i = (a', \lambda_i, \delta_i)$.
   Ce genre de situation peut arriver par exemple en cas de défaillance d'un module à distance (panne byzantine), ou encore du bus de transmission.
2. **Changer l'instant d'occurrence d'une action d'entrée**
   Cette fois, c'est l'aspect temporel qui nous intéresse. Supposons par exemple que le module qui envoie l'action soit soumis à une charge de travail anormale. Alors, il est envisageable que l'émission de l'action soit retardée. Ainsi, le concepteur choisit un triplet Tr_i de T tel que $a_i \in \mathcal{I}_\mathcal{S} - \mathcal{I}_{\mathcal{S}'}$ et que δ_i soit borné. Ensuite, on change la contrainte d'horloges δ_i par son complémentaire δ', si celui-ci n'est pas vide. En fait, en pratique, le testeur va juste retarder légèrement l'occurrence de l'action d'une petite quantité ϵ, de telle sorte qu'elle dépasse légèrement la borne, et qu'elle arrive plus tard que prévu. La séquence mutante sera alors :
   $T' = \{Tr_1.Tr_2...\ Tr_{i-1}.Tr'_i.Tr_{i+1}...Tr_n\}$ avec $Tr'_i = (a_i, \lambda_i, \delta')$.

3. **Permutation de deux actions d'entrée**
   On simule ici le fait que deux autres modules peuvent avoir des problèmes d'ordonnancement, ou que l'un d'eux soit défaillant. Dans ce cas, cela peut entraîner un changement dans l'ordre d'arrivée des actions. On propose ici simplement de permuter deux actions d'entrée dans la séquence de test. Le concepteur choisit deux triplets Tr_i et Tr_j de T tels que $a_i \in \mathcal{I}_\mathcal{S} - \mathcal{I}_{\mathcal{S}'}$ et $a_j \in \mathcal{I}_\mathcal{S} - \mathcal{I}_{\mathcal{S}'}$. Ensuite on permute les actions a_i et a_j. La séquence mutante devient alors :
   $T' = \{Tr_1.Tr_2...Tr'_i...Tr'_j...Tr_n\}$ avec $Tr'_i = (a_j, \lambda_i, \delta_i)$ et $Tr'_j = (a_i, \lambda_j, \delta_j)$
4. **Ajout d'une action d'entrée inattendue**
   On simule cette fois le fait qu'une action inattendue s'est insérée dans les échanges d'actions entre modules. Concrètement, on insère une action à l'intérieur de la séquence de test. Encore une fois, ce genre de situation est envisageable lorsqu'un module communiquant avec l'IUT est en panne de type byzantin. Le concepteur choisit un triplet Tr_i de T tel que $a_i \in \mathcal{I}_\mathcal{S}$. Il choisit aussi une action $b \in \mathcal{I}_\mathcal{S} - \mathcal{I}_{\mathcal{S}'}$ (celle qu'il veut ajouter). Nous voulons insérer l'action b dans la séquence sans avoir à modifier tout le timing de la séquence. C'est pourquoi nous décidons de l'insérer dans le temps imparti pour l'action a_i, et c'est ce qui explique pourquoi a_i est une entrée, car il faut que ce soit le testeur qui puisse contrôler l'ajout de b avant l'émission de a_i. Finalement, la séquence mutante résultat est :
   $T' = \{Tr_1.Tr_2...Tr_{i-1}.Tr'.Tr'_i.Tr_{i+1}...Tr_n\}$ avec $Tr'_i = (a_i, \lambda_i, \delta_i)$ et $Tr' = (b, -, \delta')$ tel que $\delta' \subset \delta_i$.
5. **Suppression d'une action d'entrée**
   Cette fois, on décide au contraire de supprimer une action d'entrée dans la séquence de test. Ce genre de situation peut aussi se produire en cas de panne par omission ou de panne franche d'un module qui communique avec l'IUT. Il s'agit en fait d'une perte d'information. Ce type de faute peut aussi venir d'un matériel de communication endommagé (un bus par exemple). Le concepteur choisit un triplet Tr_i de T tel que $a_i \in \mathcal{I}_\mathcal{S} - \mathcal{I}_{\mathcal{S}'}$. On supprime simplement ce triplet de la séquence. Ainsi, la séquence mutante est :
   $T' = \{Tr_1.Tr_2...Tr_{i-1}.Tr_{i+1}...Tr_n\}$.

4.3.3 Insertion probabiliste

Cette fois, nous allons essayer de nous placer dans le cadre d'une défaillance d'un module précis qui communique habituellement avec l'IUT. Il faut donc commencer par choisir ce module. Cette approche peut s'inscrire aussi dans le cadre du test d'interopérabilité. Nous venons de voir que les différents types de pannes étaient rangés dans trois catégories, franches, par omission et byzantines, et qu'elles pouvaient être transitoires, intermittentes ou permanentes. Nous allons essayer de reproduire ce genre de comportement à l'intérieur des séquences de test, pour évaluer comment l'IUT réagit. Encore une fois, nous ne pouvons modifier que les entrées. Dans ce genre de scénarii, certains aspects aléatoires sont à considérer, comme l'instant ou la fréquence de la panne. Comme précédemment, on note $T = \{Tr_1.Tr_2...Tr_n\}$ la séquence qui nous sert de base, et T' la séquence mutante résultat des mutations apportées.

Panne franche

On veut simuler la panne franche du module. Cela signifie que pendant une certaine période, il n'envoie absolument aucune information (et ne reçoit rien non plus...). C'est la panne la plus simple. Par ailleurs, nous avons forcément à notre disposition la spécification (nominale ou classique) de ce module. On note $\mathcal{C}$ ce TIOA. Grâce à lui, il est possible de savoir avec précision quelles actions il était susceptible d'envoyer, car il est probable que des sorties de $\mathcal{C}$ soient des entrées de l'IUT. Nous allons donc supprimer dans T toutes les actions d'entrée qui se trouvent dans l'ensemble de sorties de $\mathcal{C}$. Toutefois, les actions vitales ne sont pas supprimées, car elles sont par définition indispensables au fonctionnement, et les retirer n'aurait pas de sens.

Ainsi pour une panne franche, on choisit deux entiers i et j, puis on fabrique T' de la façon suivante :
$\forall k \in \mathbb{N} \mid i \leq k \leq j$,
- si $a_k \in \mathcal{O}_\mathcal{C} - \mathcal{I}_{\mathcal{S}'}$, alors $Tr'_k = (-,-,-)$
- sinon $Tr'_k = Tr_k$

Pour toutes les autres valeurs de k, $Tr'_k = Tr_k$.

Ensuite on distingue les trois cas suivants :
- panne transitoire : on choisit aléatoirement deux valeurs i et j telles que $1 < i < j < n$ et on applique la transformation ci-dessus.
- panne intermittente : on choisit aléatoirement le nombre de pannes transitoires que l'on applique sur la séquence T.
- panne permanente : on choisit aléatoirement une valeur de i telle que $1 < i < n-1$, et on fixe $j = n$ (car le phénomène se produit jusqu'à la fin de la séquence).

Panne par omission

Cette fois, nous devons simuler des pertes d'informations au niveau des séquences de test. Encore une fois, nous ne pouvons agir qu'au niveau des entrées. Ces pertes sont par définition aléatoires, mais concernent uniquement le module défaillant. Nous n'allons donc supprimer que les actions venant de ce module, dont la spécification se note encore $\mathcal{C}$. L'idée ici va être de supprimer dans la séquence T des actions qui appartiennent aux sorties de $\mathcal{C}$ mais non vitales.

Nous allons donc choisir deux entiers i et j, puis fabriquer T' de la façon suivante :

1: **for** $k = 1$ **to** n **do**
2: $Tr'_k := Tr_k$; { on recopie la séquence T dans T' }
3: **end for**
4: on choisit aléatoirement le nombre entier m d'actions que l'on risque de perdre ;
5: **for** $k = 0$ **to** $m - 1$ **do**
6: on choisit aléatoirement une action a de $\mathcal{O}_\mathcal{C} - \mathcal{I}_{\mathcal{S}'}$;
7: on cherche le premier triplet Tr_l entre i et j tel que $a_l = a$;
8: **if** l existe **then**
9: $Tr'_l := (-,-,-)$ {suppression de l'action en question}

10: **end if**
11: **end for**

Ensuite, on distingue les pannes transitoires, intermittentes et permanentes de façon identique à celle expliquée pour les pannes franches : on applique exactement le même procédé.

Panne byzantine

La panne byzantine est le comportement le plus imprévisible que le module puisse avoir. Dans ce genre de panne, il n'obéit plus à aucune règle, et se comporte de façon complètement aléatoire (voir même hostile). Dans notre cas, c'est le scénario qui fera appel au plus grand nombre de paramètres aléatoires. Les mutations proposées ici ressemblent fortement à celles proposées dans le cas de l'insertion d'aléas manuelle. En effet, le comportement de ce module étant parfaitement imprévisible, on se propose de mélanger plusieurs combinaisons d'insertion d'aléas de base. Toutefois, on fera attention de ne modifier que des entrées qui sont aussi des sorties de ce module et qui ne sont pas vitales. On note toujours $\mathcal{C}$ la spécification de ce module.

Ainsi, la séquence mutante T' sera une combinaison choisie aléatoirement des fautes suivantes. Dans tous ces cas, on choisit deux entiers i et j tels que $i < j$. Cette mutation s'inspire fortement de l'insertion manuelle d'aléas vue précédemment. Seul le choix des actions à modifier diffère. Pour faciliter la lecture, nous avons préféré réécrire intégralement la mutation.

- *Remplacement d'une action d'entrée*
  On simule l'envoi d'une action inattendue à la place d'une autre de la part du module. On choisit aléatoirement une action a telle que $a \in \mathcal{O}_\mathcal{C} - \mathcal{I}_{\mathcal{S}'}$ et que $\{\exists k \,|\, i \leq k \leq j$ et $a_k \in \mathcal{I}_\mathcal{S}$ et $a_k = a\}$. Ensuite on remplace cette action dans T par une action $a' \in \mathcal{O}_\mathcal{C} - \mathcal{I}_{\mathcal{S}'}$ choisie aussi aléatoirement. Ainsi, la séquence mutante T' devient :
  $T' = \{Tr_1...Tr_i... \; Tr_{k-1}.Tr'_k.Tr_{k+1}...Tr_j...Tr_n\}$ avec $Tr'_k = (a', \lambda_k, \delta_k)$.
- *Changement de l'instant d'occurrence d'une action d'entrée*
  Cette fois, c'est l'instant d'une action venant du module défaillant que nous allons modifier. Comme avant, on choisit aléatoirement une action a telle que $a \in \mathcal{O}_\mathcal{C} - \mathcal{I}_{\mathcal{S}'}$ et que $\{\exists k \,|\, i \leq k \leq j$ et $a_k \in \mathcal{I}_\mathcal{S}$ et $a_k = a\}$. Ensuite, on modifie dans le triplet correspondant de T la valeur de contrainte d'horloge δ_k par son complémentaire δ' s'il n'est pas à vide. Ainsi, on obtient :
  $T' = \{Tr_1...Tr_i... \; Tr_{k-1}.Tr'_k.Tr_{k+1}...Tr_j...Tr_n\}$ avec $Tr'_k = (a_k, \lambda_k, \delta')$.
- *Permutation de deux actions d'entrée*
  On va inverser dans la séquence de test deux actions qui viennent du module défaillant. On va donc choisir aléatoirement deux actions distinctes a et b telles que $a \in \mathcal{O}_\mathcal{C} - \mathcal{I}_{\mathcal{S}'}$ et $\{\exists k \,|\, i \leq k \leq j$ et $a_k \in \mathcal{I}_\mathcal{S}$ et $a_k = a\}$ et $b \in \mathcal{O}_\mathcal{C} - \mathcal{I}_{\mathcal{S}'}$ et que $\{\exists l \,|\, i \leq l \leq j$ et $a_l \in \mathcal{I}_\mathcal{S}$ et $a_l = b\}$. Ensuite, on permute ces deux actions dans T, ce qui nous donne la séquence mutante :
  $T' = \{Tr_1...Tr_i...Tr'_k...Tr'_l...Tr_j...Tr_n\}$ avec $Tr'_k = (a_l, \lambda_k, \delta_k)$ et $Tr'_l = (a_k, \lambda_l, \delta_l)$.
- *Ajout d'une action d'entrée inattendue*
  Parmi les actions que le module défaillant envoie, on va en insérer d'autres, comme vu dans le cas des insertions manuelles. Le principe sera identique, sauf que l'entrée

insérée appartient aux sorties de $\mathcal{C}$. On choisit donc aléatoirement un triplet Tr_k de T tel que $i \leq k \leq j$ et $a_k \in \mathcal{I}_\mathcal{S}$. On choisit aussi aléatoirement une action $b \in \mathcal{O}_\mathcal{C}$. Nous allons insérer l'action b dans le temps imparti pour a_k. Ainsi, la séquence mutante devient :
$T' = \{Tr_1.Tr_2...Tr_i...Tr_{k-1}.Tr'.Tr'_k.Tr_{k+1}...Tr_j...Tr_n\}$ avec $Tr'_k = (a_k, \lambda_k, \delta_k)$ et $Tr' = (b, -, \delta')$ tel que $\delta' \subset \delta_k$.

– *Suppression d'une action d'entrée*
On décide cette fois de supprimer une action venant normalement du module défaillant. Ainsi, on choisit aléatoirement un triplet Tr_k de T tel que $i \leq k \leq j$ et $a_k \in \mathcal{O}_\mathcal{C} - \mathcal{I}_{\mathcal{S}'}$. Ensuite, on supprime tout simplement ce triplet de la séquence T, ce qui nous donne :
$T' = \{Tr_1.Tr_2...Tr_i...Tr_{k-1}.Tr_{k+1}...Tr_j...Tr_n\}$.

Encore une fois, les choix des valeurs de i et j sont faits comme décrit dans le paragraphe sur l'insertion probabiliste dans le cadre d'une panne franche, selon que l'on souhaite reproduire une panne transitoire, intermittente ou permanente.

4.4 Analyse du comportement du système

Cette section va nous permettre de faire un lien entre les réponses obtenues de la part de l'IUT et la robustesse du système. Dans notre travail, le verdict sera le plus simple possible : assez robuste ou pas assez robuste, par rapport aux aléas insérés. Nous allons tout d'abord voir comment situer le comportement du système par rapport aux critères de robustesse, puis nous définirons une relation d'acceptation qui permettra de faire le lien entre les traces d'exécution obtenues et les spécifications nominale et dégradée.

Degré de robustesse

La principale difficulté dans notre approche, c'est que l'on ne connaît pas à l'avance le comportement précis que va adopter le système. C'est encore ce qui explique que les séquences de test puissent être tirées de la spécification nominale. En effet, en cas de situation critique, il se peut que le système abandonne certaines fonctionnalités non vitales, ou qu'il passe dans un mode "dégradé". Mais comme il s'agit d'un test en boîte noire, il est difficile d'évaluer précisément à quel instant l'IUT décide de passer dans un mode dégradé. Par ailleurs, rappelons que notre spécification dégradée sert juste de "collection" des actions vitales. Il se peut donc que l'IUT adopte un comportement non nominal, mais toutefois bien plus élaboré que dans la spécification dégradée.

Nous ne savons pas précisément à l'avance le comportement que l'IUT va adopter, mais en revanche, nous en connaissons en quelques sortes les "limites" autorisées. Nous allons donc considérer qu'un système est assez robuste (par rapport aux aléas insérés), si son comportement se trouve entre celui de la spécification nominale (incluse) et celui de la spécification dégradée (incluse). Par la suite, nous définirons plus précisément ce que nous entendons par "se trouve" entre les deux spécifications, autrement dit comment situer un comportement par rapport à deux spécifications (une nominale et une dégradée). La figure 4.9 résume cette idée. Supposons par exemple qu'on interroge le système sur une

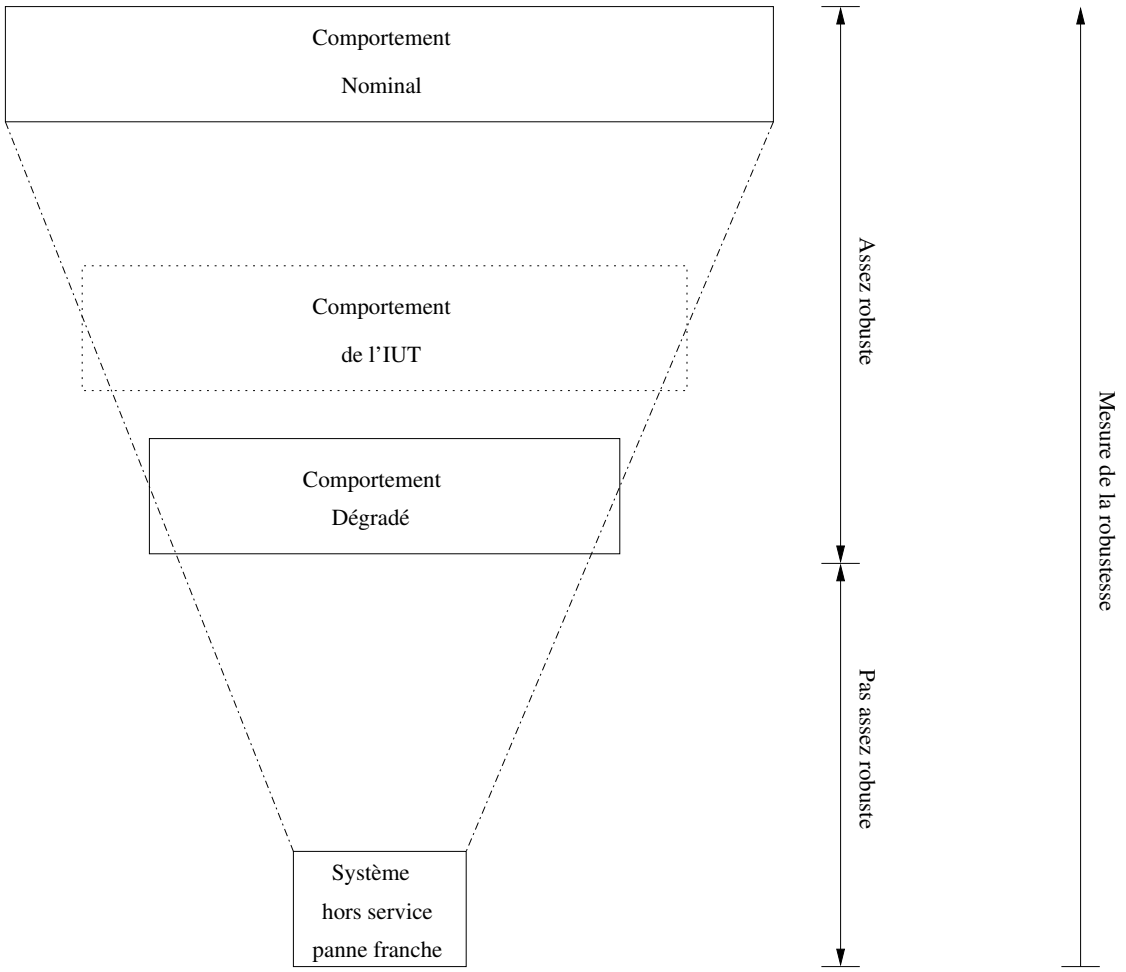


FIG. 4.9 – Degré de robustesse

donnée précise, et que le temps de réponse soit de 5s dans la spécification nominale, et de 10s dans la dégradée. Alors, si la réponse arrive au bout de 8s, le délai sera considéré comme acceptable. En revanche, si elle arrive au bout de 12s, le système sera considéré comme insuffisamment robuste.

Traces d'exécution

Lors de l'application des séquences de test sur l'IUT (ie lors de la session de test), le testeur prend soin d'enregistrer toutes les entrées et toutes les sorties, ainsi que leur instant d'occurrence, au niveau du point de communication avec l'IUT. Ces instants vont être datés avec une horloge absolue unique, initialisée au début de la session de test.

L'application d'une séquence de test sur l'IUT va être enregistrée comme une succession d'actions et d'instants, ce qui va nous donner la trace d'exécution complète du cas de test. Ainsi, nous avons décidé de représenter une trace temporisée sous la forme d'un mot temporisé. Nous rappelons quelques définitions de [AD94].

Définition 4.4.1 (Séquence temporisée) *Une séquence temporisée $\tau = \tau_1\tau_2...$ est une séquence infinie de valeurs $\tau_i \in \mathbb{R}$ avec $\tau_i > 0$, satisfaisant les contraintes suivantes :*
- *Monotonicité : τ augmente de façon strictement monotone ; ie, $\tau_i < \tau_{i+1}$ pout tout $i \geq 1$.*
- *Progression : pour tout $t \in \mathbb{R}$, il existe $i \geq 1$ tel que $\tau_i > t$.*

Définition 4.4.2 (Mot temporisé) *Un mot temporisé sur un alphabet Σ est une paire (σ, τ) où $\sigma = \sigma_1\sigma_2...$ est un infini sur Σ et τ est une séquence temporisée.*

Un exemple possible de trace d'exécution tirée d'une IUT se comportant comme spécifié figure FIG. 4.7 pourrait être :

$(?a, 3) \rightarrow (!b, 8) \rightarrow (!c, 24)$.

Relation d'acceptation

Il reste maintenant à faire le lien entre les traces d'exécution sous forme d'ensemble de mots temporisés et les spécifications du système, pour décider si le comportement est assez robuste. On note T cet ensemble de mots temporisés, qui correspond au résultat de l'ensemble de la session de test enregistré par le testeur.

Le but de cette relation est de décider si une trace d'exécution est acceptée au sens de la robustesse par la spécification dégradée $\mathcal{S}'$. En effet, la difficulté va être de faire en sorte que notre relation ne "s'intéresse" qu'aux aspects vitaux du système, décrits dans $\mathcal{S}'$, tout en "ignorant" les aspects non vitaux décrits dans $\mathcal{S}$. A la fin, si tous les mots temporisés de T sont acceptés par $\mathcal{S}'$, alors l'IUT est considérée comme assez robuste.

Nous allons nous inspirer de la notion d'"exécution" définie dans [AD94] pour écrire notre relation d'acception. Cette notion d'exécution permet de faire le lien entre un mot temporisé et un automate temporisé. Il faudra toutefois l'adapter à notre situation.

Définition 4.4.3 (Exécution) *Une exécution r, notée $(\bar{s}, \bar{\nu})$ d'un TIOA $A = (\Sigma_A, L_A, l_A^0, C_A, E_A)$ sur un mot temporisé (σ, τ) est un séquence de la forme :*
$r : \ < s_0, \nu_0 > \xrightarrow[\tau_1]{\sigma_1} < s_1, \nu_1 > \xrightarrow[\tau_2]{\sigma_2} < s_2, \nu_2 > \xrightarrow[\tau_3]{\sigma_3} ...$
avec $s_i \in L_A$ et $\nu_i \in [C_A \rightarrow \mathbb{R}]$, pour tout $i \geq 0$, satisfaisant les conditions suivantes :
- *Initiation : $\nu_0(x) = 0$ pour tout $x \in C_A$.*
- *Consécution : pour tout $i \geq 1$, il existe une transition dans E_A de la forme $< s_{i-1}$, s_i , σ_i, λ_i, $\delta_i >$ telle que $(\nu_{i-1} + \tau_i - \tau_{i-1})$ satisfait δ_i et ν_i soit égal à $[\lambda_i \longmapsto 0]$ $(\nu_{i-1} + \tau_i - \tau_{i-1})$.*

Nous cherchons à ce stade à faire l'exécution de la spécification dégradée $\mathcal{S}'$ sur une trace enregistrée. L'idée est de suivre cette exécution sur le TIOA $\mathcal{S}'$ et d'analyser sur quel état cette exécution se termine. Mais, pour procéder ainsi, il faut que l'exécution existe, ce qui est peu probable étant donné que l'IUT peut adopter un comportement bien plus évolué que dans $\mathcal{S}'$, et de plus, les séquences de test sont obtenues à partir de la spécification nominale $\mathcal{S}$, ce qui implique un grand nombre d'actions non prévues dans $\mathcal{S}'$.

Prenons l'exemple du TIOA figure FIG. 4.10 et supposons qu'il s'agisse de la spécification dégradée $\mathcal{S}'$ d'un système, n'ayant qu'une horloge x, et que l'alphabet de la spécification nominale soit :

$$\Sigma = \{?a, !b, ?c, !d, ?e, ?f, !g\}.$$

Supposons aussi qu'un cas de test nous ait donné la trace suivante :

$$T = (?a, 2)\ (!b, 12)\ (?f, 14)\ (!g, 16)\ (?f, 19)\ (?c, 25)\ (!d, 29).$$

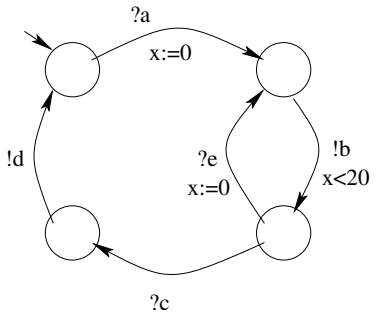


FIG. 4.10 – Un exemple de spécification dégradée

On voit sur cet exemple qu'il est impossible de trouver une exécution de $\mathcal{S}'$ sur T dans l'état actuel. En effet, l'entrée $?f$ n'est pas prévue par $\mathcal{S}'$ alors qu'elle se trouve dans T. Et pourtant, de façon intuitive, on remarque que les actions vitales exprimées dans $\mathcal{S}'$ ont bien eu lieu, et dans le bon ordre, aux instants prévus dans T. Vraisemblablement, T devrait être accepté.

Pour remédier à ce problème, nous proposons de transformer le TIOA $\mathcal{S}'$ de telle sorte que le résultat de cette transformation soit complètement spécifié. Cette transformation est la suivante : on ajoute à chaque état de $\mathcal{S}'$ des transitions boucles (ie revenant sur le même état) de deux types :

- une transition contenant toutes les actions de l'alphabet de $\mathcal{S}$ qui n'étaient pas déjà sur une transition partant de cet état ;
- pour chaque transition t partant de cet état (avant la transformation ci-dessus), on ajoute une transition boucle étiquetée par la même action, mais avec comme contrainte d'horloge le complémentaire de celle de t.

Le résultat de cette transformation se note $\mathcal{S}'_{compl}$. Remarquons au passage que $\mathcal{S}'_{compl}$ n'est absolument plus déterministe. Ainsi, $\mathcal{S}'_{compl}$ permet l'exécution de tout mot temporisé dont l'alphabet correspond à la spécification nominale. Bien entendu, il reste maintenant à trouver un moyen de savoir si une trace est ou non acceptée au sens de la robustesse. La figure FIG. 4.11 montre cette transformation appliquée au TIOA de la figure FIG. 4.10.

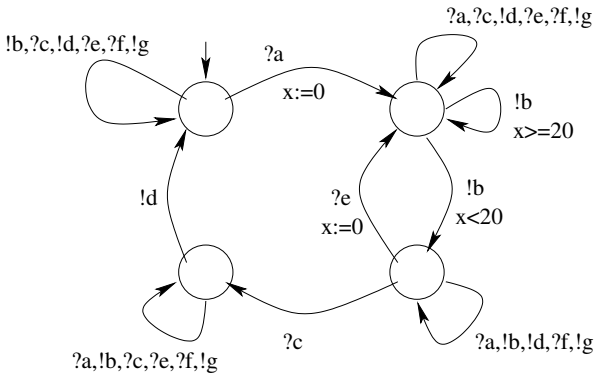


FIG. 4.11 – Le TIOA de la figure 4.10 complété

Le principe de notre méthode est d'envoyer des entrées à l'IUT, et s'il s'agit d'une entrée vitale (ie dans la spécification dégradée $\mathcal{S}'$), alors il faut s'assurer que l'IUT a bien renvoyé les réponses prévues dans $\mathcal{S}'$. L'idée est donc de parcourir le TIOA ${\mathcal{S}'}_{compl}$, c'est à dire de l'exécuter sur la trace, et de s'assurer qu'à la fin de l'exécution, on retombe bien sur un état "d'attente" d'une nouvelle entrée, qui est en fait un état contrôlable.

Si lors de cette exécution l'IUT reçoit une action d'entrée alors qu'elle se trouve sur un état non contrôlable, alors on reste sur le même état de ${\mathcal{S}'}_{compl}$.

Supposons cette fois que lors de cette exécution, l'IUT reçoive une action d'entrée, et qu'à ce moment là, elle soit sur un état contrôlable, alors deux cas se présentent :

- cette action n'est pas attendue par la spécification dégradée $\mathcal{S}'$, ce n'est pas une action vitale. Alors, l'exécution de ${\mathcal{S}'}_{compl}$ reste bloquée sur le même état, attendant un entrée vitale ;
- cette action est attendue par la spécification dégradée $\mathcal{S}'$. Alors l'exécution de ${\mathcal{S}'}_{compl}$ passe la transition. Dans ce cas, le seul moyen d'atteindre à nouveau un état contrôlable dans l'exécution de ${\mathcal{S}'}_{compl}$, c'est que l'IUT donne les réponses attendues par $\mathcal{S}'$, et avec le bon timing. Dans le cas contraire, l'exécution restera indéfiniment bloquée sur un état non contrôlable.

Finalement, nous considérerons qu'une trace w (sous forme de mot temporisé) est acceptée par $\mathcal{S}'$ au sens de la robustesse si l'exécution de ${\mathcal{S}'}_{compl}$ sur w se termine sur un état contrôlable, ce qui donne la relation suivante :

Définition 4.4.4 ($accept_{\mathcal{R}}$) *Un mot temporisé fini $w = (\sigma, \tau)$ est accepté au sens de la robustesse par un TIOA $\mathcal{S}'$, noté $\mathcal{S}' accept_{\mathcal{R}} w$ s'il existe une exécution $r = (\bar{s}, \bar{\nu})$ de ${\mathcal{S}'}_{compl}$ sur w telle que le dernier état de ${\mathcal{S}'}_{compl}$ atteint par r soit contrôlable. Par extension, pour un ensemble de mots temporisés Υ, on note $\mathcal{S}' accept_{\mathcal{R}} \Upsilon$ si et seulement si $\forall w \in \Upsilon$, $\mathcal{S}' accept_{\mathcal{R}} w$.*

Pour que l'IUT soit considérée comme assez robuste, il faut donc que toutes les traces obtenues après application des séquences sur l'IUT soient acceptées au sens de la robustesse par $\mathcal{S}'$, dans le première étape (aléas internes) et aussi dans le deuxième phase (aléas externes).

4.5 Exemple

Danc cette section, nous allons illustrer la méthode de test de robustesse que nous venons de présenter, et ce à l'aide d'un exemple complet. Nous supposons ainsi que le concepteur du système nous a fourni une spécification $\mathcal{S}$ d'un robot simple (inspiré du célèbre jeu de la tortue), donnée figure FIG. 4.12. Pour résumer, ce système peut tourner dans un sens ou aller tout droit, et peut sur demande envoyer sa température ou sa position, à condition d'être dans le bon mode. En cas, de demande de la part de l'environnement, que ce soit la température ou la position, le système possède un temps limité pour réagir.

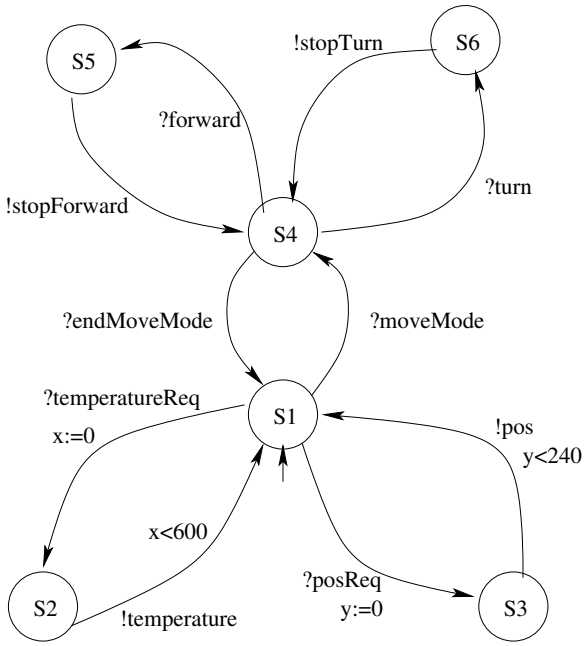


FIG. 4.12 – La spécification nominale $\mathcal{S}$ de notre robot

De même, une spécification dégradée $\mathcal{S}'$ a été fournie figure 4.13, regroupant les aspects considérés comme vitaux pour le système. Ici, il a été décidé que lorsqu'on demande sa position ou la température au système, alors la réponse doit impérativement être donnée, avec toutefois un délai supplémentaire par rapport à la spécification nominale pour la position.

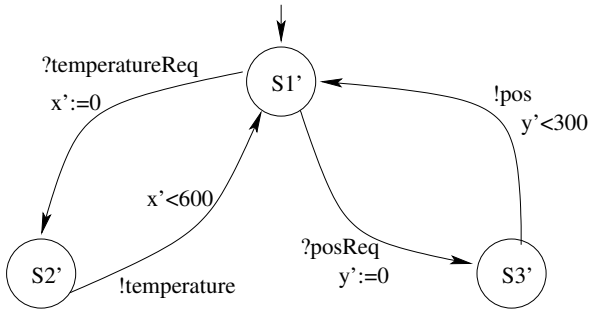


FIG. 4.13 – La spécification dégradée $\mathcal{S}'$ de notre robot

Supposons maintenant que la méthode de génération de séquences de test nous ait donné les séquences de test de la figure FIG. 4.14. Rappelons que de façon automatique, il existe une borne pour toute transition où aucune contrainte de temps n'est précisée.

- $(?posReq, y := 0, -)$, $(!pos, -, y < 240)$, $(?tempReq, x := 0, -)$, $(!temperature, -, x < 600)$,$(?moveMode, -, -)$
- $(?moveMode, -, -)$, $(?turn, -, -)$, $(!stopTurn, -, -)$, $(?endMoveMode, -, -)$, $(?posReq, y := 0, -)$, $(!pos, -, y < 240)$

FIG. 4.14 – Exemple de séquences de test générées à partir de $\mathcal{S}$

4.5.1 Première étape : aléas internes

Dans un premier temps, on ne touche pas aux séquences de test et on soumet l'IUT à des radiations pendant qu'on applique ces séquences. Supposons qu'à la fin cette étape, nous ayons obtenu les traces de la figure 4.15, et on note Seq_1 cet ensemble de traces.

- $seq_{11} = (?posReq, 3) \rightarrow (!pos, 203) \rightarrow (?tempReq, 208) \rightarrow (!temperature, 708) \rightarrow (?moveMode, 720)$
- $seq_{12} = (?moveMode, 3) \rightarrow (?turn, 8) \rightarrow (!stopTurn, 28) \rightarrow (?endMoveMode, 33) \rightarrow (?posReq, 48) \rightarrow (!pos, 248)$

FIG. 4.15 – Un exemple de traces d'exécution

Si on effectue l'exécution sur les deux traces obtenues de $\mathcal{S}'_{compl}$, qui est la transformation de $\mathcal{S}'$ en TIOA complètement spécifié, et donné figure FIG. 4.16, on voit qu'elles se terminent toutes les deux sur $S1'$, qui est un état contrôlable. Donc nous avons bien $\mathcal{S}' accept_{\mathcal{R}} Seq_1$, et ainsi la première phase est un succès.

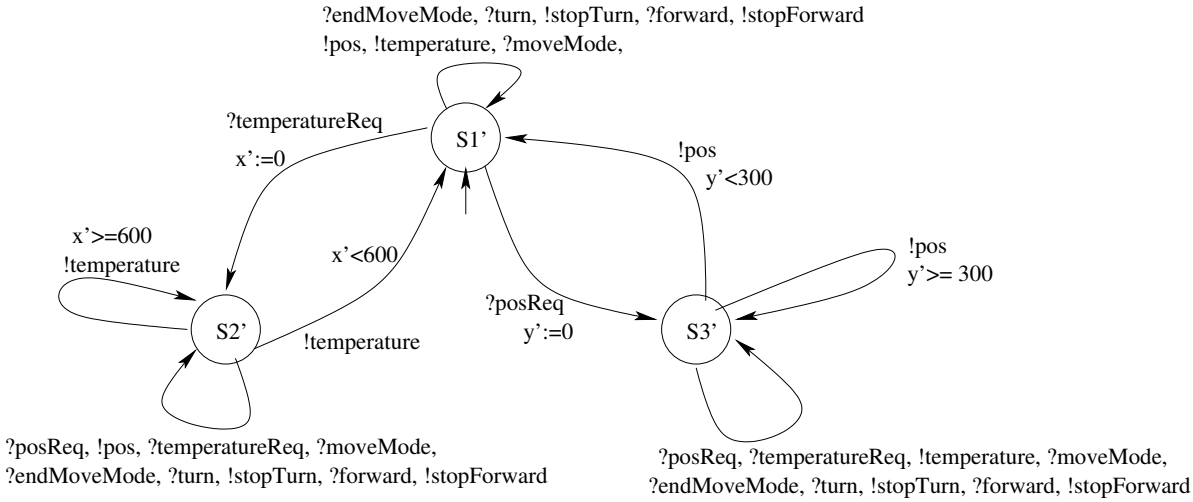


FIG. 4.16 – $\mathcal{S}'_{compl}$: le TIOA de la spécification dégradée $\mathcal{S}'$ (4.13) complété

4.5.2 Deuxième étape : aléas externes

Dans cette étape, on réutilise les mêmes séquences de test de la figure FIG. 4.14. Sur ces deux séquences va être appliquée un insertion d'aléa(s) qui peut être faite de façon manuelle ou de façon probabiliste. Supposons qu'à la suite de cette insertion, les séquences de test mutantes soient celles données figure FIG. 4.17. Dans la première séquence, l'action $?turn$ a été ajoutée. Dans la deuxième, on a effectué une permutation de deux actions : $?turn$ et $?moveMode$.

- $(?posReq, y := 0, -)$, $(!pos, -, y < 240)$, $(?\mathbf{turn}, -, -)$, $(?tempReq, x := 0, -)$, $(!temperature, -, x < 600)$,$(?moveMode, -, -)$
- $(?\mathbf{turn}, -, -)$, $(?\mathbf{moveMode}, -, -)$, $(!stopTurn, -, -)$, $(?endMoveMode, -, -)$, $(?posReq, y := 0, -)$, $(!pos, -, y < 240)$

FIG. 4.17 – Exemple de séquences mutantes

Supposons que l'application de ces séquences sur l'IUT donne comme résultat les traces de la figure FIG 4.18. On note Seq_2 cet ensemble.

- $seq_{21} = (?posReq, 5) \rightarrow (!pos, 201) \rightarrow (?turn, 220) \rightarrow (?tempReq, 278) \rightarrow (!temperature, 403) \rightarrow (?moveMode, 700)$
- $seq_{22} = (?turn, 12) \rightarrow (?moveMode, 45) \rightarrow (?endMoveMode, 62) \rightarrow (?posReq, 100) \rightarrow (!pos, 380)$

FIG. 4.18 – Un exemple de traces d'exécution

Encore une fois, si on effectue l'exécution de $\mathcal{S}'_{compl}$ sur seq_{21} et seq_{22}, on se rend compte qu'elles se terminent toutes deux sur l'état $S1'$, qui est contrôlable. Et ainsi, on

a bien $\mathcal{S}' accept_{\mathcal{R}} Seq_2$, ce qui signifie que le deuxième étape a aussi un verdict positif, et que l'on peut estimer le système comme assez robuste par rapport aux aléas insérés.

Supposons cette fois que l'application des séquences de test sur l'IUT ait donné les traces de la figure FIG 4.19 au lieu de celles de la figure FIG 4.18. On note Seq'_2 cet ensemble.

- $seq'_{21} = (?posReq, 5) \rightarrow (!temperature, 201) \rightarrow (?turn, 220) \rightarrow (?tempReq, 278) \rightarrow (!temperature, 403) \rightarrow (?moveMode, 700)$
- $seq'_{22} = (?turn, 12) \rightarrow (?moveMode, 45) \rightarrow (?endMoveMode, 62) \rightarrow (?posReq, 100) \rightarrow (!pos, 380)$

FIG. 4.19 – Un exemple de traces d'exécution

Alors lorsqu'on pratique l'exécution de $\mathcal{S}'_{compl}$ sur seq'_{21}, l'erreur de $(!temperature, 201)$ provoque l'absence de passage de la transition correspondante, ce qui a pour conséquence que l'on reste bloqué sur l'état $S3'$, attendant indéfiniment l'action $!pos$. C'est le cas car l'action $!pos$ est une action vitale. A la fin de l'exécution, on se trouve donc dans l'état $S3'$ qui n'est pas contrôlable. Donc la relation $\mathcal{S}' accept_{\mathcal{R}} Seq'_2$ est fausse, et le système est considéré comme insuffisamment robuste à cause de seq'_{21}. On remarque que le même résultat aurait eu lieu si l'action $!pos$ avait bien été reçue, mais en retard.

4.6 Une autre approche : générer les séquences de test à partir de la spécification dégradée

Nous proposons ici une autre approche qui permet de tester les aspects vitaux du système. Le principe cette fois va être de faire en sorte que toutes les entrées vitales du système soient testées, ainsi que leurs réponses correspondantes. Encore une fois, nous allons aussi appliquer au système des entrées inattendues. Il faudra donc encore que l'IUT soit capable de différencier les entrées vitales des autres, et par ailleurs, il faudra contrôler que les actions vitales ont bien été exécutées, tout en acceptant que des sorties considérées comme non vitales puissent être émises. Le but de cette méthode est de vérifier que l'IUT ne sera pas perturbée par des actions d'entrée non vitales, mais qui vont arriver de façon complètement imprévue.

4.6.1 Principe général

On part encore avec deux spécifications du système, une spécification *nominale* $\mathcal{S}$ et une *dégradée* $\mathcal{S}'$ (pour les actions vitales), toutes deux sous forme de TIOA, et respectant exactement les mêmes caractéristiques et hypothèses que dans l'approche précédente. Les hypothèses sont donc les suivantes :

- les spécifications nominale $\mathcal{S}$ et dégradée $\mathcal{S}'$ sont déterministes, vérifiées, fortement connexes, minimales,

- l'implantation du système est un TIOA complet au niveau des entrées,
- toute transition de $\mathcal{S}$ et $\mathcal{S}'$ étiquetée par une action de sortie est bornée dans le temps,
- les horloges de $\mathcal{S}$ et de $\mathcal{S}'$ sont indépendantes,
- $\mathcal{S}' \subset_{\mathcal{D}} \mathcal{S}$.

Cette fois, on utilise la spécification dégradée $\mathcal{S}'$ pour générer les séquences de test, ce qui signifie en fait que nous partons cette fois des actions vitales pour construire nos séquences de test. Nous utilisons une méthode de génération de séquences basée sur les TIOA. Les méthodes citées précédemment ([EDK02] et [SVD01]) sont toujours utilisables. A la suite de cette génération, nous avons à notre disposition un ensemble T de séquences de test. Nous appliquons aussi les mêmes hypothèses sur ces séquences, et cela pour les mêmes raisons que la méthode précédente. Ainsi, une séquence doit être extraite à partir d'un chemin de la spécification dégradée qui :

- commence par une action d'entrée ;
- finit sur un état contrôlable.

Par ailleurs, l'architecture de test est inchangée.

4.6.2 Intégration des aléas

Nous proposons de séparer la phase pour les aléas internes de celle pour les aléas externes. La phase pour les aléas internes reste inchangée par rapport à la méthode précédente, à part bien sûr que les séquences appliquées sont différentes, puisque générées à partir de la spécification dégradée $\mathcal{S}'$. En résumé, on applique directement les séquences générées à partir de $\mathcal{S}'$ sur l'IUT, sans la moindre modification sur ces séquences, et on soumet en même temps l'IUT à des radiations.

Dans un deuxième temps, on s'intéresse aux aléas externes, qui seront encore une fois ajoutés dans les séquences de test. Pour intégrer ces aléas, nous avons comme base de travail un ensemble de séquences, et nous allons donc appliquer l'insertion sur chacune d'elles. Encore une fois, on notera $T = \{Tr_1.Tr_2...Tr_n\}$ la séquence obtenue par la méthode de génération et T' la séquence mutante.

La particularité de T, c'est qu'elle ne contient que des actions vitales. Cette donnée change complètement le mode d'insertion des aléas dans la séquence. Tout d'abord il n'est plus question de retirer ou changer une action de T, puisque T est déjà réduite au minimum. De même, changer l'instant d'occurrence d'une action ou bien échanger deux actions n'est plus possible pour les mêmes raisons. Finalement, l'ajout d'actions (non vitales) dans la séquence de test est la seule possibilité qu'il nous reste.

L'ajout de ce genre d'aléa est assez simple, on va choisir (de façon manuelle ou aléatoire) un triplet Tr_i de T tel que $a_i \in \mathcal{I}_{\mathcal{S}'}$. Juste avant d'envoyer cette action, le testeur va envoyer une action non vitale. Ainsi, on choisit aussi (manuellement ou aléatoirement) une action $b \in \mathcal{I}_{\mathcal{S}} - \mathcal{I}_{\mathcal{S}'}$, qui devra être émise dans le temps imparti pour a_i. Finalement, la séquence mutante résultat est :
$T' = \{Tr_1.Tr_2...Tr_{i-1}.Tr'.Tr'_i.Tr_{i+1}...Tr_n\}$ avec $Tr'_i = (a_i, \lambda_i, \delta_i)$ et $Tr' = (b, -, \delta')$ tel que $\delta' \subset \delta_i$.
Ensuite on reproduit cette opération un certain nombre de fois (aléatoire ou manuel) sur

la même séquence. A la fin, nous obtenons une séquence avec les entrées vitales, les sorties vitales correspondantes, et un certain nombre d'entrées superflues et inattendues.

4.6.3 Verdict du test

Dans les étapes avec aléas internes ou externes, les séquences de test sont appliquées à l'IUT. Pour juger de la robustesse d'après le résultat, il faut :

- que l'IUT accepte toutes les entrées non vitales sans s'en préoccuper ;
- que l'IUT accepte les actions vitales et les gère correctement ;
- que l'on tolère les sorties superflues venant de l'IUT (elles ne doivent pas bloquer l'exécution).

Ce genre de verdict est le même que dans la méthode précédente, donc il est possible de réutiliser la même relation, $accept_{\mathcal{R}}$ pour déterminer si le système est assez robuste. Ainsi, on dira encore une fois que le système est assez robuste si toutes ses traces obtenues en appliquant les séquences de test sur l'IUT sont acceptées au sens de la robustesse par $\mathcal{S}'$, dans la première étape et dans la deuxième.

Cette autre méthode peut se résumer par le cheminement ci-dessous.

1. `Génération de séquences de test à partir de` $\mathcal{S}'$
2. `Début d'application de radiations sur l'IUT`
3. `Application des séquences sur l'IUT`
4. `Fin d'application de radiations sur l'IUT`
5. `Analyse des résultats et verdict partiel`
6. `Ajout d'aléas aux séquences de test`
7. `Application des séquences mutantes sur l'IUT`
8. `Analyse des résultats et verdict final`

Dans cette méthode, le fait que les séquences soient générées à partir de la spécification dégradée implique que les séquences de test sont plus courtes, et donc que la génération des séquences ainsi que les jeux de test seront plus rapides. En effet, par définition, la spécification dégradée possède un nombre d'états inférieur ou égal à celui de la nominale. En revanche, cette méthode est plus ciblée sur les aspects vitaux du système, et teste moins les aspects "superflus".

4.6.4 Exemple

Reprenons les spécifications de l'exemple section 4.5. Le spécification nominale $\mathcal{S}$ est donnée figure FIG. 4.12 et la spécification dégradée FIG. 4.13 que nous redonnons ici par commodité de lecture.

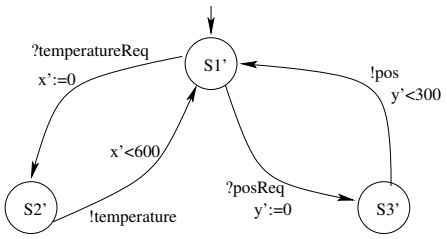


FIG. 4.20 – La spécification dégradée $\mathcal{S}'$ de notre robot

Un exemple de séquences de test générées à partir de $\mathcal{S}'$ pourrait être :

- $(?posReq, y := 0, -)$, $(!pos, -, y < 300)$, $(?tempReq, x := 0, -)$, $(!temperature, -, x < 600)$
- $(?posReq, y := 0, -)$, $(!pos, -, y < 300)$, $(?tempReq, x := 0, -)$, $(!temperature, -, x < 600)$, $(?posReq, y := 0, -)$, $(!pos, -, y < 300)$

FIG. 4.21 – Exemple de séquences de test générées à partir de $\mathcal{S}'$

Première étape

Dans un premier temps, cette séquence est appliquée telle quelle à l'IUT, en présence de radiations. Un ensemble de traces possibles (noté Seq_1) pourrait être :

- $seq_{11} = (?posReq, 3) \rightarrow (!pos, 203) \rightarrow (?tempReq, 208) \rightarrow (!stopTurn, 502) \rightarrow (!temperature, 708)$
- $seq_{12} = (?posReq, 6) \rightarrow (!pos, 187) \rightarrow (?tempReq, 200) \rightarrow (!temperature, 708) \rightarrow (?posReq, 789) \rightarrow (!pos, 1000)$

FIG. 4.22 – Un exemple de traces d'exécution

On peut voir que seq_{11} et seq_{12} sont acceptés au sens de la robustesse par $\mathcal{S}'$ (en effet l'exécution de seq_{11} et seq_{12} sur $\mathcal{S}'_)compl$ termine sur $S1'$ qui est contrôlable). Donc la première étape donne un verdict positif.

Deuxième étape

Lors de cette étape, des aléas externes sont insérés dans les séquences de test. Un exemple de séquences mutantes obtenues à partir de la génération de séquences précédente pourrait être :

- $(?forward, -, -)$, $(?posReq, y := 0, -)$, $(!pos, -, y < 300)$, $(?turn, -, -)$, $(?tempReq, x := 0, -)$, $(!temperature, -, x < 600)$
- $(?posReq, y := 0, -)$, $(!pos, -, y < 300)$, $(?turn, -, -)$, $(?forward, -, -)$ $(?tempReq, x := 0, -)$, $(!temperature, -, x < 600)$, $(?posReq, y := 0, -)$, $(!pos, -, y < 300)$

FIG. 4.23 – Exemple de séquences mutantes

Supposons que l'application de ces séquences de test sur l'IUT est donné l'ensemble Seq_2 de traces d'exécution suivant :

- $seq_{21} = (?posReq, 3) \rightarrow (!pos, 203) \rightarrow (?tempReq, 208) \rightarrow (!stopTurn, 708)$
- $seq_{22} = (?posReq, 6) \rightarrow (!pos, 187) \rightarrow (!stopTurn, 195) \rightarrow (?tempReq, 200) \rightarrow (!stopTurn, 371) \rightarrow (!temperature, 708) \rightarrow (?posReq, 789) \rightarrow (!pos, 1000)$

FIG. 4.24 – Un exemple de traces d'exécution

Sur ces traces, on peut voir que seq_{22} est acceptée au sens de la robustesse par $\mathcal{S}'$. En effet son exécution sur ${\mathcal{S}'}_{compl}$ termine sur $S1'$ qui est contrôlable. En revanche, seq_{21} n'est pas acceptée car il manque la réponse à l'action $?tempReq$ (soit $!tempReq$) à la fin de la séquence, ce qui signifie que l'exécution se termine sur l'état $S2'$ qui n'est pas contrôlable.

4.7 Conclusion

Dans ce chapitre, nous avons proposé deux méthodes pour tester la robustesse d'un système temps-réel. Dans la première, nous utilisons la spécification nominale du système pour générer des séquences de test temporisées, puis nous les appliquons sur l'IUT en enregistrant les traces d'exécution. Après la session de test, nous analysons celles-ci à l'aide d'une relation d'acceptation que nous avons définie, et nous exprimons un verdict pour la robustesse. Dans le deuxième méthode, le principe est similaire, mais cette fois les séquences sont générées à partir de la spécification dégradée.

La première possède l'avantage majeur d'utiliser tous les événements du système, et les mutations s'opèrent sur des événements réels et prévus, ce qui est plus proche de la réalité. En revanche, la deuxième se concentre plus sur les aspects uniquement vitaux, ce qui implique des mutations d'actions moins élaborées étant donné qu'elles ne se basent pas sur les actions prévues du système. L'avantage de cette seconde méthode par rapport à la première, c'est que les séquences générées sont plus courtes.

Pour réaliser ces méthodes de test en pratique, la difficulté principale est d'écrire la spécification dégradée du système. En effet, il peut s'avérer difficile de décider si une action est considérée comme vitale. D'autre part, il est préférable de considérer des systèmes conçus pour gérer prioritairement certaines actions (vitales) par rapport à d'autres. Cette hypothèse s'avère réaliste dans le domaine des systèmes embarqués par exemple.

Les résultats de ce chapitre sont publiés dans [FR03], [Rol03] et [RF03].

Chapitre 5

Une introduction au test de robustesse de systèmes à base de composants

Dans un système à base de composants (SBC), les composants doivent interagir entre eux et avec des éléments externes ([Zal01]). Ces composants peuvent s'exécuter sur des systèmes simples, multiples, distribués ou en réseau, de manière séquentielle ou bien concurrente, et doivent pouvoir communiquer avec n'importe quel autre composant. Malgré tous les avantages procurés par les SBC, certains aspects temps-réel ne peuvent pas à l'heure actuelle être encapsulés dans le composant, comme la synchronisation ou l'optimisation de mémoire.

Pour pouvoir modéliser un système temps-réel à base de composants (STRBC), il faut d'abord définir ce qu'on appelle un composant, qui est en fait la brique de ce genre de système. Szyperski ([Szy98]) définit un composant ainsi :

Définition 5.0.1 (Composant) *Un composant est une unité de composition ayant des interfaces spécifiées à l'aide d'un contrat et ayant des dépendances contextuelles totalement explicites, qui peut être déployé indépendamment et est sujet à une composition avec un tiers.*

Ensuite, lorsqu'on souhaite construire un système à base de composants, [BCC03] fait une distinction entre différents aspects du composant :

- l'*implantation* du composant : la réalisation exécutable du composant, obéissant aux règles dictées par son modèle ;
- l'*interface* du composant : résume les propriétés du composant qui sont visibles à l'extérieur pour les autres parties du système, et qui peuvent être utilisées pour élaborer le système ;
- Une technologie de composant pour systèmes temps-réel devrait permettre la spécification et la prédiction du temps et des propriétés de qualité de service.

Quelques travaux ont déjà été effectués pour contribuer à la modélisation et à la conception de systèmes à base de composants, et notamment les apsects qualité de service. En ce qui concerne la formalisation, on peut citer les outil WRIGHT ([AG97]) et RAPIDE ([DCL95]), des langages de description d'architectures qui permettent de décrire les interactions entre composants. En ce qui concerne les composants embarqués, KOALA

([vO02], [vOvdLKM00]) est un modèle de composants et une architecture permettant de travailler spécialement sur la conception de composants électroniques, utilisé notamment par Philips. Dans ce modèle, un composant interagit avec son environnement uniquement par des interfaces explicites : des interfaces dites "providing" et des interfaces dites "requirement". Les dépendances au niveau hardware sont encapsulées dans des composants particuliers. En ce qui concerne les composants temps-réel, on peut citer aussi le projet TURTLE ([ACLdSS04]), avec une syntaxe UML qui étend les diagrammes de classes et les diagrammes d'activité par des opérateurs de composition et des opérateurs temporels. Sa sémantique formelle est exprimée en RT-LOTOS. En fait, TURTLE fournit un outil de conception de système temps-réel, qui offre en plus des possibilités de validation. Il est nécessaire aussi de parler de FRACTAL ([BCS02]), qui a permis d'améliorer la composition et le dynamisme des modèles de composants, en incluant en particulier la hiérarchie, et une nouvelle approche pour la composition. D'autre part, [FSLM02] propose un framework appelé THINK, dont le but est d'implémenter des noyaux de systèmes d'exploitation à partir de composants de taille arbitraire. Enfin, un important travail sur la conception, la fiabilité et la validation de systèmes de composants a été réalisé à l'IRISA, au cours du projet TRISKELL ([Jéz03]).

Après avoir rappelé les divers besoins d'un STRBC, ce chapitre va proposer une piste et une architecture pour en tester la robustesse.

5.1 Besoins au niveau des composants temps-réels

Les STRBC sont progressivement devenus à la mode dans le développement de systèmes. En effet, la possibilité de réutiliser des composants certifiés offre un avantage économique certain. Toutefois, il faut reconnaître qu'au niveau temps-réel, il reste un bon nombre d'améliorations à apporter. [BCC03] propose une étude complète sur les composants temps-réel. Nous allons ici résumer les besoins principaux.

Un modèle de composant normalisé

Chaque organisme a développé son propre modèle de composant ce qui entraîne inévitablement une incompatibilité entre différents modèles (pas d'interopérabilité). L'idéal serait donc d'obtenir un modèle normalisé pour tous, ce qui est bien sûr improbable. La solution pourrait passer par la normalisation des interfaces des composants ou par la conception de "convertisseurs" permettant de mettre en relation les différentes interfaces. Dans tous les cas, il faudrait un langage de haut niveau avec une sémantique formelle pour décrire les composants, la possibilité de "lier" le composant au framework, et enfin un effort de standardisation.

D'autre part, une description précise, normalisée et sémantique du comportement du composant et de son interaction avec les autres composants est indispensable pour espérer une validation ou un environnement de développement utilisable.

Des implantations plus légères

Les technologies basées sur les composants existantes nécessitent à l'heure actuelle trop de ressources pour être appliquées aux systèmes embarqués. Il est donc nécessaire de les alléger et de les optimiser (accès mémoire, ordonnancement, ...). On retrouve ce problème dans de nombreux domaines, et notamment dans l'automobile. Il faudrait donc développer des composants sur ressources limitées ou optimiser les systèmes d'exploitation temps-réel.

Incorporation dans un système de composants autonomes et vérifiés

Il est nécessaire d'utiliser des techniques de validation et d'évaluation de performances sur le composant avant de l'utiliser. Cela permet d'obtenir un composant certifié. Par extension, il faudrait mettre au point des techniques de conception ou de développement éprouvées. Dans ce chapitre, nous allons nous intéresser principalement à la validation d'un composant, puis de son incorporation.

Obtenir des propriétés "extra-fonctionnelles" sur le composant

Un modèle de composant satisfaisant devrait permettre de prendre en compte les paramètres matériels (ou bas niveau) au moment de la conception haut niveau pour garantir des propriétés cruciales telles que la fiabilité et la sûreté. C'est aspect devient indispensable lorsqu'on parle de prédiction au niveau de l'analyse temporelle et de qualité de service. Ces propriétés concernent entre autres le temps d'exécution, les *deadlines*, les synchronisations et l'utilisation des ressources. Ces propriétés peuvent s'avérer utiles pour toute vérification ou évaluation de performance. D'autre part, il faut leur trouver une forme de représentation, qui pourrait être sous forme d'équation ou de contrat.

Indépendance du composant par rapport à la plate-forme d'éxecution

Actuellement, un composant ne fonctionne que sur sa plate-forme associée. Pour résoudre ce problème, l'idéal serait qu'une seule plate-forme émerge, ce qui est peu probable. Sinon, il faudrait mettre au point des "convertisseurs" pour transformer un composant d'une plate-forme vers une autre, ou alors développer des composants "multi-plate-forme", ie prévus pour différentes plate-formes. Une solution satisfaisante consisterait à pouvoir développer le composant à haut niveau, de façon indépendante de la plate-forme, puis de générer du code spécifique.

Propriétés de prévision du système

Il faudrait mettre au point des techniques pour prédire le comportement et la complexité du système global à partir de ses composants. Ainsi, cela permettrait de choisir des ensembles de composants, avec une certaine architecture, et de prévoir leur comportement. Toutefois, les systèmes actuels sont trop complexes pour espérer ce genre de prédictions.

Propriétés de temps

Il faudrait être capable au moment de la conception d'analyser les propriétés de temps d'un composant et d'en déduire des renseignements comme le temps d'exécution (maximum, statistique, ...), les deadlines, ..., le tout sans lien avec la plate-forme correspondante. Pour cela, il est indispensable que le composant intègre des propriétés matérielles.

Certification de composants

Il est nécessaire d'avoir une description précise et certifiée des propriétés intrinsèques du composant pour pouvoir réaliser des échanges de composants. Cela nécessite des techniques de développement fiables et validées, des composants vérifiés et testés, et une documentation complète. Ce genre de certification peut induire des problèmes de propriétés, au sens commercial du terme.

Adaptabilité du composant

Il serait très utile que le composant puisse s'adapter à son contexte d'exécution, son environnement (par exemple les ressources), ce qui entraînerait une plus grande souplesse d'utilisation et de réutilisation. Dans ce cas, il est toutefois difficile de prévoir les différents modes de fonctionnement. De même, il n'est pas évident de décider quel niveau d'adaptabilité on souhaite, de définir un comportement "minimum", et en même temps de certifier les propriétés temporelles ou de sûreté. Cette problématique se rapproche fortement de celle de la robustesse.

Non interférence entre composants

Dans le cadre de système critiques, il est nécessaire de mettre en place des mécanismes qui gèrent de façon fiable le partage de ressources et les accès mémoire, afin de protéger des données. Il peut s'agir de séparer les données des différents composants, ou éventuellement de mettre en relation certaines données de façon contrôlée. Pour cela, un partage avancé des ressources est donc nécessaire.

Outils de support

Un STRBC nécessite un environnement de développement complet, comprenant au moins les éléments suivants :

- outils de conception (design) ;
- génération de code ;
- compilation / optimisation ;
- analyses diverses ;
- validation (vérification, test) ;
- évaluation de performances, statistiques ;
- documentation ;
- description de la plate-forme.

Modèle de composition

Pour qu'un STRBC puisse avoir un sens, il faut bien définir les interactions entre composants. Cela signifie qu'il faut :

- définir les techniques de communication et synchronisation ;
- donner un modèle standard de communication ;
- permettre la composition de propriétés (par exemple propriétés comportementales) pour garantir des propriétés globales du système.

5.2 Architecture et exécution sans prise en compte des délais

5.2.1 Choix du modèle

Dans l'éventail disponible des modèles, nous avons choisi de spécifier un composant à l'aide d'un TIOA. En effet, ce modèle permet de gérer les communications entre composants grâce aux entrées/sorties, et permet de gérer les aspects temporels, grâce aux horloges. On notera toutefois de grosses lacunes dans ce modèle au niveau des propriétés "extra-fonctionnelles", notamment au niveau de la gestion des ressources, et plus généralement au niveau matériel.

Ainsi, un STRBC se présente comme une collection de composants où chacun d'eux est décrit par une spécification nominale et une dégradée, modélisées sous forme de TIOA. A cause de cette modélisation et par souci de simplicité, nous ne pouvons avoir qu'un seul service par composant.

5.2.2 Représentation du testeur local

L'architecture locale du testeur est similaire à celle utilisée pour le test de robustesse précédemment. La figure FIG 5.1 donne cette architecture. Elle se compose d'un exécuteur de test, qui a pour rôle d'appliquer les entrées au composant et d'un moniteur de test qui est chargé d'enregistrer toutes les traces d'exécution du composant. Pour que cet enregistrement soit complet, l'exécuteur de test envoie aussi les entrées au moniteur. Par ailleurs, le testeur contient une file d'attente pour stocker les entrées venant des autres composants, ainsi que ses propres horloges, synchronisées avec les horloges du composant.

Les testeurs communiquent entre eux à l'aide de files d'attente selon le fonctionnement suivant : une transition de sortie venant d'un testeur T_i, envoyée comme une entrée vers le testeur T_j va attendre sur la file $ioQj$ jusqu'à ce que la transition soit autorisée dans T_j. D'autre part, si lors de l'exécution d'une séquence, le testeur T_i arrive sur une transition nécessitant une entrée venant d'un autre composant T_j, alors T_i va interrompre l'application de la séquence sur P_i. Celle-ci reprendra lorsque l'entrée correspondante arrivera dans la file $ioQi$. Bien sûr, si l'entrée correspondante se trouve déjà dans $ioQi$, il n'y a pas d'interruption dans la séquence.

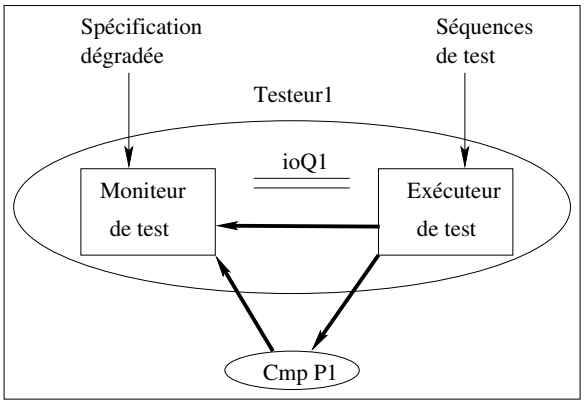


FIG. 5.1 – Détail d'un testeur local

5.2.3 Schéma des communications entre composants

Dans notre approche, nous allons considérer un STRBC comme un ensemble de composants qui peuvent tous interagir avec l'environnement. Ce travail s'inspire de [NH84]. Pour pouvoir tester ce système, nous proposons de dédier pour chaque composant P_m un testeur T_m. Toutes les interactions entre composants doivent passer par les testeurs correspondants. Tous les testeurs peuvent interagir entre eux. Une file d'attente est prévue sur chaque testeur pour stocker les entrées venant des autres testeurs. La figure 5.2 illustre l'architecture de test.

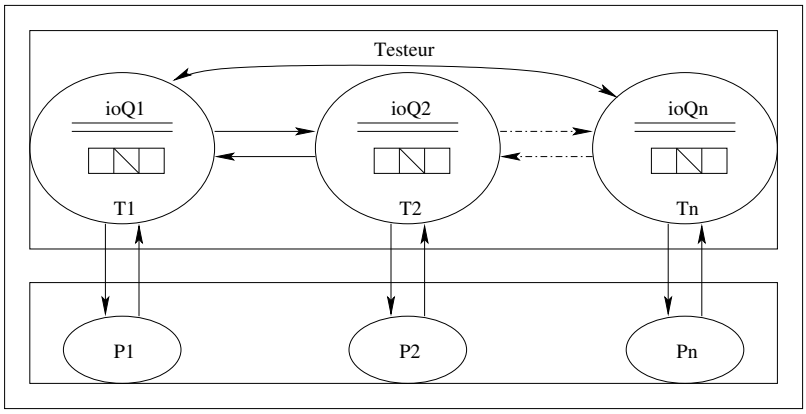


FIG. 5.2 – Architecture de test pour le système complet

Ainsi, une sortie venant du composant P_i à l'intention du composant P_j va en fait être envoyée à T_i qui va à son tour l'envoyer à T_j, qui finalement va l'appliquer à P_j. Dans cette partie, il faut bien noter que nous ignorons le temps de communication provoqué par les testeurs entre les composants.

Le but de notre travail va être à la fois de vérifier la robustesse des composants du système, mais aussi d'identifier des situations qui mènent le système dans un état dit de

deadlock (blocage), ie que deux composants attendent mutuellement (indéfiniment) une action venant de l'autre.

Il est possible dans un premier temps de soumettre le système global à des radiations (aléas internes) pour tester sa robustesse. Dans ce cas, les séquences de test générées ne subissent pas de mutation. Dans un deuxième temps, ces séquences sont mutées pour pouvoir tester la robustesse face aux aléas externes. Comme précédemment, c'est la spécification dégradée de chaque composant qui nous permettra d'évaluer si le système s'est comporté de façon robuste.

Notations

Dans la suite de cette section, nous allons utiliser les notations suivantes :
- T_m, P_m : T_m est le testeur du composant P_m ;
- N correspond au nombre total de composants et donc de testeurs ;
- $O_m^{T_i}$ est une sortie partant du composant P_m, et envoyée vers le testeur T_i (le testeur du composant P_i) en passant par T_m ;
- Ts_m est l'ensemble des séquences (mutantes ou non, selon le choix sur les aléas : internes ou externes) générées pour tester le composant P_m ;
- ITs_m est un sous-ensemble de Ts_m ne contenant que les séquences ayant au moins une sortie vers un autre composant P_i ; on l'appelle l'ensemble des *séquences de communication* du composant P_m ;
- S^{Pm} est la spécification nominale du composant P_m ;
- S_d^{Pm} est la spécification dégradée du composant P_m.

5.2.4 Exécution des tests

Le verdict de l'application d'une séquence de test sera considéré comme positif si et seulement si l'exécution des actions vitales respecte les contraintes exprimées par les spécifications dégradées des composants, ie respectent la relation $accept_{\mathcal{R}}$.

L'exécution des test se fait en deux phases. La première teste la robustesse de chaque composant de façon indépendante, et la seconde teste la robustesse des communications entre composants.

Dans les deux phases, chaque testeur T_m applique les séquences de test (éventuellement mutantes) grâce à son exécuteur de test, et enregistre toutes les traces grâce au moniteur de test. Ensuite, à partir de ces traces, de la spécification dégradée de chaque composant, et grâce à la relation $accept_{\mathcal{R}}$, la robustesse peut être évaluée.

Lors de la première phase, toutes les communications entre les composants sont ignorées, et par conséquent, chaque testeur envoie à son composant correspondant les entrées nécessaires, qu'il s'agisse d'une entrée venant d'un autre composant ou venant de l'environnement. En fait, on teste chaque composant indépendamment du système global. A la fin de cette étape, un premier verdict de robustesse est envisageable pour chaque composant. Lors de cette phase, toutes les séquences qui mettent en évidence une interaction entre deux composants (c'est a priori le cas d'une grande majorité des séquences) sont ajoutées aux ensembles ITs_m en prévision de la deuxième phase.

Dans la deuxième phase, tous les testeurs exécutent les séquences de leur ensemble ITs_m en parallèle, et cette fois, les communications entre composants sont aussi testées grâce aux files d'attentes entre testeurs. De façon plus détaillée, chaque testeur T_m exécute une des séquences de ITs_m en simultané avec tous les autres testeurs, de telle sorte que toutes les combinaisons possibles de séquences soient exécutées. Il faut admettre toutefois que ce principe donne des séquences de test qui s'avèrent très longues à exécuter (exponentiel par rapport au nombre de séquences pour chaque composant). A la fin, ce sont encore les traces d'exécution de chaque composant, les spécifications dégradées et la relation $accept_{\mathcal{R}}$ qui permettent d'évaluer la robustesse du système. En même temps, parmi toutes ces séquences, on repère les combinaisons qui ont mené à une situation de deadlock.

La méthode complète de ce test est décrite figure FIG. 5.3, et est donnée en détail dans l'algorithme ci-dessous.

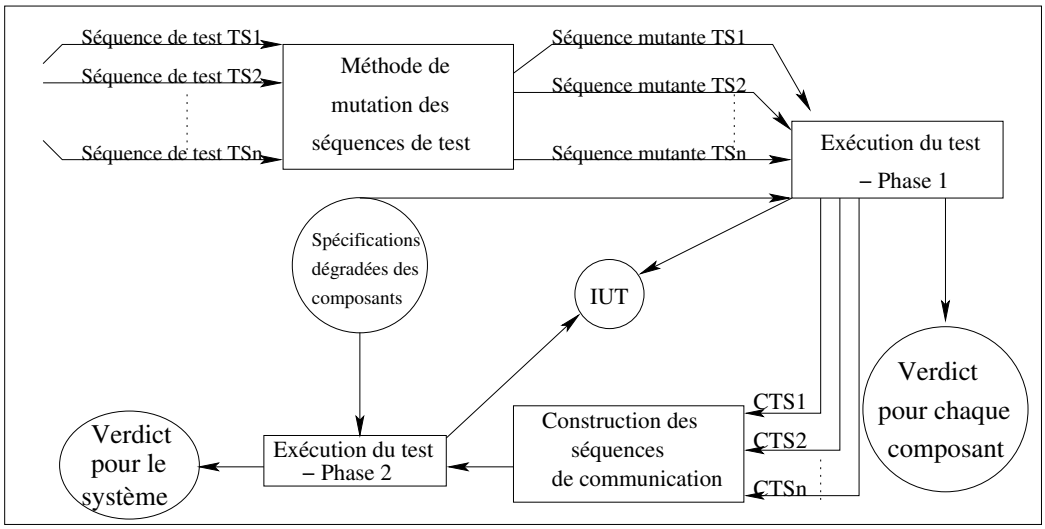


FIG. 5.3 – Schéma global de la méthode de test

5.2.5 Algorithme détaillé pour l'exécution du test

```
1    Algorithme: exécution de test phase 1
2
3    Données: les spécifications nominales SP_1 ... SP_N
4             les spécifications dégradées SdP_1 ... SdP_N
5             les ensembles de séquences de test TS_1 ... TS_N
6             les ensembles de séquences de communication ITS_1 ... ITS_N
7             (initialement vides)
8    Résultats: verdict: Robuste ou non
9
10   Début
11      Pour tous les composants P_m faire
12         Pour toutes les séquences de test TS_m_j de TS_m faire
```

```
13          En respectant les contraintes de temps, appliquer chaque action
14          d'entrée de TS_m_j via testeur T_m sur P_m ;
15          Enregistrer chaque action (avec son timing) dans les traces de P_m;
16          Vérifier la relation d'acceptation par rapport à la spécification
17          dégradée SdP_m;
18          Si une sortie de TS_m_j a pour destination un autre composant P_i
19             Ajouter TS_m_j à l'ensemble des séquences de communication
20             ITS_m;
21          FinSi
22       FinPour
23    FinPour
24    Donner un verdict (Robuste ou non);
25 Fin
```

```
1  Algorithme: exécution de test phase 2
2
3  NB: Cet algorithme s'exécute en parallèle sur tous les testeurs
4      On suppose ici que P_m est le composant, et T_m le testeur
5
6  Données: les spécifications nominales SP_1 ... SP_N
7           les spécifications dégradées SdP_1 ... SdP_N
8           les ensembles de séquences de communication ITS_1 ... ITS_N
9           les files de chaque testeur: ioQ_1 ... ioQ_N
10 Résultats: verdict: Robuste ou non
11
12 Début
13    Pour toutes les séquences de test ITS_m_j de ITS_m faire
14       Tant qu'il reste des transitions non traitées dans ITS_m_j faire
15          Tr <- première transition d'entrée de ITS_m_j non traitée;
16          Si Tr est une transition locale
17             // entrée ne venant pas d'un autre testeur
18             En respectant les contraintes de temps, T_m applique l'action
19             de Tr sur P_m;
20             Enregistrer l'action dans les traces de P_m;
21             Tant que la transition suivante de Tr est une sortie faire
22                Tr <- transition suivante de Tr;
23                Si l'action de Tr est destinée à un composant P_i
24                   T_m envoie l'action de Tr dans la file ioQ_i du testeur T_i;
25                   Si (T_i attendait cette entrée)
26                      T_m envoie un signal de reprise à T_i: signal(T_i);
27                   FinSi
28                FinSi
29             FinTantque
30          Sinon // entrée venant d'un autre testeur T_j
31             Si l'action de Tr se trouve dans la file ioQ_m
32                En respectant les  contraintes de temps, T_m applique l'action
33                de Tr sur P_m;
34                Enregistrer l'action dans les traces de P_m;
35                Tant que la transition suivante de Tr est une sortie faire
```

```
36                    Tr <- transition suivante de Tr;
37                    Si l'action de Tr est destinée à un composant P_i
38                       T_m envoie l'action de Tr dans la file ioQ_i du testeur T_i;
39                       Si (T_i attendait cette entrée)
40                          T_m envoie un signal de reprise à T_i: signal(T_i);
41                       FinSi
42                    FinSi
43                 FinTantque
44              Sinon // l'action de Tr pas dans ioQ_m
45                 Si T_j n'attend pas une entrée de ce testeur // pas de deadlock
46                    Suspendre l'exécution de T_m jusqu'à la bonne entrée venant
47                    de T_j: wait(T_m) ;
48                 Sinon // cas de deadlock
49                    T_m applique l'entrée appropriée à P_m, d'après la
50                    spécification nominale SP_m;
51                    Identifier cette combinaison de séquences comme deadlock;
52                 FinSi
53                 Enregistrer l'action dans les traces de P_m;
54              FinSi
55           FinSi
56        FinTantque
57     Vérifier la relation d'acceptation par rapport à la spécification
58     dégradée SdP_m;
59     FinPour
60     Donner un verdict (Robuste ou non);
61  Fin
```

Explications

La première phase teste localement chaque composant à l'aide de ses spécifications, et sert aussi pour identifier les séquences de communication. La technique est similaire au chapitre 4.

Dans la deuxième phase, chaque testeur T_m sélectionne une de ses séquences de ITs_m, pour l'exécuter pendant que les autres testeurs font pareil avec d'autres séquences. Au final, toutes les combinaisons possibles d'exécution de séquences doivent avoir été testées. En supposant qu'il y ait k séquences en moyenne par composant, et N composants, cela donne donc de l'ordre k^N cas de test.

Pour chaque transition de sortie de la séquence ITs_m, le testeur T_m est responsable de l'envoi de l'action au testeur correspondant, et éventuellement de lui ordonner de reprendre son exécution. S'il s'agit d'une entrée, le testeur identifie si elle vient de l'environnement ou d'un autre testeur. Dans le dernier cas, le testeur T_m n'applique l'action correspondante que si un autre composant a déjà envoyé cette action à P_m, ie si cette action se trouve dans la file ioQ_m.

Si un testeur T_i attend une entrée de T_j, celui-ci suspend l'exécution du test jusqu'à l'arrivée de l'action correspondante. En revanche, si un testeur T_i attend une entrée de T_j et réciproquement, alors un cas de blocage a été identifié.

5.3 Prise en compte des délais

Pour l'instant, nous avons décidé de négliger le temps de traitement provoqué par le passage des communications à travers les testeurs. Toutefois, cette hypothèse peut s'avérer peu réaliste dans certains cas, en particulier si les communications entre testeurs et composants sont quasiment immédiates. Dans ce cas, le traitement des données dans un testeur va prendre un temps non négligeable qu'il faudrait considérer.

Nous proposons ici de prendre en compte ces délais pour évaluer les instants des transitions suivantes, et ainsi obtenir une exécution de test plus réaliste. Ces délais peuvent être identifiés dans trois situation :

1. délai entre l'instant de réception par le testeur T_m d'une entrée venant de P_m, et l'instant d'émission de la réponse de T_m vers P_m ;
2. délai entre l'instant de réception par le testeur T_m d'une entrée venant de P_m, et l'envoi d'une réponse partant de T_m à l'intention du testeur T_i ;
3. délai entre l'instant de réception par le testeur T_m d'une entrée venant de T_i, et l'envoi d'une réponse partant de T_m à l'intention du composant P_m.

Les séquences de test sont de la forme : $Ts_m = Tr_1^m.Tr_2^m...Tr_n^m$ qui s'exécute sur le composant P_m. On note t_k^m l'instant où la k-ème transition s'exécute sur le composant P_m. On note aussi δ_k^m la condition d'horloge de la k-ème transition. Ct est le temps courant.

Pour illustrer notre façon de traiter ces délais, nous utiliserons les séquences de test suivante :

- $Ts_1 = (!b, -, c_1 > 200), (?a, c_1 := 0, c_1 < 240), (!e, c_1 := 0, c_1 < 200)...$, et
- $Ts_2 = ...(?b, c_2 := 0, c_2 < 270), (!d, c_1 := 0, c_1 < 300)$.

ainsi que la figure FIG. 5.4.

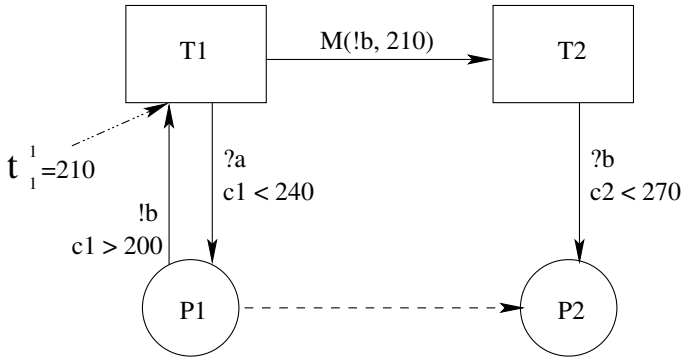


FIG. 5.4 – Prise en compte des délais de communication

Respectivement selon les trois cas précédents, le testeur va gérer les trois situations de façon bien distincte.

1. Dans ce cas, le testeur T_m enregistre l'instant t_k^m de réception de l'entrée Tr_k^m venant du composant P_m. Pour compenser le délai, T_m doit exécuter la $(k+1)$-ème transition, Tr_{k+1}^m, avec la contrainte d'horloge $\delta_{k+1}^m + Ct - t_k^m$.

Par exemple, si on considère Ts_1 et la figure FIG. 5.4, en supposant que $!b$ vient de P_1 et est reçu par le testeur T_1 à l'instant $t_1^1 = 210$, si T_1 est prêt à envoyer $?a$ à l'instant $Ct = 260$ par exemple, alors la condition d'horloge doit satisfaire : $240 + 260 - 210$.

2. Dans cette situation, le testeur T_m enregistre l'instant t_k^m de réception de l'entrée Tr_k^m venant du composant P_m, puis l'envoie au testeur T_i, en ajoutant au message t_k^m, ce qui signifie que le message entre les deux testeurs sera de la forme : $M(Tr_k^m, t_k^m)$.
   Par exemple, toujours en considérant Ts_1 et la figure FIG. 5.4, si $!b$, venant de P_1, est reçu par T_1 à l'instant $t_1^1 = 210$, T_1 envoie le message $M(!b, 210)$ au testeur T_2.
3. Dans cette dernière situation, T_m reçoit une entrée de la forme $M(Tr_k^i, t_k^i)$ venant du testeur T_i. Ainsi, lorsque T_m est prêt à exécuter la $(k+1)$-ème transition Tr_{k+1}^m, il l'exécute avec la condition d'horloge $\delta_{k+1}^m + Ct - t_k^i$.
   Par exemple, en considérant T_2 et la figure FIG. 5.4, avec $M(!b, 210)$ venait de T_1, et en supposant que T_2 est prêt à envoyer $?b$ à l'instant $Ct = 280$, alors la condition d'horloge sera : $270 + 280 - 210$.

L'algorithme général de l'exécution des tests, donné précédemment, ne change quasiment pas. Il suffit de lui ajouter la prise en compte des délais lors de chaque communication de type composant/testeur ou testeur/testeur.

5.4 Conclusion

Dans ce chapitre, nous nous sommes intéressés en particulier aux aspects temps-réel et certification d'un composant. Grâce au modèle des automates temporisés, les contraintes de temps peuvent être gérées. D'autre part, l'architecture de test et l'algorithme présentés dans ce chapitre, permettent de vérifier la robustesse du composant. Si on ajoute cela au test de conformité, on peut considérer que nous avons les bases d'une certification.

Par ailleurs, on peut directement fournir les séquences test avec le composant. Cela évite au concepteur du système global de les générer à chaque fois. Ainsi, il ne lui reste plus qu'à les appliquer sur chaque composant, en parallèle, afin de valider la totalité du système. Ces séquences, et éventuellement les aléas, peuvent être considérés comme propriétés extra-fonctionnelles nécessaires pour la qualité de service.

Les résultats de ce chapitre sont publiés dans [TRF05] et [FRT05].

Chapitre 6

Un outil de génération de séquences pour la conformité et la robustesse

Dans ce chapitre, nous allons présenter un outil de génération de séquences de test que nous avons développé. Cet outil permet de partir de différents modèles (RT-LOTOS, IF, UPPAAL ou KRONOS) pour générer des séquences.

Cette génération est prévue à la base pour le test de conformité. L'idée est de réussir à identifier les états de la spécification dans le système final. Pour cela, l'outil se base sur un algorithme proche des méthodes UIO et Wp appliquées habituellement sur des FSM. Par ailleurs, pour faciliter l'utilisation, une interface graphique développée en Java a été ajoutée.

Afin de montrer l'efficacité de la méthode pour identifier des états, nous présentons des statistiques obtenues à partir d'automates générés aléatoirement.

Ensuite, les séquences sont réutilisées pour pouvoir servir dans le cadre du test de robustesse.

Le schéma général de l'outil est donné figure FIG. 6.1.

6.1 Différents formats d'automates et différents outils

Le modèle de base que nous avons choisi pour notre méthode de test est le TIOA, que ce soit pour la conformité ou pour la robustesse. Mais dans le cadre d'une application logicielle, il faut choisir un format d'automate interprétable par celle-ci, ou encore créer le notre. Pour des raisons de compatibilité avec les outils développés par les autres laboratoires, nous avons choisi KRONOS comme format de base.

Toutefois, il existe plusieurs formats d'automates utilisés dans différentes applications, c'est pourquoi nous nous sommes attachés dans un premier temps à convertir, moyennant éventuellement certaines pertes d'informations, différents formats (IF, UPPAAL et RT-LOTOS) vers le format *KRONOS*.

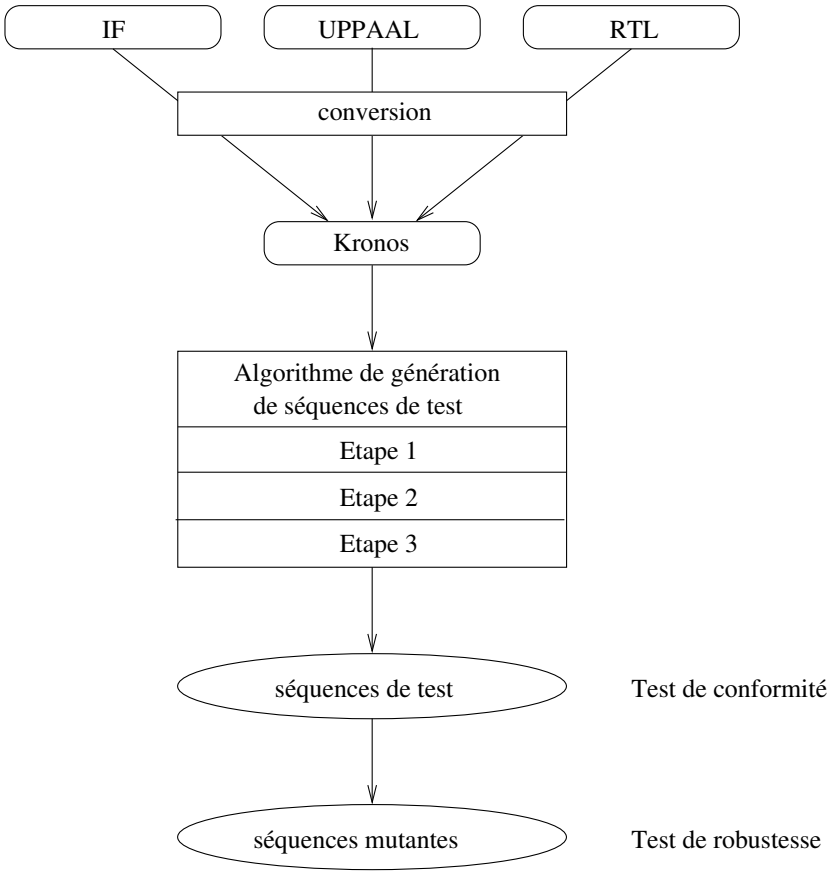


FIG. 6.1 – Schéma global de l'outil de génération de séquences

6.1.1 *KRONOS*

Les outils *KRONOS*

KRONOS [Yov97] est une série d'outils permettant de travailler sur les automates temporisés. Son objectif principal est de fournir un moteur de vérification afin d'être intégré dans des environnements pour des systèmes temps-réel. Développé par VERIMAG à l'Université de Grenoble, *KRONOS* ne permet pas de travailler sur des automates temporisés d'entrée/sortie, ce qui va donc impliquer quelques modifications au niveau du format.

Le format *KRONOS*

Ce format d'enregistrement des automates temporisés, utilisé en entrée du programme de génération de séquences de test, est décrit ci-dessous.

- L'en-tête du fichier est composée du nombre d'états qui compose l'automate ainsi que du nombre de transitions et d'horloges comme suit :
  #**states** *s*
  où *s* est le nombre d'états qui composent l'automate.

#trans *t*
avec t le nombre de transitions.
#clock *nb C1 C2 ...*
nb étant le nombre d'horloges et *C1 C2...* le nom de chacune d'elle.

– L'automate lui-même est défini dans le corps du fichier par une suite d'expressions de forme :
#state :*n*
défini le début de l'état n.
#prop :*nom*
désigne le label de l'état.
#invar :*invariant*
associe une ou plusieurs conditions d'invariant sur les horloges à l'état.
#trans :*$trans_1 \ldots trans_n$
associe une liste de transitions à l'état avec chaque transition de la forme :
guard`=>`*label*`; reset{`$clock_1 \ldots clock_j$`}; goto` k où *guard* est la condition temporelle de la transition, *label* est l'étiquette de la transition, *reset* contient la liste des horloges à remettre à zéro et k l'état cible.

Le format des automates temporisés qui sera utilisé lors du programme de génération de séquences de test est le format *KRONOS*. Or, d'autres applications travaillent également sur ce type d'automates mais utilisent d'autres formats. Il va donc falloir convertir ces différents formats vers *KRONOS* pour pouvoir tester les automates créés.

6.1.2 Un outil de modélisation et vérification des automates temporisés : *UPPAAL*

Description de *UPPAAL* et de son format (*XML*)

UPPAAL [LPY97] est un outil de modélisation, validation et vérification de systèmes temps-réel, ceux-ci étant modélisés comme des réseaux d'automates temporisés. Le format des fichiers utilisés par *UPPAAL* est le *XML*. Deux autres formats sont disponibles pour *UPPAAL*, le format *TA* et une extension de celui-ci, *XTA*, mais ne sont plus gérés dans les versions récentes.

Chaque automate du système représente un processus de ce système et communique avec les autres processus à travers des tubes de communication (les *channels*). Chaque automate est également régi par des horloges, celles-ci pouvant être locales (n'affectant que l'automate en question) ou globales (affectant tout le système). Tous les automates du système doivent être déterministes.

Passage au format *KRONOS*

L'outil que nous avons implémenté pour réaliser cette conversion est appelée *xml2tg*. C'est un analyseur lexical réalisé à l'aide de *Lex* et *Yacc*. Il prend en entrée un fichier *UPPAAL* et écrit autant de fichiers *KRONOS* (d'extension *TG*) qu'il y a d'automates (de processus dans le système). Il écrit en plus un fichier de description des automates

(d'extension *aut*) qui contient la liste de tous les automates qui composent le système ainsi que tous les attributs de ce système (canaux de communications, horloges globales au système et les instantiations des automates pour lancer le système). De cette manière, on pourra donc par la suite avoir la possibilité de tester chaque automate séparément et le fichier global généré permettra de tester le système complet.

6.1.3 Un environnement de validation des automates temporisés : IF

Description de *IF* et de son format (*IF*)

IF [BFG+99a] est un environnement libre de validation pour les systèmes d'applications distribuées utilisé notamment par l'IMAG (Grenoble). Il est construit à partir d'un langage intermédiaire et comprend de multiples outils pour la validation. On s'intéressera ici au langage issu de SDL (*Specification and Description Language*).

Les fichiers *IF* comprennent la description d'un système constitué de différents processus communiquant entre eux par l'intermédiaire de tampons de communication qui se font soit par des échanges de signaux, soit par l'intermédiaire de variables partagées. Les processus peuvent être non déterministes et plus d'une transition peut être possible : ainsi, durant l'exécution, toutes les situations doivent être envisagées. Le langage inclut aussi la création et la destruction dynamiques de processus.

Passage au format *KRONOS*

Le programme écrit est aussi un analyseur *Lex* et *Yacc* appelé *if2tg* permettant de réaliser la conversion des fichiers d'extension *IF*. Il s'appuie sur la syntaxe trouvée dans la documentation trouvée sur le site de *IF*. Comme pour *xml2tg*, un fichier par automate sera créé et un fichier d'extension *aut* sera également généré et contiendra les noms des automates ainsi que la définition du système.

Chaque fichier *IF* correspond à la description d'un système complet, constitué de différents `process`. Chaque `process` est assimilé à un automate temporisé avec ses états et ses transitions. Le langage *IF* étant beaucoup plus riche que le format *KRONOS* de nombreuses options sont ignorées lors de la conversion. Ainsi, toutes les informations concernant les priorités et l'atomicité des actions ne sont pas retranscrites.

Toutes les actions sont définies dans les transitions. Tout d'abord, on trouve les gardes, qui ne sont pas forcément fonction du temps, un champ `input` et différentes actions dont certaines seulement seront transcrites. Les actions d'entrée/sortie seront les actions relatives aux signaux (qui ont la même fonction que les `channels` pour *UPPAAL*). Les précisions concernant la route que devront emprunter les signaux seront ignorées.

En outre, plusieurs actions d'entrée (le champ `input`) et sortie (les actions `output`) peuvent se côtoyer dans une seule transition. Il est donc nécessaire d'effectuer une conversion. Pour cela, on va créer un nouvel état pour chaque action d'entrée/sortie de telle sorte que l'on ait plus qu'une seule action de ce type sur chaque transition. Bien entendu,

puisque toutes les actions sont ordonnées, on est obligé de garder cet ordre et les actions autres que entrée/sortie seront réparties entre les nouveaux états et les nouvelles transitions. Les nouveaux états que l'on crée de cette manière ne possèdent pas de garde.

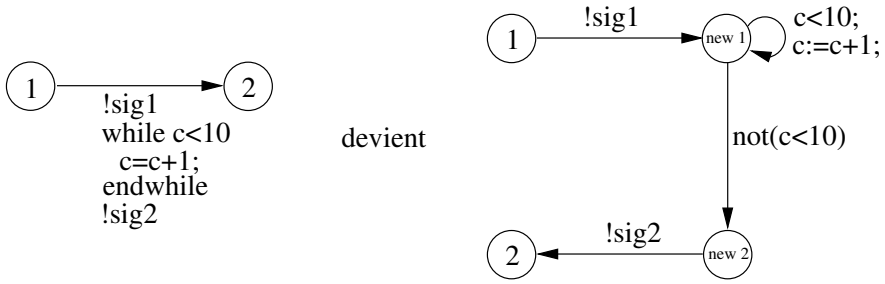


FIG. 6.2 – Transformation du while

L'autre problème, toujours au niveau des transitions, vient du fait que l'on peut trouver des boucles `while` et des `if/then/else`. Là encore, il faut créer de nouveaux états suivant le schéma de la figure 6.2.

Pour les opérations sur les horloges, on trouve trois actions différentes : les actions `task` (pour les assignements de variables), les actions `set` (pour les assignements d'horloges) et les actions `reset` (pour la remise à zéro des horloges et des variables). Les modifications sur les variables ne sont pas prises en compte.

6.1.4 Un outil de modélisation des automates temporisés : RTL

Description de *RTL* et de son format (*LOTOS*)

RTL [CO95], [CSLO00]) est une application développée par le LAAS permettant de modéliser des automates temporisés en se servant du langage *RT-LOTOS*. Contrairement aux formats *UPPAAL* et *IF*, cet outil manipule en général des automates simples. Le langage *RT-LOTOS*[1], reconnu par l'outil *rtl*, est une extension du langage *LOTOS*. Une spécification *RT-LOTOS* est souvent exploitée à des fins de validation (simulation/vérification), ou encore à des fins d'ordonnancement.

L'outil *rtl* possède de nombreuses fonctionnalités. Celle qui nous intéresse est la possibilité de compiler une spécification donnée au format *RT-LOTOS* (d'extension *.lot*) en un automate temporisé de type *Dynamic Timed Atomaton (DTA)* (et d'extension *.dta*). Un automate temporisé de type *DTA* est en fait un *4-uplet* (S, N_{Clocks}, E, s_0) construit comme suit :

- **S** un ensemble fini d'états
- N_{Clocks} une fonction associant un indexe aux contraintes sur les horloges
- **E** un ensemble fini de transitions où chaque transition est de la forme (s, a, K, C, θ, s') où :
  - s est l'état de départ de la transition

[1]Real-Time LOTOS

  - s' est l'état d'arrivée de la transition
  - K est la condition temporelle
  - a est le label d'une action associée
  - C $\subseteq \{1 \ldots N_{Clocks}(s')\}$ définit les indexes des horloges devant être remises à zéro.
  - $\theta \in N^{+}_{Clocks}$ désigne la ou les fonctions associées à la transition
- s_0 est l'état initial

Passage au format *KRONOS*

Le programme utilisé pour permettre la conversion du format de sortie de *RTL* (fichiers d'extension *DTA*) en un fichier compréhensible par le programme de génération de séquence de test est appelé *dta2kronos*. Ce programme est le résultat de la modification d'un programme du même nom téléchargé à partir du site officiel de *RTL* ([RTL]). Des modifications lui ont été apportées dans le but d'assurer la compatibilité avec nos travaux précédents.

6.1.5 La librairie PolyLib

La PolyLib ([Loe99]) est une librairie logicielle qui fournit des solutions pour la manipulation des polyhèdres. Elle permet d'effectuer un grand nombre d'opérations sur les ensembles construits à partir d'union de polyhèdres, ou encore des opérations sur les matrices et les vecteurs. Dans notre outil, l'utilisation de cette librairie est nécessaire pour pouvoir comparer les ensembles d'inégalités représentant les contraintes d'horloges, pour savoir si deux transitions avec la même suite d'actions peuvent malgré tout être acceptée par l'IUT pour l'une et refusée pour l'autre, et ce grâce à leurs contraintes de temps (ie leur garde). Ce problème est non trivial lorsque l'on manipule plusieurs horloges.

6.2 Algorithme de génération de séquences

Une description complète de cet algorithme se trouve dans [FH02]. Il se décompose en trois étapes distinctes, avec toutefois certains aspects d'une partie réutilisés dans les suivantes. Les étapes sont classées selon leur efficacité : la plus efficace en premier. Le principe de cet algorithme est d'essayer d'identifier tous les états contrôlables de la spécification $\mathcal{S} = (\Sigma, S, s^0, C, E)$ dans l'IUT. On ne peut identifier que les états contrôlables, car il faut que ce soit une entrée qui initie la séquence de test. Nous allons travailler sur des séquences extraites du TIOA, et pour cela, quelques définitions sont nécessaires. Notons que la définition de séquence reste inchangée.

Définition 6.2.1 (Séquence possible d'identification) *T est une séquence possible d'identification si :*

- *elle est extraite à partir d'un chemin de la spécification $\mathcal{S}$, partant d'un état contrôlable ;*
- *la première action de la séquence est une entrée ;*
- *la dernière action de la séquence est une sortie.*

Définition 6.2.2 (Séquence acceptée par un état) *Une séquence* $T = \{Tr_1.Tr_2...Tr_n\}$ *avec* $Tr_i = (a_i, \lambda_i, \delta_i)$, $1 \leq i \leq n$ *est acceptée par un état* s *de* $\mathcal{S}$ *s'il existe un chemin* $T' = \{t'_1.t'_2...t'_n\} \in E^n$ *(ie une suite de transitions) de* $\mathcal{S}$, *avec* $t'_i =< s_i, s'_i, a'_i, \lambda'_i, \delta'_i >$ *tel que :*

- $s_1 = s$
- $\forall i | 1 \leq i \leq n$, $a_i = a'_i$ *et* $\lambda_i = \lambda'_i$ *et* $\delta_i \cap \delta'_i \neq \emptyset$.

Définition 6.2.3 (Profondeur) *On appelle profondeur d'une séquence le nombre d'actions d'entrée qu'elle contient.*

Définition 6.2.4 (Séquences similaires) *Deux séquences* T *et* T' *sont dites similaires si* $\forall i | 1 \leq i \leq n$, $a_i = a'_i$ *et* $\lambda_i = \lambda'_i$ *et* $\delta_i \cap \delta'_i \neq \emptyset$.

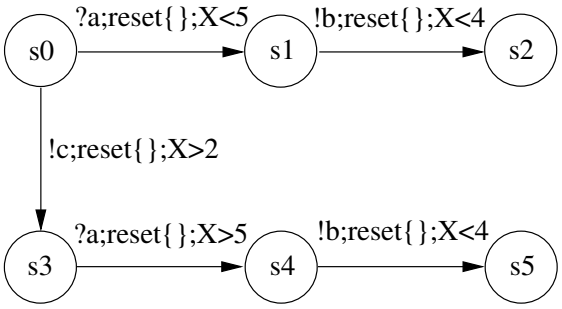


FIG. 6.3 – Exemple pour les séquences acceptées et similaires

Sur la figure 6.3, l'état s_0 accepte comme séquence $(?a, -, X < 1), (!b, -, X < 4)$. De même, l'état s_3 accepte comme séquence $(?a, -, X > 2), (!b, -, X < 4)$. Ces deux séquences sont différentes uniquement sur les deux inéquations $(X < 1)$ et $(X > 2)$ qui sont disjointes. Si on avait eu $(X < 1)$ et $(X < 2)$, ces deux séquences auraient été similaires (elles auraient été considérées comme acceptées par les deux états). Notons que l'affichage est un peu différent ici, car il s'agit d'un TIOA tel que l'outil KRONOS l'affiche.

La première étape de l'algorithme s'inspire de la méthode UIO, adaptée dans le cas d'un TIOA. Ainsi, on essaie d'identifier un maximum d'états contrôlables à l'aide de cette méthode. Celle-ci a l'avantage de donner des séquences de test assez courtes et d'avoir une complexité plus faible que les étapes suivantes. En revanche, il n'est pas toujours possible de trouver une telle séquence par état contrôlable. Dans le même temps, si on ne trouve pas de séquence UIO pour un état, on mémorise les séquences acceptées par le moins d'états différents de celui-ci, afin de préparer la deuxième étape.

La deuxième étape est une étape plus "empirique". Elle se base sur la méthode Wp, appliquée uniquement aux états contrôlables non identifiés à la première étape. Le principe est d'appliquer des séquences à un ensemble d'états, et de créer de nouveaux ensembles selon le fait qu'ils acceptent ou pas la séquence. Tous les états sont identifiés lorsque les ensembles sont tous des singletons. Lors de cette étape, on applique uniquement les séquences mémorisées lors de l'étape précédente, partant du principe qu'elles ont plus de

chances de mener à des singletons. La complexité de cette étape est plus élevée que la précédente, et les séquences de test sont en moyenne plus longues.

La troisième étape est une adaptation de la méthode Wp aux TIOA. Il s'agit d'une méthode systématique qui se base sur la même principe que la deuxième étape, mais en essayant toutes les séquences possibles. On l'applique à tous les états contrôlables non identifiés, ie qui ne sont pas dans des singletons. La minimilaté de l'automate nous assure que la méthode se termine : il existe toujours un chemin pour identifier deux états. En revanche, la complexité théorique est certes identique à l'étape précédente, mais le fait d'essayer les séquences de façon arbitraire et systématique augmente énormément le temps de calcul par rapport à la deuxième étape, où le choix des séquences était plus judicieux. C'est ce qui explique que cette technique soit finalement le dernier recours.

6.2.1 Première étape

Présentation de l'étape

Cette première partie consiste à générer, pour chaque état contrôlable, toutes les séquences possibles partant de celui-ci avec une profondeur maximale donnée (en paramètre lors du lancement du programme). Ensuite, on prend chaque séquence une par une et on vérifie si un autre état l'accepte (différent de celui à partir duquel elle à été générée). Deux cas se présentent alors :

- Si aucun autre état ne l'accepte, on a alors identifié l'état courant par cette séquence. On la garde donc, on marque l'état comme identifié et on passe à un autre.
- Sinon, on continue jusqu'à avoir épuisé toutes les séquences partant de l'état courant. Si aucune d'elles n'identifie l'état, on garde celle qui a été reconnue par le moins d'états et elle sera utilisée dans la seconde partie de l'algorithme. On note $LRSeq$ l'ensemble de ces séquences conservées.

Si à la fin de cette étape, tous les états contrôlables ont été identifiés, alors l'algorithme est terminé. Cette partie s'inspire de la méthode UIO ([SD88]).

Algorithme de cette étape

```
1    fonction step1
2    debut
3      d <- 1;
4      tantque (( il reste des états à identifier )
5                        && ( d < profondeur_max ))
6        état_courant = prochain état à identifier;
7        tantque ( état_courant <> NULL )
8          identifié <- FAUX;
9          tantque (( identifié == FAUX )  &&
10                ((séquence_courante=prochaine séquence correcte) !=NULL))
11           autre_état <- (un des états identifiable <> état_courant);
12           reconnu <- 0;
13           tantque ( autre_état <> NULL )
```

```
14              si ( autre_état accepte séquence_courante )
15                reconnu <- reconnu + 1;
16              finsi
17              autre_état <- (état identifiable suivant <> état_courant);
18            fintantque
19            si ( reconnu == 0 )
20              identifié = VRAI;
21              on marque état_courant comme identifié;
22              SavedSeq.seq[état_courant] = séquence_courante;
23            sinon // reconu > 0
24              si ( SavedSeq.seq[état_courant] == NULL )
25                SavedSeq.seq[état_courant] = séquence_courante;
26                SavedSeq.reco <- reconnu;
27              sinon
28                si ( reconnu < SavedSeq.reco )
29                  SavedSeq.seq[état_courant] = séquence_courante;
30                  SavedSeq.reco <- reconnu;
31                finsi
32              finsi
33            finsi
34          fintantque
35          état_courant = prochain état à identifier;
36        fintantque
37        d <- d + 1;
38      fintanque
39    fin
```

Exemple d'exécution

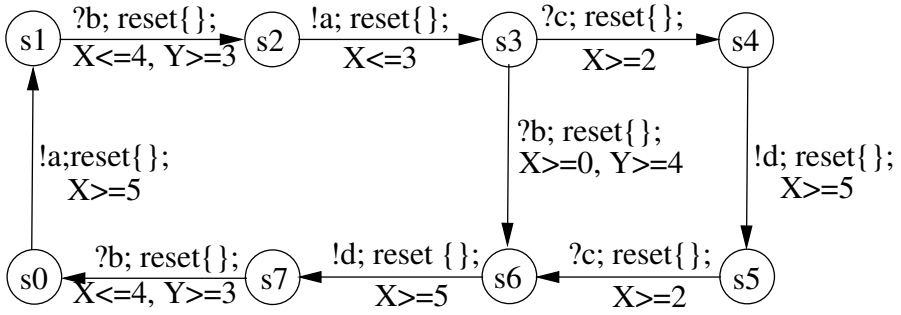


FIG. 6.4 – Automate ne nécessitant que le passage à l'étape 1

Sur l'automate de la figure 6.4. On s'aperçoit qu'il y a 4 états de cet automate qui sont contrôlables : s_1, s_3, s_5 et s_7.
Pour la profondeur 1, les séquences suivantes sont des séquences possibles d'identification :

– s_1 : 1 seule séquence : $((?b, -, X <= 4 \text{ et } Y >= 3), (!a, -, X <= 3))$,

- s_3 : 2 séquences : $((?c, -, X >= 2), (!d, -, X >= 5))$ et $((?b, -, X >= 0 \text{ et } Y >= 4), (!d, -, X >= 5))$
- s_5 : 1 seule séquence : $((?c, -, X >= 2), (!d, -, X >= 5))$,
- s_7 : 1 seule séquence : $((?b, -, X <= 4 \text{ et } Y >= 3), (!a, -, X >= 5))$.

Les deux états s_1 et s_7 ont les même étiquettes. Cependant, en regardant les gardes sur la deuxième transition de ces deux séquences, on s'aperçoit qu'elles sont toutes les deux disjointes. Ce qui veut dire que ces deux séquences sont non similaires. Ces deux états sont donc identifiés.

Pour s_3, la seule séquence qui le différencie est $((?b, -, X >= 0 \text{ et } Y >= 4), (!d, -, X >= 5))$.

Par contre, pour l'état s_5, sa seule séquence possible est aussi acceptée par l'état s_3. Il faut donc aller à la profondeur 2 pour trouver une séquence pour l'état s_5 : ($?c$, $!d$, $?b$, $!a$). Notons que pour alléger les écritures, une séquence est parfois écrite uniquement comme une succession d'actions et non de triplets, lorsque il n'y a aucune ambiguité par rapport à la spécification (et lorsqu'on connaît l'état de départ). Il faut toutefois lire ces séquences comme une suite de triplets.

Pour résumer, voici le résultat pour cet automate :

- s_0 : pas contrôlable,
- s_1 : identifié par la séquence ($?b$, $!a$),
- s_2 : pas contrôlable,
- s_3 : identifié par la séquence ($?b$, $!d$),
- s_4 : pas contrôlable,
- s_5 : identifié par la séquence ($?c$, $!d$, $?b$, $!a$),
- s_6 : pas contrôlable,
- s_7 : identifié par la séquence ($?b$, $!a$).

Cet exemple d'exécution est donné en Annexe B.

6.2.2 Deuxième étape

Cette partie n'est utilisée que s'il reste des états contrôlables non identifiés après la première étape.

Présentation de l'étape

Cette seconde partie s'appuie sur les résultats de la première. On construit un arbre dont la racine contient l'ensemble des états restant à identifier. On cherche ensuite à diviser cet ensemble de la manière suivante :

- On prend une séquence d'un des états présents dans l'ensemble $LRSeq$ (dit ensemble des meilleures séquences)
- On teste ensuite cette séquence sur chaque état présent dans l'ensemble
- Tous les états acceptant cette séquence seront placés dans le fils droit
- Tous les états n'acceptant pas cette séquence seront placés dans le fils gauche

Et on recommence ceci récursivement sur les fils et ainsi de suite jusqu'à ce que chaque feuille de l'arbre soit un singleton ou qu'il n'y ait plus de séquence dans $LRSeq$.

A la fin, deux cas se présentent.

- Toutes les feuilles de l'arbre sont des singletons. Alors, cela signifie que tous les états sont identifiés, et que l'ensemble des séquences identifiant un état se lit en parcourant le chemin de l'arbre allant de la racine à cet état, et en stockant les séquences ayant permis de scinder les ensembles.
- Il reste des feuilles qui ne sont pas des singletons. Dans ce cas, tous les états ne sont pas identifiés, et la troisième étape est alors nécessaire.

Une fois une série de séquences de test trouvée (pour un ou plusieurs états), on sait que cette série distingue le ou les états de tous les autres présents au départ à la racine de l'arbre. Mais on ne peut pas assurer qu'aucun état identifié à la première étape (et donc non présent à la racine) ne reconnaît cette série. Il faut donc tester les séries de séquences sur chaque état identifié à la première étape. Si l'un d'eux répond positivement, alors on ajoute sa séquence (qui est unique de par sa construction) à la série mais en précisant qu'elle ne sera pas acceptée. De cette manière la série obtenue distinguera bien le ou les états de tous les autres.

Lors de cette étape, on utilise l'ensemble $LRSeq$ pour scinder les ensembles d'états car il y a plus de chance que des séquences ayant été acceptées par le plus petit nombre d'états identifient plus rapidement tous les états. C'est donc une étape plus "pragmatique" mais dont la complexité théorique est égale à la troisième étape. Dans la pratique, cette étape couplée à la première donne de très bons résultats. Il est très rare que l'étape suivante soit utilisée.

Algorithme de cette étape

```
1    fonction buildtree_step2(arbre)
2    debut
3      trouve <- 0;
4      tantque (( trouve == 0 ) && ( il reste des états non marqués ))
5        état_courant <- un des états non marqués de arbre;
6        séquence_courante <- SavedSeq.seq[état_courant];
7        on marque état_courant;
8        node_equ <- nouveau noeud;
9        node_non_equ <- nouveau noeud;
10       pour chaque élément de arbre
11         si ( élément accepte séquence_courante )
12           node_equ <- élément;
13         sinon
14           node_non_equ <- élément;
15         finsi
16       finpour
17       si (( nb_élément(node_equ) != 0 ) && ( nb_élément(node_equ) != 0 ))
18         trouve <- 1;
19       sinon
```

```
20          détruire node_equ et node_non_equ;
21        finsi
22      fintantque
23      si ( trouve == 1 )
24        fils_droit <- node_equ;
25        si ( nb_élément(node_equ) > 1 )
26          buildtree_step2(fils_droit);
27        finsi
28        fils_gauche <- node_non_equ;
29        si ( nb_élément(node_non_equ) > 1 )
30          buildtree_step2(fils_gauche);
31        finsi
32      finsi
33    fin
34
35    fonction get_seq_tree_step2(arbre)
36    debut
37      parcours de l'arbre en profondeur pour récupérer les
38                                        listes de séquences;
39    fin
40
41    fonction step2
42    debut
43      OS <- ensemble des états non identifiés à l'étape 1 ayant
44                                         une meilleure séquence;
45      arbre <- nouvel_arbre();
46      arbre.elem <- OS;
47      buildtree_step2(arbre);
48      get_seq_tree_step2(arbre)
49    fin
```

Exemple d'exécution

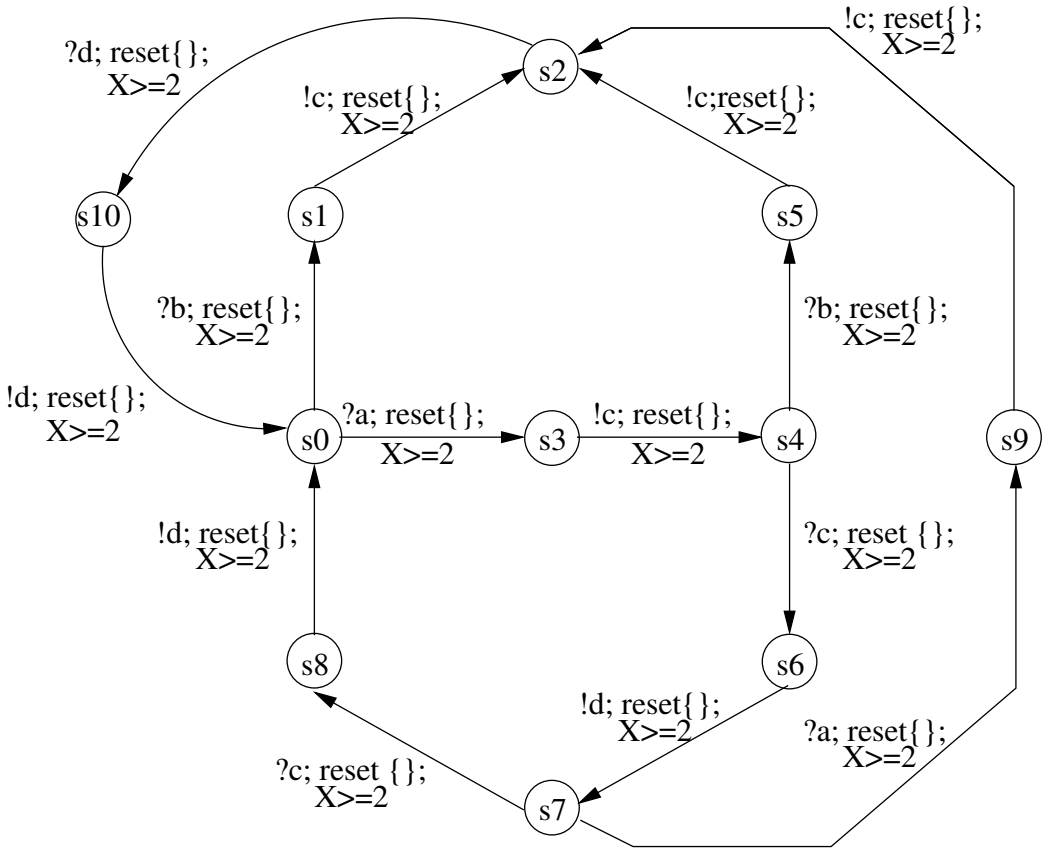


FIG. 6.5 – Automate nécessitant un passage à l'étape 2 mais pas à l'étape 3

L'automate pris est celui de la figure 6.5. On remarque que celui-ci est déterministe et qu'il aurait très bien pu être reconnu à l'étape 1 avec une profondeur appropriée (au moins 2). Ici, on prendra une profondeur de 1. Les états contrôlables {0, 4 et 7} ne sont pas identifiés à l'étape 1. Comme meilleure séquence pour chaque, on prend respectivement ($?b$, $!c$), ($?c$, $!d$) et ($?a$, $!c$), qui représentent l'ensemble $LRSeq$. Toutes ces séquences sont acceptées par deux états.

On démarre donc avec un arbre contenant à la racine ces états. Puis, on prend le premier d'entre eux (ici 0) et on regarde sa séquence qui est ($?b$, $!c$). 0 et 4 l'acceptent mais pas 7. On place donc les deux premiers dans le fils droit et l'autre dans le fils gauche de l'arbre. Puis, comme le fils droit contient encore un ensemble, on cherche à le diviser.

On prend l'état 4 et sa séquence ($?c$, $!d$). 0 ne la reconnaît pas, donc on le place dans le fils gauche et on met 4 dans le fils droit. On se retrouve donc avec un arbre ne contenant que des feuilles avec des ensembles singleton et on a donc fini d'identifier ceux-ci. L'arbre obtenu est celui de la figure 6.6.

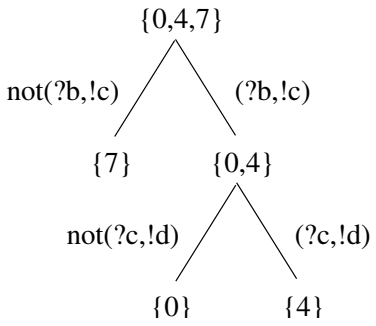


FIG. 6.6 – Arbre obtenu à l'étape 2 sur l'automate de la figure 6.5

Il reste ensuite à parcourir cet arbre pour obtenir la liste des séquences de chaque état. On obtient donc le résultat suivant :

- s_0 : contrôlable par les séquences ($?b$, $!c$) et non($?c$, $!d$),
- s_1 : pas contrôlable,
- s_2 : contrôlable par la séquence ($?d$, $!d$),
- s_3 : pas contrôlable,
- s_4 : contrôlable par les séquences ($?b$, $!c$) et ($?c$, $!d$),
- s_5 : pas contrôlable,
- s_6 : pas contrôlable,
- s_7 : contrôlable par les séquences ($?a$, $!c$) et non($?b$, $!c$)
- s_8 : pas contrôlable,
- s_9 : pas contrôlable,
- s_{10} : pas contrôlable.

La notation non($?b$, $!c$) signifie que l'IUT doit refuser la séquence ($?b$, $!c$).

L'Annexe C montre que notre outil donne des résultats différents, mais aussi corrects. Cela dépend de l'ordre des séquences choisies.

6.2.3 Troisième et dernière étape

Cette étape n'est utile que s'il reste des états contrôlables non identifiés à la suite des deux étapes précédentes.

Présentation de l'étape

Le principe de cette étape est relativement similaire à l'étape précédente, mais cette fois de façon plus systématique. Cette partie s'inspire de la méthode Wp ([FBK+91]). On reprend chaque ensemble d'états de l'étape 2 qui est une feuille de l'arbre mais pas un singleton, et on lui applique le même procédé que l'étape 2, mais cette fois avec toutes les séquences possibles d'identification si nécessaire. On essaie avec des séquences de plus en plus longues. Cette étape est certes assez peu efficace, mais en cas d'automate minimal, nous sommes certains qu'elle se termine, car il existe toujours un chemin pour identifier

deux états (grâce à la minimalité). Dans le cas d'un automate non minimal, une profondeur limite de séquences à essayer est fixée, mais dans ce cas, on ne garantie pas le succès de l'algorithme, en revanche on garantie qu'il se termine.

Algorithme de cette étape

```
1    fonction buildtree_step3(arbre,d)
2    debut
3      trouve <- 0;
4      tantque ( trouve == 0 )
5        i <- 0;
6        tantque (( i < nb_élément(arbre) ) && ( trouve == 0 ))
7          état_courant <- état numéro i de l'arbre;
8          tantque (( il reste des séquences ) && ( trouve == 0 ))
9            node_equ <- nouveau noeud;
10           node_non_equ <- nouveau noeud;
11           pour chaque élément de arbre
12             si ( accepte séquence_courante )
13               node_equ <- élément; // ( un des états )
14             sinon
15               node_non_equ <- élément;
16             finsi
17           finpour
18           si (( nb_élément(node_equ) != 0 ) &&
19                   ( nb_élément(node_equ) != 0 ))
20             trouve <- 1;
21           sinon
22             détruire node_equ et node_non_equ;
23           finsi
24         fintantque
25         si ( trouve != 1 )
26           i <- i + 1 ;
27         finsi
28       fintantque
29       si ( trouve != 1 )
30         d <- d + 1 ;
31       finsi
32     fintantque
33     si ( trouve == 1 )
34       fils_droit <- node_equ;
35       si ( nb_élément(node_equ) > 1 )
36         buildtree_step3(fils_droit,d);
37       finsi
38       fils_gauche <- node_non_equ;
39       si ( nb_élément(node_non_equ) > 1 )
```

```
40          buildtree_step3(fils_gauche,d);
41        finsi
42      finsi
43    fin
44
45    fonction get_seq_tree_step3(arbre)
46    debut
47      parcours de l'arbre en profondeur pour récupérer les
48                                        listes de séquences;
49    fin
50
51    fonction step3
52    debut
53      ajout d'un ensemble contenant des états n'ayant pas de meilleure
54      séquence à l'étape 1 dans la liste d'ensemble contenant les états
55      non identifiés à l'étape 2;
56      pour chaque élément de la liste d'ensemble non identifiés
57                                                    à l'étape 2
58        arbre <- nouvel_arbre();
59        arbre.elem <- élément_courant;
60        buildtree_step3(arbre,1);
61      finpour
62      pour chaque élément de la liste d'ensemble
63        get_seq_tree_step3(arbre);
64      finpour
65    fin
```

Exemple d'exécution

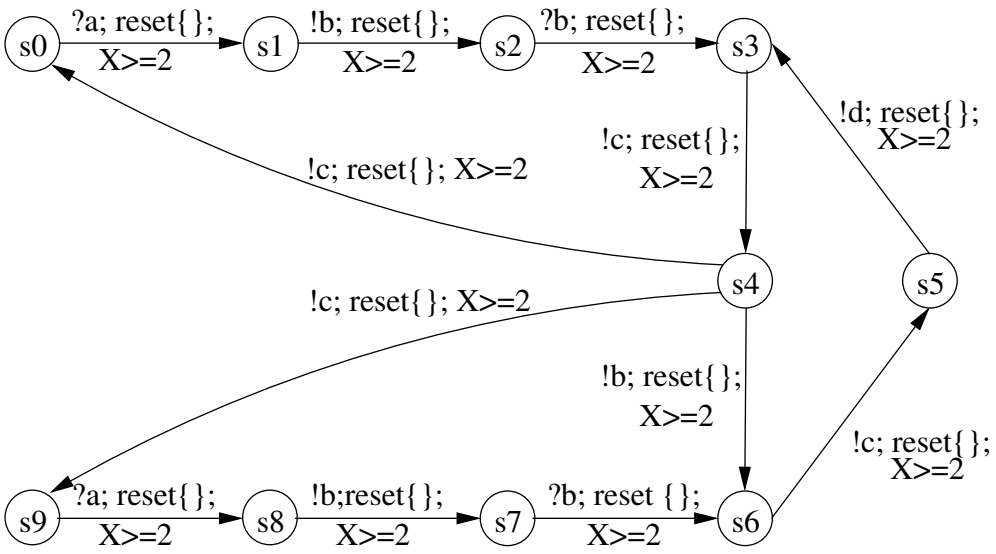


FIG. 6.7 – Automate nécessitant un passage à l'étape 3

L'exemple pris est l'automate de la figure 6.7. On remarque qu'il aurait suffi de l'étape 1 pour identifier tous les états, en fixant une profondeur adaptée. Les états contrôlables sont les états {0, 2, 7 et 9}.

Si on ne prend qu'une profondeur de 1 pour l'exécution, seuls les états 2 et 7 sont identifiés. Pour l'étape 2, on part donc avec cet ensemble : {0, 9}.

Vu qu'ils reconnaissent tous les deux la même séquence, l'étape 2 est inutile et on n'arrive pas à les séparer. On est donc obligé d'aller à l'étape 3. Pour les différencier, on trouve une séquence de profondeur 2.

Au final, voici le résultat pour cet automate :
- s_0 : contrôlable par les séquences ($?a$, $!b$) (meilleure séquence trouvée à l'étape 1) et ($?a$, $!b$, $?b$, $!c$, $!c$),
- s_1 : pas contrôlable,
- s_2 : contrôlable par la séquence ($?b$, $!c$, $!c$),
- s_3 : pas contrôlable,
- s_4 : pas contrôlable,
- s_5 : pas contrôlable,
- s_6 : pas contrôlable,
- s_7 : contrôlable par les séquences ($?b$, $!c$, $!d$),
- s_8 : pas contrôlable,
- s_9 : contrôlable par les séquences ($?a$, $!b$) (meilleure séquence trouvée à l'étape 1), non($?b$, $!c$, $!d$) (négation de 7), non($?b$, $!c$, $!c$) (négation de 2) et non($?a$, $!b$, $?b$, $!c$, $!c$) (négation de 0).

L'exemple donné en Annexe D donne les mêmes résultats.

6.3 Analyse quantitative de la méthode

Pour analyser l'efficacité de cet outil, nous avons effectué des simulations de génération de test sur un nombre élevé d'automates et nous avons observé les résultats.

6.3.1 Générateur d'automates et statistiques

Afin de pouvoir évaluer l'efficacité de chaque partie de la méthode de génération, nous avons développé en parallèle un générateur aléatoire d'automates. Celui-ci a pour rôle de construire aléatoirement une grande quantité d'automates (des TIOA), qui serviront ensuite d'entrée pour la génération des séquences (il s'agit d'un système d'automates). Cette outil assure certaines conditions sur les automates :

- connexité des automates (sinon, risques de bouclages infinis dans l'étape systématique),
- les états doivent être soit des états de sortie, soit des états d'entrée (c'est à dire un état n'ayant que des transitions de sortie pour un état de sortie et que des transitions d'entrée pour un état d'entrée),
- déterministes.

Il est aussi possible de choisir le nombre d'automates que l'on souhaite créer.

6.3.2 Résultats

Nous exposons ici les statistiques obtenues, en jouant sur différents paramètres : la profondeur, le nombre d'états et le nombre d'horloges. Nous avons sélectionné ici les valeurs qui présentent un intérêt. Ces résultats ont été expérimentés sur 100 différents automates, avec 10, 20, 50 et 100 états. Sur ces figures, l'axe des ordonnées représente le pourcentage d'états identifiés par rapport au nombre total d'états contrôlables.

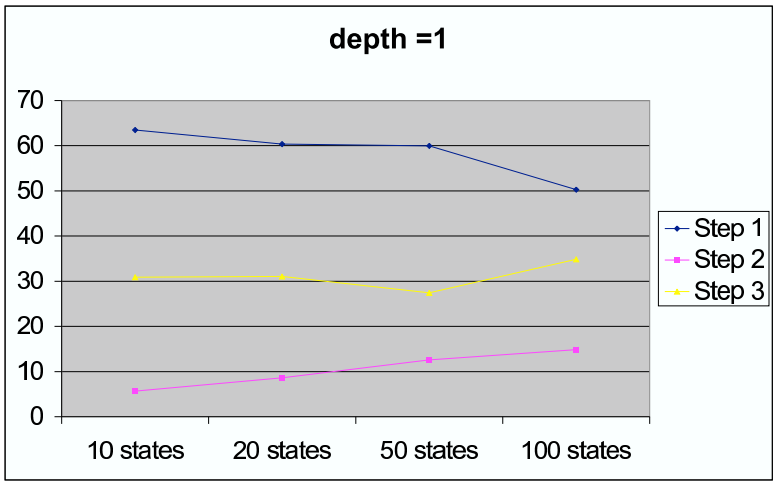


FIG. 6.8 – Analyse de l'algorithme à la profondeur 1

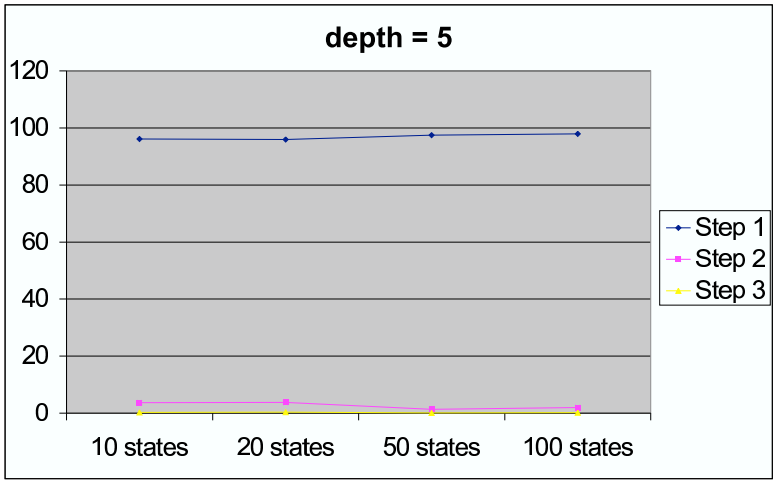


FIG. 6.9 – Analyse de l'algorithme à la profondeur 5

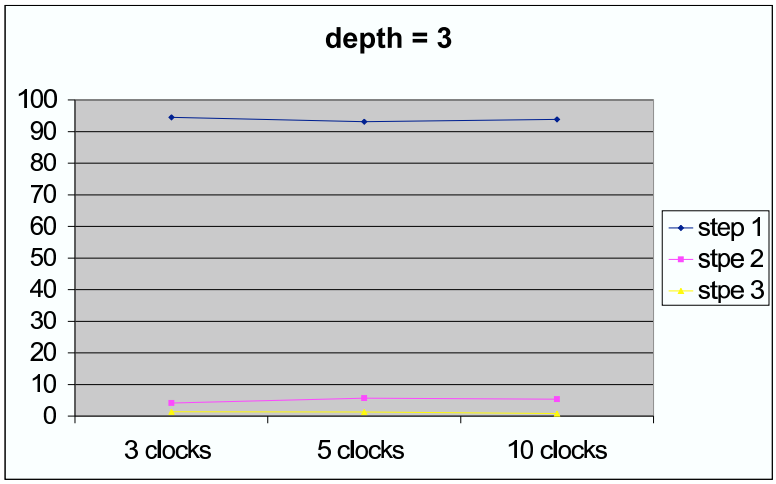


FIG. 6.10 – Analyse de l'algorithme avec différents nombres d'horloges

L'élément essentiel de ces résultats, montré nettement par les trois figures, c'est que la grande majorité des états sont identifiés dès la première étape de l'algorithme. C'est la cas de façon très nette à partir de la profondeur 3. La deuxième étape permet d'identifier quelques états, alors que le nombre d'états identifiés par la dernière phase (la plus coûteuse) est minime.

Par ailleurs, la figure FIG. 6.8 montre que le pourcentage d'états identifiés à la première étape diminue avec le nombre d'états pour une profondeur faible. Cela s'explique parce

que la profondeur n'est que de 1 : il est plus difficile de trouver une séquence avec une seule entrée pour chaque état, et en particulier quand le nombre d'états augmente.

La figure FIG. 6.9 montre qu'à partir d'une certaine profondeur (de l'ordre de 3), le nombre d'états n'a quasiment plus d'influence sur les résultats.

La figure FIG. 6.10 montre que le nombre d'horloges n'a pas d'influence sur les résultats.

Il faut noter que la génération aléatoire d'automates peut être considérée comme éloignée de spécifications réelles, ie peu réaliste, et que par conséquent ces résultats peuvent s'avérer discutables. Toutefois, ils permettent d'obtenir un ordre de grandeur sur la répartition des états identifiés par les différentes étapes de notre outil. Il y a de grandes chances que sur des spécifications réelles, la répartition suive le même ordre de grandeur.

Ainsi, la très grande majorité des états contrôlables est identifiée lors de la première étape de l'algorithme. Celle-ci a une complexité plus faible que les autres étapes, et les séquences générées sont plus courtes. Il reste une faible proportion d'états contrôlables qui sont en général tous identifiés à la deuxième étape, qui a une complexité plus grande. La troisième étape, dont l'efficacité est la plus mauvaise, n'est quasiment jamais nécessaire.

6.4 Application pour la robustesse

Pour obtenir des séquences applicables pour tester la robustesse, nous offrons la possibilité à l'utilisateur d'insérer des aléas dans les séquences déjà générées. Cette insertion se fait selon les choix de l'utilisateur, mais toutefois dans un cadre prédéfini. Ensuite, ces séquences mutantes pourront être données au testeur.

Nous avons vu dans les chapitres précédents que pour qu'une séquence soit utilisable pour tester la robustesse, nous faisions l'hypothèse qu'elle était extraite d'un chemin de la spécification qui finissait sur un état contrôlable. Pour pouvoir prendre en compte ce point, nous modifions la définition précédente de séquence possible d'identification dans le cadre de la robustesse.

Définition 6.4.1 (Séquence possible d'identification pour la robustesse) *T est une séquence possible d'identification pour la robustesse si :*

- *elle est extraite à partir d'un chemin de la spécification $\mathcal{S}$, partant d'un état contrôlable et finissant sur un état contrôlable ;*
- *la première action de la séquence est une entrée ;*
- *la dernière action de la séquence est une sortie.*

Seules les séquences devant être acceptées par un état seront retenues. On n'applique pas les séquences censées être refusées, car ici ce n'est pas la conformité qui est évaluée, mais la robustesse. Dans ce cas, la notion de refus n'a pas de sens.

Une fois l'ensemble de séquences obtenu, l'utilisateur peut les modifier de façon assez libre. Toutefois, il ne peut modifier que des actions d'entrée dans la séquence. En effet, modifier une action de sortie n'aurait aucun sens, étant donné que c'est une réponse de la part de l'IUT.

Ainsi, l'utilisateur peut modifier une ou plusieurs séquences de la façon suivante :

- **Modification d'un triplet correspondant à une entrée** :
  L'utilisateur peut modifier le label de l'action (cas de remplacement d'une action d'entrée) par une autre entrée, ou modifier la contrainte d'horloge du triplet (modification de l'instant d'occurrence).
- **Ajout d'une action d'entrée** :
  L'utilisateur choisit un endroit dans la séquence où il souhaite insérer une action d'entrée supplémentaire. Ainsi, il ajoute un triplet contenant une action d'entrée et une contrainte d'horloge. Il ne peut pas écrire de réinitialisation.
- **Suppression d'une action d'entrée** :
  L'utilisateur choisit un triplet contenant une entrée dans la séquence. Ensuite, ce triplet est supprimé de celle-ci.

Ces trois types de modifications laissent une large liberté à l'utilisateur pour modifier les séquences. Il peut notamment effectuer toutes les possibilités d'insertion manuelle d'aléas vues au chapitre 4. En revanche, il doit prendre garde à ce que les séquences nouvellement créées soient cohérentes, en particulier au niveau des timings.

Les résultats de ce chapitre sont publiés dans [FRRT04] et [FPG+05].

Chapitre 7

Conclusion et perspectives

L'objectif principal de ce travail était de donner des pistes pour tester la robustesse d'un système temps-réel. Nous nous sommes essentiellement intéressés à son aspect comportemental en cas de situation inattendue.

7.1 Contributions

Dans le chapitre 4, nous avons exposé une méthode pour tester la robustesse, avec prise en compte des aspects temporels. Le concepteur fournissait deux spécifications, une nominale pour décrire le comportement en conditions normales, et une dégradée regroupant les actions vitales, qui devaient être assurées par l'IUT dans les situations critiques et inattendues. A l'aide d'une des deux spécifications (selon la méthode choisie), nous avons généré des séquences de test temporisées. Dans le cas de l'insertion d'aléas externes, nous avons inséré dans les séquences des événements inattendus, afin de créer artificiellement une situation critique. Toutes l'exécution était enregistrée dans les traces. Après la session de test, nous avons utilisé ces traces pour évaluer si le comportement peut être considéré comme "acceptable". Pour cela, nous nous sommes intéressés aux actions vitales, décrites dans la spécification dégradée du système, et nous avons utilisé une relation d'acceptation permettant de vérifier si les traces d'exécution obtenues lors de la session de test révélaient un comportement robuste. Pour évaluer ce comportement, la relation d'acceptation s'est appliquée sur les traces d'exécution, en se basant sur la spécification dégradée. L'originalité de ce travail, c'est que nous ne connaissions pas à l'avance le comportement précis de l'IUT, mais nous devions nous assurer que celui se trouvait dans les limites acceptables, représentées par les deux spécifications : la nominale et la dégradée.

Par ailleurs, dans le chapitre 5, nous avons proposé une approche pour tester un système temps-réel à base de composants. La contribution principale est la présentation d'une architecture de test adaptée pour ce genre de système, ainsi qu'un algorithme complet pour l'exécution des tests. Chaque composant est relié à son propre testeur, et toutes les communications entre composants vont devoir passer par les testeurs correspondants. Cette architecture est applicable aussi bien au test de conformité qu'au test de robus-

tesse. Dans le dernier cas, il est nécessaire d'avoir en plus une spécification dégradée par composant, afin de pouvoir évaluer si son comportement a été acceptable.

Enfin nous avons présenté au chapitre 6 un outil de génération de séquences de test que nous avons développé. Il permet de traiter divers formats de fichiers correspondant aux automates temporisés, et de générer des séquences temporisées, utilisables pour tester la conformité. Il est aussi possible d'insérer des aléas dans ces séquences, et ainsi de les utiliser pour tester la robustesse du système. Le principe de cette méthode est de générer des séquences permettant d'identifier précisément les états de la spécification dans l'IUT. Il s'inspire des méthodes UIO et Wp, adaptées pour un système temporisé.

7.2 Nouvelles perspectives

Pour conclure cette thèse, nous présentons certaines pistes de recherche et extensions possibles dans le domaine du test de robustesse.

Application sur un système réel

A court terme, il serait intéressant de tester notre outil de génération de séquences sur un système réel. Il faut pour cela trouver une spécification qui respecte les restrictions exprimées dans les parties précédentes (déterminisme, connexité, ...). Nous envisageons de travailler sur le protocole multimédia H223.

Verdict de robustesse et spécifications

En ce qui concerne le verdict de robustesse, nous avons choisi une réponse booléenne (robuste ou non). Or, étant donné que celui-ci va dépendre des aléas que l'on insère, il serait intéressant de pouvoir l'exprimer de façon quantitative, ie une sorte de mesure de la robustesse. Comme nous l'avons vu, l'évaluation du verdict de robustesse que nous appliquons se présente comme une réponse à la question : " le système a-t-il un comportement situé entre celui décrit par la spécification dégradée et celui décrit par la nominale ?". Ainsi, en travaillant sur un représentation plus fine des spécifications, il serait possible d'affiner le verdict. On peut envisager par exemple de travailler avec plus de spécifications différentes, et définir des transformations permettant de définir les spécifications les unes par rapport aux autres. Ainsi, on pourrait introduire plusieurs niveaux de spécification dégradée, comme proposé dans [AS202]. Par extension, on peut envisager de travailler sur la possibilité de situer précisément le comportement du système (entre les deux spécifications) uniquement en lui appliquant des séquences de test bien choisies.

Insertion des aléas

D'autre part, l'insertion des aléas dans les séquences de test pourrait sans doutes être améliorée. Il faudrait pour cela s'inspirer plus du côté expérimental. L'idée serait de collecter les différents aléas rencontrés dans la vie de divers systèmes dans une base de données,

et de la mettre à jour au fur et à mesure. Ainsi, on obtiendrait une collection complète d'aléas adaptés au système que l'on teste, ce qui permettrait de cibler les situations que l'on souhaite appliquer au système.

Il serait aussi intéressant d'introduire des métarègles pour l'insertion des aléas et la génération des séquences mutantes.

Choix du modèle

On pourrait aussi envisager de travailler sur des modèles plus adaptés que les automates temporisés pour le test de robustesse, qui permettent de distinguer directement les aspects vitaux sans avoir besoin d'une deuxième spécification. On pourrait par exemple étudier la possibilité de définir une partie de l'automate comme spécification dégradée.

Dans le domaine des systèmes à base de composant, nous avons décidé d'utiliser les automates temporisés comme représentation. Or, il serait bienvenu de s'orienter vers un modèle plus adapté aux composants, et éprouvé. En effet, les automates temporisés ne permettent pas de modéliser des informations importantes, comme celles relatives à la qualité de service par exemple. Par extension, il serait intéressant de trouver des méthodes de génération de séquences de test adaptées à ces nouveaux modèles.

Un environnement général de certification

Toujours dans le domaine des systèmes à base de composant, il serait intéressant de mettre au point un environnement complet de validation à partir d'un modèle normalisé de SBC. L'idée ensuite serait de l'inclure dans un environnement complet de développement de ce genre de systèmes.

Bibliographie

[ABP+97] C. André, F. Boulanger, M.A. Péraldi, J.P. Rigault, and G. Vidal-Naquet. Objects and synchronous programming. *RAIRO–APII–JESA*, 31(3) :417–432, 1997.

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In LICS'90 [LIC90], pages 414–425.

[ACE87] Eric Audureau, Luis Fariñas Del Cerro, and Patrice Enjalbert. Théorie de la programmation et logique temporelle, validation d'algorithmes séquentiels. *Technique et Science Informatiques*, 6(6) :527–540, November 1987.

[ACH+92] R. Alur, C. Courcoubetis, N. Halbwachs, D. Dill, and H. Wong-Toi. Minimization of timed transition systems. In Cleaveland [Cle92], pages 340–354.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T.A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138 :3–34, 1995.

[ACLdSS04] Ludovic Apvrille, Jean-Pierre Courtiat, Christophe Lohr, and Pierre de Saqui-Sannes. Turtle : A real-time uml profile supported by a formal validation toolkit. *IEEE Trans. Software Eng*, 30(7) :473–487, 2004.

[ACM+96] R. Anido, A.R. Cavalli, T. Macavei, L.P. Lima, M. Clatin, and M. Phalippou. Testing a real protocol with the aid of verification techniques. In *XXII SEMISH, Brazil*, pages 237–248, August 1996.

[AD94] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126 :183–235, 1994.

[AD01] C. André and R. De Simone. Programmation synchrone : Propriétés dans une réaction. In *MSR'2001*, pages 547–561, Toulouse (F), 17-19 octobre 2001. Hermès Science Publications.

[AdSSL+01] L. Apvrille, P. de Saqui-Sannes, C. Lohr, P. Sénac, and J.-P. Courtiat. A new uml profile for real-time system formal design and validation. In *Fourth International Conference on the Unified Modeling Language (UML'2001), Toronto, Canada*, lncs, pages 287–301. Springer, October 2001.

[AFH94] R. Alur, L. Fix, and T.A. Henzinger. Event-clock automata : a determinizable class of timed automata. In Dill [Dil94], pages 1–13.

[AG97] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3) :213–249, 1997.

[AH90] L. Aceto and M. Hennessy. Adding action refinement to a finite process algebra. Report 6/90, Computer Science, University of Sussex, Brighton, 1990.

[AH92] R. Alur and T.A. Henzinger. Logics and models of real time : a survey. In Bakker et al. [BHRR92], pages 74–106.

[And96] 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.

[AS202] Action spécifique 23 du cnrs, département stic : Test avancé de systèmes complexes, test de robustesse, 2002. Animateurs : Richard Castanet et Hélène Waeselynck.

[BALC95] Hanene Ben-Abdallah, Insup Lee, and Jin-Young Choi. A graphical language with formal semantics for the specification and analysis of real-time systems. In IEEE Computer Society Press, editor, *16th IEEE Real-Time Systems Symposium (RTSS'95), Palazzo dei Congressi, Via Matteotti, 1, Pisa, Italy*, December 1995.

[BB04] L. B. Briones and E. Brinskma. A test generation framework for quiescent real-time systems. In *4th International Workshop on Formal Approaches to TEsting of Software (FATES 2004), Linz, Austria*, September 2004.

[BCC03] Ed Brinksma, Geoff Coulson, and Ivica Crnkovic. Project ist-2001-34820 - artist- advanced real-time systems. roadmap : Component-based design and integration platforms, 2003. http ://www.systemes-critiques.org/ARTIST/.

[BCF03] I. Berrada, R. Castanet, and P. Félix. A formal approach for real-time test generation. In *Proceedings of Workshop On Testing Real-Time and Embedded Systems (WTRTES), Satellite Workshop of FM 2003 Symposium, Pisa, Italy*, September 2003.

[BCIM00] A. Bertolino, F. Corradini, P. Inveradi, and H. Muccini. Deriving test plans from architectural descriptions. In *ACM Proceedings, International Conference on Software Engineering ICSE2000*, June 2000.

[BCKZ02] Cédric Besse, Ana R. Cavalli, Myungchul Kim, and Fatiha Zaidi. Automated generation of interoperability tests. In *Testing of Communicating Systems XIV, Applications to Internet Technologies and Services, Proceedings of the IFIP 14th International Conference on Testing Communicating Systems - TestCom 2002, Berlin, Germany*, pages 169–187, March 2002.

[BCS02] E. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and dynamic software composition with sharing. In *Seventh International Workshop on Component-Oriented Programming (WCOP02), At ECOOP 2002, Malaga, Spain*, June 2002.

[BCSS90] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek. Fault injection experiments using fiat. *IEEE Trans. Comput.*, 39(4) :575–582, 1990.

[BD88] S. Budkowski and P. Dembinski. An introduction to estelle : A specification language for distributed systems. *Computer Networks and ISDN Systems*, 14 :3–24, Jan 1988.

[Ber97] G. Berry. The esterel v5 language primer. Technical report, Inria Sophia Antipolis, http ://www.inria.fr/meije/esterel, 1997.

[BFG+99a] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, and Laurent Mounier. If : An intermediate representation and validation environment for timed asynchronous systems. In *World Congress on Formal Methods (1)*, pages 307–327, 1999.

[BFG+99b] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, Laurent Mounier, and Joseph Sifakis. If : An intermediate representation for sdl and its applications. In *World Congress on Formal Methods (1)*, pages 307–327, 1999.

[BFG00a] Marius Bozga, Jean-Claude Fernandez, and Lucian Ghirvu. Using static analysis to improve automatic test generation. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS'2000), Berlin, Germany*, pages 235–250, 2000.

[BFG+00b] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Susanne Graf, Jean-Pierre Krimm, and Laurent Mounier. IF : A validation environment for timed asynchronous systems. In *Computer Aided Verification*, pages 543–547, 2000.

[BGK+00] Marius Bozga, Susanne Graf, Alain Kerbrat, Daniel Vincent, Laurent Mounier, and Iulian Ober. Sdl for real-time : What is missing ? In *The 2nd Workshop of the SDL Forum Society on SDL and MSC (SAM2000), Grenoble - France*, June 2000.

[BGM+01] M. Bozga, S. Graf, L. Mounier, I. Ober, J.-L. Roux, and D. Vincent. Timed extensions for SDL. *Lecture Notes in Computer Science*, 2078 :223–? ?, 2001.

[BHRR92] J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors. *Proceedings REX Workshop on Real-Time : Theory in Practice,* Mook, The Netherlands, June 1991, volume 600 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[BORZ99] L. Du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Lutess : A specification-driven testing environment for synchronous software. In *International Conference on Software Engineering*, pages 267–276, 1999.

[Bou97] J. Boue. *Test de la tolérance aux fautes par injection de fautes dans des modèles de simulation VHDL.* PhD thesis, LAAS, November 1997.

[Bou99] Lydie Du Bousquet. *Test fonctionnel statistique de systèmes spécifiés en Lustre.* PhD thesis, Université Joseph Fourier - Grenoble I, 1999.

[Bou01] P. Bouyer. A logical characterization of data languages. Technical Report LSV-01-10, ENS Cachan, France, 2001.

[Bou02] Patricia Bouyer. *Modèles et Algorithmes pour la Vérification des Systèmes Temporisés*. PhD thesis, Ecole Normale Supérieure de Cachan, 2002.

[Bow01] Howard Bowman. Time and action lock freedom properties for timed automata. In *Formal Techniques for Networked and Distributed Systems, FORTE 2001, IFIP TC6/WG6.1 - 21st International Conference on Formal Techniques for Networked and Distributed Systems, Cheju Island, Korea*, pages 119–134, August 2001.

[Bri87] E. Brinksma. A theory for the derivation of tests. In S. Aggarwal, editor, *Proceedings of the Eighth International Conference on Protocol Specification, Testing and Verification*. North-Holland, 1987.

[BS02] Jan Bredereke and Bernd-Holger Schlingloff. An automated, flexible testing environment for umts. In *Testing of Communicating Systems XIV, Applications to Internet Technologies and Services, Proceedings of the IFIP 14th International Conference on Testing Communicating Systems - TestCom 2002, Berlin, Germany*, pages 79–94, March 2002.

[BSV90] E. Brinksma, G. Scollo, and C.A. Vissers, editors. *Protocol Specification, Testing, and Verification, IX,* Enschede The Netherlands, June 6-9, 1989. North-Holland, 1990.

[BT00] E. Brinksma and J. Tretmans. Testing Transition Systems : An Annotated Bibliography. In F. Cassez, C. Jard, B. Rozoy, and M. Ryan, editors, *Summer School MOVEP'2k – Modelling and Verification of Parallel Processes*, pages 44–50, Nantes, July 2000.

[CA92] W. Chung and P. Amer. Improved on UIO Sequence Generation and Partial UIO Sequences. In Linn and Uyar [LU92].

[Cal00] Projet rnrt calife : Environnement pour la preuve formelle et le test d'algorithmes utilisés en télécommunications, 2000.

[Car96] Goerge J. Carrette. Crashme : Random input testing. http ://people.delphiforums.com/gjc/crashme.html, 1996.

[CAV96] *Proceedings of the 8th International Conference on Computer Aided Verification,* New Brunswick, New Jersey, USA, volume 1102 of *Lecture Notes in Computer Science*. Springer-Verlag, August 1996.

[CE81] E.M. Clarke and E.A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs,* Yorktown Heights, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[CES86] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2) :244–263, 1986.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. Model-checking. *The MIT Press, Cambridge, Massachusets*, 1999.

[Cho78] T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3) :178–187, 1978.

[CI92] G. S. Choi and R. K. Iyer. Focus : An experimental environment for fault sensitivity analysis. *IEEE Transactions on Computers*, 41(12) :1515–1526, December 1992.

[CKL98] Richard Castanet, Ousmane Koné, and Patrice Laurençot. On the Fly Test Generation for Real Time Protocols. In *International Conference on Comnputer Communications and Networks, Lafayette, Louisiana, U.S.A.* IEEE Computer Society Press, October 1998.

[CL97] Duncan Clarke and Insup Lee. Automatic generation of tests for timing constraints from requirements. In *Proceedings of the Third International Workshop on Object-Oriented Real-Time Dependable Systems,* Newport Beach, California, February 1997.

[Cle92] R. Cleaveland, editor. *Proceedings CONCUR 92,* Stony Brook, NY, USA, volume 630 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992.

[CLX95] D. Clarke, I. Lee, and H. Xie. Versa : A tool for the specification and analysis of resource-bound real-time systems. *Journal of Computer and Software Engineering*, 3(2), April 1995.

[CO95] Jean-Pierre Courtiat and Roberto C. De Oliveira. On rt-lotos and its application to the formal design of multimedia protocols. *Annals of Telecommunications*, 50(11-12) :888–906, nov 1995.

[CO00] Rachel Cardell-Oliver. Conformance testing of real-time systems with timed automata specifications. *Formal Aspects of Computing Journal*, 12(5) :350–371, 2000.

[CO02] Rachel Cardell-Oliver. Conformance test experiments for distributed real-time systems. In *International Symposium on Software Testing and Analysis (ISSTA ?02), ACM Press*, July 2002.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre : A declarative language for programming synchronous systems. In *Symposium on Principles of Programming Languages,POPL 1987 : Munich, Germany*, pages 178–188, 1987.

[CPK00] Shelton Charles, Koopman Philip, and DeVale Kobey. Robustness testing of the microsoft win32 api. In *International Conference on Dependable Systems and Networks*, New York City, 2000.

[CSLO00] J.-P. Courtiat, C.A.S. Santos, C. Lohr, and B. Outtaj. Experience with RT-LOTOS, a Temporal Extension of the LOTOS Formal Description Technique. *Computer Communications*, 23(12) :1104–1123, 2000.

[CVI90] W. Chan, S. Vuong, and R. Ilto. An improved protocol test generation procedure based on uios. In Brinksma et al. [BSV90].

[CW02] Richard Castanet and Hélène Waeselynck. Les enjeux du test de robustesse : résultats de l'as 23 du cnrs. In *Journées du Réseau Thématique Prioritaire SECC*, November 2002.

[CYSY01] Jin-Young Choi, Hee Yong Youn, Soonuk Seol, and Chuck Yoo. Distributed test using logical clock. In *Formal Techniques for Networked and Distributed Systems, FORTE 2001, IFIP TC6/WG6.1 - 21st International Conference on Formal Techniques for Networked and Distributed Systems, Cheju Island, Korea*, pages 69–84, August 2001.

[CZ91] R. Cleaveland and A.E. Zwarico. A theory of testing for real-time. In LICS'91 [LIC91], pages 110–119.

[D'A97] Pedro R. D'Argenio. Regular processes and timed automata. In *ransformation-Based Reactive Systems Development, 4th International AMAST Workshop on Real-Time Systems and Concurrent and Distributed Software, ARTS'97, Palma, Mallorca, Spain, Proceedings*, pages 141–155, May 1997.

[Dav] Alexandre David. *Uppaal2k : Small Tutorial.* Aalborg University, Denmark, http ://www.uppaal.com/. http ://www.it.uu.se/research/group/darts/uppaal/documentation.html.

[DBALS97] D.Clarke, H. Ben-Abdallah, I. Lee, and O. Sokolsky. Paragon : A paradigm for the specification, verification and testing of real-time systems. In *1997 IEEE Aerospace Conference, IEEE Computer Society Press*, February 1997.

[DCL95] James Vera David C. Luckham. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9) :717–734, September 1995.

[DFYQO01] Cavaliere D., Simonot-Lion F., Song Y.-Q., and Hembert O. A component model approach for modelling and validation of an automated manufacturing system. In *Proceedings of 8th IEEE International Conference on Emerging Technologies and Factory Automation, Antibes - Juan Les Pins*, 2001.

[Dil90] D. Dill. Timing assumptions and verification of finite-state concurrent systems. In Sifakis [Sif90], pages 197–212.

[Dil94] D. Dill, editor. *Proceedings of the 6th International Conference on Computer Aided Verification,* Stanford, USA, volume 818 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[D.K93] Clark J.A.and Pradhan D.K. React : a synthesis and evaluation tool for fault-tolerant multiprocessor architectures. In *Reliability and Maintainability Symposium*, pages 428–435, January 1993.

[DK99] A. EnNouaary R. Dssouli and F. Khendek. Fault Coverage in Testing Real-Time Systems. In *RTCSA'99*, 1999.

[DKPD99] DeVale, J. Koopman, P., and Guttendorf D. The ballista software robustness testing service. In *Testing Computer Software Conference (TCSC99)*, June 1999.

[EDE97] A. EnNouaary, R. Dssouli, and A. Elqortobi. Génération de tests temporisés. In *Proceedings of the* 6^{th} *bi-Annual Colloque Francophone de l'ingénierie des Protocoles,* Lièges, Belgique, 1997.

[EDK02] Abdeslam EnNouaary, Rachida Dssouli, and Ferhat Khendek. Timed wp-method : Testing real-time systems. *IEEE Transactions on Software Engineering (TSE)*, 28(11) :1023–1038, 2002.

[EDKE98] A. EnNouaary, R. Dssouli, F. Khendek, and A. Elqortobi. Timed test cases generation based on state characterization technique. In RTSS98 [RTS98].

[EKD00] A. EnNouaary, F. Khendek, and R. Dssouli. Testing embedded real-time systems. In *Proceedings of the seventh international conference on Real-Time Computing Systems and Applications (RTCSA'2000), Cheju Island, South Korea*, pages 417–423, December 2000.

[FBK+91] S. Fujiwara, G. Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite-state models. *IEEE Transactions on Software Engineering*, 17(6) :591–603, June 1991.

[Fer89] Jean-Claude Fernandez. *ALDEBARAN : User's Manual.* Laboratoire de Génie Informatique — Institut IMAG, Grenoble, January 1989.

[FGMT02] Loe M. G. Feijs, Nicolae Goga, Sjouke Mauw, and Jan Tretmans. Test selection, trace distance and heuristics. In *Testing of Communicating Systems XIV, Applications to Internet Technologies and Services, Proceedings of the IFIP 14th International Conference on Testing Communicating Systems - TestCom 2002, Berlin, Germany*, pages 267–282, March 2002.

[FH02] H. Fouchal and T. Higashino. Test case generation algorithm for timed automata (technical report). Technical report, Université de Reims Champagne Ardenne, 2002.

[FHvLP00] J. Fischer, E. Holz, M. v. Lowis, and A. Prinz. Sdl-2000 : A language with a formal semantics. In *3rd Workshop on Rigorous Object-Oriented Methods, University of York, UK*, January 2000.

[FJJV96] J.C. Fernandez, C. Jard, T. Jeron, and G. Viho. Using on-the-fly verification techniques for the generation of test suites. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification (CAV)*, volume 1102, pages 348–359. Springer, 1996.

[FK99] K. Fernsler and P. Koopman. Robustness testing of a distributed simulation backplane. In *ISSRE 99, Boca Raton, Florida*, November 1999.

[FP01] Hacène Fouchal and Eric Petitjean. Fault Detection on Timed Systems. *Studia Informatica Universalis*, 4(2), 2001.

[FPG+05] H. Fouchal, L. Pierre, S. Gruson, C. Rabat, and A. Rollet. Integrated tool for testing timed systems. In *Proceedings of 5th IEEE International Symposium and School on Advance Distributed Systems (ISSADS 2005), Guadalajara, Jalisco, Mexico*, Lecture Notes in Computer Science. Springer, January 2005.

[FPS00] Hacène Fouchal, Eric Petitjean, and Sébastien Salva. Testing Timed Systems with Timed Purposes. In *Proceedings of the 7th International Conference on Real-Time Computing Systems and Applications, RTCSA'00 (Cheju Island, South Korea), IEEE Computer Society*, pages 166–171, December 2000.

[FR03] H. Fouchal and A. Rollet. Embedded system testing. In *Proceedings of 7th International Conference on Principles of Distributed Systems (OPODIS 2003), La Martinique, France*, volume 3144 of *Lecture Notes in Computer Science*, pages 159–170. Springer-Verlag, December 2003.

[FRRT04] H. Fouchal, C. Rabat, A. Rollet, and A. Tarhini. Experimental results on testing real-time systems. In *13th International Conference on Intelligent and Adaptive Systems and Software Engineering (IASSE-2004), Nice, France*, pages 284–289, July 2004.

[FRT05] H. Fouchal, A. Rollet, and A. Tarhini. Robustness of composed timed systems. In *31st Annual Conference on Current Trends in Theory and Practice of Informatics (SOFSEM05), Liptovsky Jan, Slovak Republic, Europe*, lncs. Springer, January 2005.

[FSLM02] Jean-Philippe Fassino, Jean-Bernard Stefani, Julia L. Lawall, and Gilles Muller. Think : A software framework for component-based operating system kernels. In *USENIX Annual Technical Conference, General Track*, pages 73–86, 2002.

[FUDA00] M.A. Fecko, M.Ü. Uyar, A.Y. Duale, and P.D. Amer. Efficient test generation for army network protocols with conflicting timers. In *MILCOM 2000 - IEEE Military Communications Conference*, pages 133–138, October 2000.

[GAO95] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch : Why reuse is so hard. *IEEE software*, November 1995.

[Gar90] Hubert Garavel. Introduction au langage LOTOS. *Revue de l'Association des Anciens Elèves de l'ENSIMAG*, 1990.

[GHN93] J. Grabowski, D. Hogrefe, and R. Nahm. Test case generation with test purpose specification by mscs, 1993.

[GI90] K. K. Goswami and R. K. Iyer. A design environment for prediction and evaluation of system dependability. In *9th Digital Avionics Systems Conference*, pages 87–92, October 90.

[Gil62] A. Gill. *Introduction to the theory of finite-state machines*. Mc Graw-Hill, New York – USA, 1962.

[GJ95] L. Gallon and G. Juanole. Critical time distributed systems : Qualitative and quantitative analysis based on stochastic petri nets. In *8th International Conference on Formal Description Techniques (FORTE'95), Montreal, Canada*, October 1995.

[GJP95] A.K. Ghosh, B.W. Johnson, and J.A. Iii Profeta. System-level modeling in the adept environment of a distributed computer system for real-time applications. In *International Computer Performance and Dependability Symposium (IPDS'95),Erlangen, Germany*, pages 194–203, April 1995.

[GKT89] U. Gunneflo, J. Karlsson, and J. Torin. Evaluation of error detection schemes using fault injection by heavy-ion radiation. In *19th International Symposium on Fault-Tolerant Computing (FTCS-19)*, pages 340–347, june 1989.

[Gla90] R.J. van Glabbeek. The linear time – branching time spectrum (extended abstract). In J.C.M. Baeten and J.W. Klop, editors, *CONCUR '90, Theories of Concurrency : Unification and Extension, Amsterdam, August*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297. Springer-Verlag, 1990.

[Gom01] Hassan Gomaa. Designing concurrent, distributed, and real-time applications with uml. In *23rd International Conference on Software Engineering, ICSE 2001, Toronto, Ontario, Canada*, pages 737–738, May 2001.

[Gon70] G. Gonenc. A method for the design of fault detection experiment. *IEEE transactions on Computers*, C-19 :551–558, 1970.

[GPR04] S. Gruson, L. Pierre, and C. Rabat. Générateur de séquences de test. Master's thesis, Université de Reims Champagne Ardenne, 2004.

[Gri99] John Linwood Griffin. Testing protocol implementation robustness. In *29th Annual International Symposium on Fault-Tolerant Computing, Madison, Wisconsin*, June 1999.

[GS95] J. Guthoff and V. Sieh. Combining software-implemented and simulation-based injection into a single fault injection method. In *25th International Symposium on Fault-Tolerant Computing (FTCS-25)*, pages 196–206, June 1995.

[Hab01] H. Habrias. *Spécification formelle avec B*. Lavoisier, 2001.

[HHWT95] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. A user guide to hytech. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 41–71, 1995.

[HKPM97] G. Huet, G. Kahn, and C. Paulin-Mohring. The coq proof assistant - a tutorial, version 6.1. Technical Report 204, INRIA, 1997.

[HNSY92] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. In LICS'92 [LIC92], pages 394–406.

[HNTC99] T. Higashino, A. Nakata, K. Taniguchi, and A. Cavalli. Generating test cases for a timed i/o automaton model. In *IFIP 12th International Workshop on Testing of Communicating Systems (IWTCS'99), Budapest Hungary*, September 1999.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8) :666–677, 1978.

[HP85] D. Harel and A. Pnueli. On the development of reactive systems in logic and models of concurrent systems. *NATO ASI Series*, 13 :477–498, 1985.

[IEE90] IEEE. *IEEE Standard Glossary of Software Engineering Terminology, IEEEE Std 610.12-1990*, December 1990.

[ISO86] ISO. *Estelle : a formal description technique based on an extended state transition model* ISO/TC97/SC21/WG16–1 DP9074, 1986. ISO/TC97/SC21/WG16–1 DP9074.

[ISO87] ISO. *Information processing systems – open systems interconnection – LOTOS – a formal description technique based on the temporal ordering of observational behaviour* ISO/TC97/SC21/N DIS8807, 1987.

[ISO91] ISO. Conformance Testing Methodology and Framework. International Standard 9646, International Organization for Standardization — Information Technology — Open Systems Interconnection, Genève, 1991.

[ITSJ00] Hwang I., Kim T., Hong S., and Lee J. Test selection for a nondeterministic fsm. *Computer Communications*, 24 :1213–1223, August 2000.

[JAR+94] Eric Jenn, Jean Arlat, Marcus Rimen, Joakim Ohlsson, and Johan Karlsson. Fault injection into VHDL models : The MEFISTO tool. In *Proceedings of the 24th International Symposium on Fault Tolerant Computing, (FTCS-24), IEEE, Austin, Texas, USA*, pages 66–75, 1994.

[Jar03] M.T. Jarboui. *Sûreté de fonctionnement de systèmes informatiques. Etalonnage et représentativité des fautes.* PhD thesis, LAAS, May 2003.

[JF02] Elloy J.P-. and Simonot-Lion F. An architecture description language for in-vehicle embedded system development. In *Proceedings of IFAC 15th World Congress B'02*, 2002.

[JGH00] Colin Willcock Jens Grabowski, Anthony Wiles and Dieter Hogrefe. On the design of the new testing language ttcn-3. In *Testing of Communicating Systems : Tools and Techniques, IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems (TestCom 2000), Ottawa, Canada*, pages 161–176, September 2000.

[JP96] B. Jonsson and J. Parrow, editors. *Proceedings of the 4th International School and Symposium on Formal Techniques in Real Time and Fault Tolerant Systems,* Uppsala, Sweden, volume 1135 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[Jr98] Luiz Augusto De Paula Lima Jr. *A Pragmatic Method to Generate Test Sequences for Embedded Systems.* PhD thesis, Université d'Evry - Val d'Essonne, Institut National des Télécommunications, Evry, France, 1998.

[JSD+97] Philip J. Koopman Jr., John Sung, Christopher P. Dingman, Daniel P. Siewiorek, and Ted Marz. Comparing operating systems using robustness benchmarks. In *Symposium on Reliable Distributed Systems*, pages 72–79, 1997.

[Jéz03] Jean-Marc Jézéquel. Projet triskell : construction fiable et efficace d'applications par assemblage de composants logiciels, 2003. IRISA.

[Ka94] J. Karlsson and al. Using heavy-ion radiation to validate fault-handling mechanisms. *IEEE Micro*, 14(1) :25–40, February 1994.

[Kaa99] M. Kaaniche. *Evaluation de la sûreté de fonctionnement informatique. Fautes physiques, fautes de conception, malveillances.* Habilitation à Diriger des Recherches en Informatique, LAAS, 1999.

[KAD+00] Ahmed Khoumsi, Mehdi Akalay, Rachida Dssouli, Abdeslam EnNouaary, and Louis Granger. An approach for testing real time protocol entities. In *Testing of Communicating Systems : Tools and Techniques, IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems (TestCom 2000), Ottawa, Canada*, pages 281–300, September 2000.

[KEDA00] A. Khoumsi, A. EnNouaary, R. Dssouli, and M. Akalay. A new method for testing real time systems. In *Proceedings of the seventh international conference on Real-Time Computing Systems and Applications (RTCSA'2000), Cheju Island, South Korea*, pages 441–450, December 2000.

[KFA+95] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, and G. Leber. Integration and comparison of three physical fault injection techniques. In *Predictably Dependable Computing Systems, chapter V : Fault Injection*, pages 309–329. Springer Verlag, 1995.

[KGJ95] N.A. Kanawati, G.Kanawati, and J.Abraham. Dependability evaluation using hybrid fault/error injection. In *International Computer Performance and Dependability Symposium (IPDS'95), Erlangen, Germany*, April 1995.

[KGLY99] Carsten Weise Kim Guldstrand Larsen, Justin Pearson and Wang Yi. Clock difference diagrams. *Nordic Journal of Computing*, 6(3), 1999.

[Kho00] A. Khoumsi. Timing issues in testing distributed systems. In *4th IASTED International Conference on Software Engineering and Applications, SEA2000*, November 2000.

[Kho01a] A. Khoumsi. Testing distributed real-time reactive systems using a centralized test architecture. In *North Atlantic Test Workshop (NATW), Gloucester, Massachusetts, USA*, May 2001.

[Kho01b] Ahmed Khoumsi. Testing distributed real-time reactive systems using a centralized test architecture. In *IEEE North Atlantic Test Workshop (NATW), Gloucester, Massachusetts, USA*, May 2001.

[Kho02] Ahmed Khoumsi. A method for testing the conformance of real time systems. In *Formal Techniques in Real-Time and Fault-Tolerant Systems, 7th International Symposium, FTRTFT 2002, Co-sponsored by IFIP WG 2.2, Oldenburg, Germany*, pages 331–354, September 2002.

[Kho03] A. Khoumsi. Testing distributed real-time systems in the presence of inaccurate clock synchronization. *Journal of Information Soft. Technology (IST)*, 45, December 2003.

[KJS98] Nathan P. Kropp, Philip J. Koopman Jr., and Daniel P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Symposium on Fault-Tolerant Computing*, pages 230–239, 1998.

[KKA92] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham. Ferrari : A tool for the validation of system dependability properties. In *22nd International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 336–344, July 1992.

[KKA95] G.A. Kanawati, N.A. Kanawati, and J.A. Abraham. Ferrari : a flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 44(2) :248–60, 1995.

[KLK96] Inhye Kang, Insup Lee, and Young Si Kim. A Efficient State Space Generation for The Analysis of Real-Time Systems. In *Proceedings of International Symposium on SOoftware Testing and Analysis*, February 1996.

[Koe] D. Koenig. Surveillance : Approche sûreté de fonctionnment (cours inpg : école esisar).

[Kon01] Ousmane Koné. A local approach to the testing of real-time systems. *The Computer Journal*, 44(5) :435–447, 2001.

[Kon02] Ousmane Koné. Compliance of wireless application protocols. In *Testing of Communicating Systems XIV, Applications to Internet Technologies and Services, Proceedings of the IFIP 14th International Conference on Testing Communicating Systems - TestCom 2002, Berlin, Germany*, pages 129–138, March 2002.

[KS01] J.M. Küster and J. Stroop. Consistent design of embedded real-time systems with uml-rt. In *Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, page 31. IEEE Computer Society, 2001.

[LAP97] Roberto C. de Oliveira L. Andriantsiferana, Jean-Pierre Courtiat and L. Picci. An experiment in using rt-lotos for the formal specification and verification of a distributed scheduling algorithm in a nuclear power plant monitoring system. In *Formal Description Techniques and Protocol Specification, Testing and Verification, FORTE X / PSTV XVII'97, IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE X) and Protocol Specification, Testing and Verification (PSTV XVII), Osaka, Japan*, pages 433–448, November 1997.

[Lau99] P. Laurençot. *Integration du temps dans les tests de protocoles de communication.* PhD thesis, Univ. of Bordeaux 1, January 1999.

[LC97a] Patrice Laurencot and Richard Castanet. Integration of Time in Canonical Testers for Real-Time Systems. In *International Workshop on Object-Oriented Real-Time Dependable Systems, California.* IEEE Computer Society Press, 1997.

[LC97b] Luiz Paula Lima and Ana R. Cavalli. A pragmatic approach to generating test sequences for embedded systems. In *Proceeding of IWTCS'97/ 10th internationa IFIP TC6 /WG6.1 Workshop on Testing of Communication Systems, Cheju Island, (Corée)*, September 1997.

[LH01] D. Lee and Ruibing Ha. Test sequence selection. In *Formal Techniques for Networked and Distributed Systems, FORTE 2001, IFIP TC6/WG6.1 - 21st International Conference on Formal Techniques for Networked and Distributed Systems, Cheju Island, Korea*, pages 269–284, August 2001.

[LIC90] *Proceedings 5th Annual Symposium on Logic in Computer Science,* Philadelphia, USA. IEEE Computer Society Press, 1990.

[LIC91] *Proceedings 6th Annual Symposium on Logic in Computer Science,* Amsterdam. IEEE Computer Society Press, 1991.

[LIC92] *Proceedings 7th Annual Symposium on Logic in Computer Science,* Santa Cruz, California. IEEE Computer Society Press, 1992.

[Lio96] J-L Lions. Ariane 5 - flight 501 failure. Technical report, CNES, http ://ravel.esrin.esa.it/docs/esa-x-1819eng.pdf, 1996.

[LL97] L. Leonard and G. Leduc. An introduction to et-lotos for the description of time-sensitive systems. *Computer Networks and ISDN Systems*, 29 :271, August 1997.

[LLPY97] Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. Efficient verification of real-time systems : compact data structure and state-space reduction. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS '97), San Francisco, CA, USA*, pages 14–24, December 1997.

[Loe99] V. Loechner. Polylib : A library for manipulating parameterized polyhedra, 1999.

[LPY95] K.G. Larsen, P. Pettersson, and W. Yi. Model-checking for real-time systems. In *Proceedings of Fundamentals of Computation Theory*, pages – , Dresden, Germany, August 1995.

[LPY97] K.G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1-2) :134–152, 1997.

[LT93] N. G. Leveson and C. S. Turner. Investigation of the therac-25 accidents. *IEEE Computer*, 26(7) :18–41, 1993.

[LU92] R.J. Linn and M.U. Uyar, editors. *Protocol Specification, Testing, and Verification, XII,* Lake Buena Vista, Florida, USA. North-Holland, June 1992.

[LV92] N.A. Lynch and F.W. Vaandrager. Forward and backward simulations for timing-based systems. In Bakker et al. [BHRR92], pages 397–446.

[Mam01] Z. Mammeri. Introduction au langage de description et de spécification (sdl). Technical report, Université Paul Sabatier - Toulouse, 2001.

[MD02] Prabhat Mishra and Nikil Dutt. Architecture description language driven functional test program generation for microprocessors using smv. Technical report, University of California, 2002.

[MF76] P. Merlin and D. J. Faber. Recoverability of communication protocols. *IEEE Trans Comm*, 1976.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, 1980.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice-Hall International, 1989.

[MTT+01] Takanori Mori, Kohei Tokuda, Harumasa Tada, Masahiro Higuchi, and Teruo Hogashino. A Method to Generate Conformance Test Sequences

for FSM with Timer System Call. In *Proceedings of 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed Systems*, August 2001.

[MY01] T. Margaria and W. Yi, editors. *Proceedings of the Workshop on Tools and Algorithms for the Construction and Analysis of Systems,* Genova, Italy, volume 2031 of *Lecture Notes in Computer Science*. Springer-Verlag, April 2001.

[NAS97] NASA. Mars pathfinder mission status. Technical report, Jet Propulsion Laboratory, http ://mars.jpl.nasa.gov/MPF/mpf/status/statuslist.html, July 1997.

[NH84] R. De Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34 :83–133, August 1984.

[NH01] Akio Nakata and Teruo Hogashino. Deriving Parameter Conditions for Periodic Timed Automata Satisfaying Real-Time Temporal Logic Formulas. In *Proceedings of 21st IFIP WG 6.1 International Conference on Formal Techniques for Networked and Distributed System*, November 2001.

[Nie00] Brian Nielsen. *Specification and Test of Real-Time Systems*. PhD thesis, Department of Computer Science, Aalborg University, 2000.

[NS01] Brian Nielsen and Arne Skou. Automated Test Generation from Timed Automata. In Margaria and Yi [MY01], pages 343–357.

[NT81] S. Naito and M. Tsunoyama. Fault Detection for Sequential Machines by Transition- Tours. *Proceedings of Fault Tolerant Computer Systems*, pages 238–243, 1981.

[OMG03a] OMG. *UML 2.0 Infrastructure Specification, OMG Adopted Specification*, September 2003.

[OMG03b] OMG. *UML 2.0 Superstructure Specification, OMG Adopted Specification*, August 2003.

[OP94] F. Ouabdesselam and I. Parissis. Testing synchronous critical software. In *5th Int'l Symposium on Software Reliability Engineering, Monterey, USA*, November 1994.

[Ost90] J. Ostroff. Deciding properties of timed transitions models. *IEEE Transactions on Parallel Systems*, 1 :170–183, 1990.

[Ost92] J. S. Ostroff. Formal methods for the specification and design of real-time safety critical systems. *Systems and Software*, 1992.

[OW90] J. Ostroff and W. Wonham. A framework for real-time discret event control. *IEEE Transactions on Automatic Control*, 4 :386–397, 1990.

[Par04] Stéphane Parpinelli. Spirit, le robot martien, sauvé par ń la présence d'esprit ż de ses batteries. Press article : 01 Informatique no 1757, Feb 2004. URL : http ://www.01net.com/article/233505.html.

[Pel02] Jan Peleska. An approach for testing real time protocol entities. In *Testing of Communicating Systems XIV, Applications to Internet Technologies and Services, Proceedings of the IFIP 14th International Conference on*

Testing Communicating Systems - TestCom 2002, Berlin, Germany, pages 335–355, March 2002.

[Pet62] C.A. Petri. Kommunikation mit Automaten. Schriften des IIM Nr. 2, Institut für Instrumentelle Mathematik, Bonn, 1962.

[Pet00] Eric Petitjean. *Etude des méthodes de test pour les systèmes temporisés.* PhD thesis, Doctorat 3eme cycle en Informatique, Université de Reims Champagne-Ardenne, 2000.

[Pha94] M. Phalippou. *Relation d'implantation et hypothèses de test sur des automates à entrées et sorties.* PhD thesis, Univ. of Bordeaux, September 1994.

[Phi03] S. Philippi. Analysis of fault tolerance and reliability in distributed real-time system architectures. *Reliability Engineering and System Safety*, 82, 2003.

[PKS+01] Jiantao Pan, Philip Koopman, Daniel P. Siewiorek, Yennun Huang, Robert Gruber, and Mimi Ling Jiang. Robustness testing and hardening of corba orb implementations. In *2001 International Conference on Dependable Systems and Networks (DSN 2001) (FTCS), Göteborg, Sweden.* IEEE Computer Society, July 2001.

[PPD99] J. Pan, Koopman P., and Siewiorek D. A dimensionality model approach to testing and improving software robustness. In *Autotestcon99, San Antonio, TX.*, August 1999.

[PRBP94] A. Pnueli, W.P. Roever, G. Berry, and A. Poigné. Synchronous languages. In *Report on the Dagstuhl Seminar 9448 on Synchronous Languages : Schloss Dagsthul International Conference And Research Center For Computer Science, Germany*, November 1994.

[PU00] Alexandre Petrenko and Andreas Ulrich. Verification and testing of concurrent systems with action races. In *Testing of Communicating Systems : Tools and Techniques, IFIP TC6/WG6.1 13th International Conference on Testing Communicating Systems (TestCom 2000), Ottawa, Canada*, pages 261–280, September 2000.

[Ram74] C. Ramchandani. *Analysis of Asynchronous Concurrent Systems by Timed Petri Nets.* PhD thesis, MIT, 1974.

[RF03] A. Rollet and H. Fouchal. Testing protocol robustness. In *Proceedings of Innovative Internet Community Systems (I2CS'03), Leipzig Germany*, volume 2877 of *Lecture Notes in Computer Science*, pages 201–215. Springer-Verlag, June 2003.

[Rif01] J. M. Rifflet. Tolérance aux pannes. Technical report, Université Paris VII, http ://www.pps.jussieu.fr/ rifflet/tolerance1.html, 2001.

[RNHW98] P. Raymond, X. Nicollin, N. Halbwatchs, and D. Waber. Automatic testing of reactive systems, madrid, spain. In *Proceedings of the 1998 IEEE Real-Time Systems Symposium, RTSS'98*, pages 200–209. IEEE Computer Society Press, December 1998.

[Rol03] A. Rollet. Testing robustness of real-time embedded systems. In *Proceedings of Workshop On Testing Real-Time and Embedded Systems (WTRTES), Satellite Workshop of FM 2003 Symposium, Pisa, Italy*, September 2003.

[RSSL02] X. Rebeuf, G. Satriano, and F. Simonot-Lion. A distributed algorithm for the validation of timed state machines. In *Proceedings of International Conference On Principles Of DIstributed Systems (OPODIS'02), Reims*, December 2002.

[RTL] site officiel de rt-lotos : http ://www.laas.fr/rt-lotos /distrib/dta2kronos.0.7.tar.gz.

[RTS98] *19th IEEE Real Time Systems Symposium (RTSS'98)* Madrid, Spain, 1998.

[Rus01] J.M. Rushby. Theorem proving for verification. In *Modelling and Verification of Parallel Processes (MOVEP2K)*, volume 2067 of *lncs*, pages 39–57. Springer, 2001.

[Sal01] Sébastien Salva. *Etude de la qualité de test des systèmes temporisés.* PhD thesis, Doctorat 3eme cycle en Informatique, Université de Reims Champagne-Ardenne, 2001.

[Sch99] P. Schnoebelen. *Vérification de logiciels – Techniques et outils du model-checking.* Eyrolles, 1999.

[Sch03] H. Schmidt. Trustworthy components-compositionality and prediction. *The Journal of Systems and Software*, 65 :215–225, 2003.

[SD88] K. Sabnani and A. Dahbura. A protocol test generation procedure. *Computer Networks and ISDN Systems*, 15 :285–297, 1988.

[Sel98] B. Selic. Using uml for modeling complex real-time systems. In *Proceedings of the ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems*, pages 250–260. Springer-Verlag, 1998.

[Sif90] J. Sifakis, editor. *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems,* Grenoble, France, volume 407 of *Lecture Notes in Computer Science.* Springer-Verlag, 1990.

[SMF98] J. R. Samson, W. Moreno, and F. Falquez. A technique for automated validation of fault tolerant designs using laser fault injection (lfi). In *28th International Symp. on Fault-Tolerant Computing (FTCS-28)*, pages 162–167, June 1998.

[SRF02] Sébastien Salva, Antoine Rollet, and Hacène Fouchal. Temporal and Behavior Characterizatrion of States in Timed Systems. In *ACIS Annnual International Conference on Computer and Information Science, ICIS'02, Seoul, South Korea*, 2002.

[SV96] J. Springintveld and F. Vaandrager. Minimizable timed automata. In Jonsson and Parrow [JP96], pages –.

[SVD97] J. Springintveld, F.W. Vaandrager, and P. R. D'Argenio. Timed Testing Automata. Report CS-R9712, CWI, Amsterdam, August 1997.

[SVD01] J. Springintveld, F.W. Vaandrager, and P. R. D'Argenio. Timed Testing Automata. *Theoretical Computer Science*, 254(254) :225–257, 2001.

[Szy98] C. Szyperski. *Component Software : Beyond Object Oriented Programming*. Addison Wesley, Harlow, England, 1998.

[TB99] J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99 : 7^{th} European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999. EuroStar Conferences, Galway, Ireland.

[TIJ96] Timothy K. Tsai, Ravishankar K. Iyer, and Doug Jewitt. An approach towards benchmarking of fault-tolerant commercial systems. In *Symposium on Fault-Tolerant Computing*, pages 314–323, 1996.

[TNHN03] A. Tesanovic, D. Nystrom, J. Hansson, and C. Norstrom. Towards aspectual component-based development of real-time systems. In *Proceeding of the 9th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2003)*, February 2003.

[Tre96a] J. Tretmans. Test generation with inputs, outputs, and repetitive quiescence. *Software–Concepts and Tools*, 17 :103–120, 1996.

[Tre96b] J. Tretmans. Test generation with inputs, outputs and repetitive quiescence. *Software - Concepts and Tools*, 17, 1996.

[Tre97] J. Tretmans. Repetitive Quiescence in Implementation and Testing (Extended Abstract). In A. Wolisz, I. Schieferdecker, and A. Rennoch, editors, *Formale Beschreibungstechniken für verteilte Systeme*, number Nr. 315 in GMD-Studien, pages 23–37, St. Augustin, 1997. GI/ITG-Fachgespräch, GMD.

[TRF05] A. Tarhini, A. Rollet, and H. Fouchal. A pragmatic approach for testing robustness on real-time component based systems. In *3rd ACS/IEEE International Conference on Computer Systems and Applications (AICCSA'05), Cairo, Egypt*, January 2005.

[TS96] S. Tripakis and S.Yovine. Analysis of timed systems based on time-abstracting bisimulations. In CAV'96 [CAV96].

[Tur93] K.J. Turner, editor. *Using Formal Description Techniques - An Introduction to Estelle, LOTOS and SDL*. John Wiley and Sons Ltd., 1993.

[UD99] M.Ü. Uyar and A.Y. Dual. Resolving inconsistencies in efsm-modeled specifications. In *MILCOM 1999 - IEEE Military Communications Conference*, pages 135–139, October 1999.

[UTY+02] Takaaki Umedu, Yoshiki Terashima, Keiichi Yasumoto, Akio Nakata, Teruo Higashino, and Kenichi Taniguchi. A language for describing wireless mobile applications with dynamic establishment of multi-way synchronization channels. In *FME 2002 : Formal Methods - Getting IT Right, International Symposium of Formal Methods Europe, Copenhagen, Denmark*, pages 607–624, July 2002.

[VCI89] S. Vuong, W. Chan, and M. Ito. The UIOv-Method for Protocol Test Sequence Generation. In *2nd IWPTS International Workshop on Protocol Test Systems, Berlin*, 1989.

[Ver99] Verilog SA. *TestComposer Reference Manual, 4.1 edition*, 1999.

[vO02] Rob van Ommering. *Building Reliable Component-Based Software Systems*. Artech House Publishers, July 2002.

[vOvdLKM00] Rob van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The koala component model for consumer electronics software. *Computer*, 33(3) :78–85, 2000.

[Wan01] Farn Wang. Symbolic verification of complex real-time systems with clock-restriction diagram. In *Formal Techniques for Networked and Distributed Systems, FORTE 2001, IFIP TC6/WG6.1 - 21st International Conference on Formal Techniques for Networked and Distributed Systems, Cheju Island, Korea*, pages 235–250, August 2001.

[YAIG93] L. Young, C. Alonso, R. Iyer, and K. Goswami. A hybrid monitor assisted fault injection environment. *Dependable Computing for Critical Applications*, 3 :281–302, 1993.

[Yov93] S. Yovine. *Méthodes et outils pour la vérification symbolique de systèmes temporisés*. PhD thesis, Institut National Polytechnique de Grenoble, France, May 1993.

[Yov97] Sergio Yovine. Kronos : A verification tool for real-time systems. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1-2) :123–133, dec 1997.

[Z1299] Itu telecommunication standards sector sg 10. itu-t recommandation z.120 : Message sequence chart (msc). itu, geneva, 1999.

[Zal01] J. Zalewski. Developing component-based software for real-time systems. In *27th Euromicro Conference 2001 : A Net Odyssey (euromicro'01)*, September 2001.

[Zim99] Peter Zimmerer. Test architectures for testing distributed systems. In *12th International software quality week (QW'99)*, May 1999.

[ZRDN02] Jens Grabowski Zhen Ru Dai and Helmut Neukirchen. Timed ttcn-3 - a real-time extension for ttcn-3. In *Testing of Communicating Systems XIV, Applications to Internet Technologies and Services, Proceedings of the IFIP 14th International Conference on Testing Communicating Systems - TestCom 2002, Berlin, Germany*, pages 407–424, March 2002.

Annexe A

Anomalies répertoriées dans le domaine spatial en 1999

Voici une liste non officielle des défaillances dans le domaine spatial courant 1999. Elle est tirée de l'URL suivante : *http ://eurespace.online.fr/anomalie/anomali9.htm*

A.1 Lanceurs

Matériel	Date	Compagnie	Famille	Nom	Charge utile	Défaillance
1999-01 Lanceur	Avril 1999	Boeing	Delta	Delta III	?	4 tentatives de décollage avortées.
Lanceur	Avril 1999	Boeing	Titan	Titan IV B	Mil US - DSP 19	Séparation des 2 éléments du dernier étage IUS non effectuée. Satellite sur mauvaise orbite. Récupéré mais inexploitable.
Lanceur	Avril 1999	Lockeed Martin	Athena	Athena II	Ikonos 1 (imageur)	Vitesse d'injection insuffisante. Défaut de commande pyrotechnique du largage de la coiffe. Satellite évanoui dans l'espace.
Lanceur	30 Avril 1999	Lockeed Martin	Titan	Titan IV B	Milstar II F3	Système de pilotage du dernier étage Centaur défaillant. Poussée de compensation. Consommation de carburant trop importante. Erreur humaine dans le chargement du logiciel. Satellite à 5000 Km au lieu de 36000 Km. Satellite mis en sommeil et perdu.
Lanceur	Mai 1999	Boeing	Delta	Delta III	Loral - Orion 3	Tir d'essai. Procédures erronées. Satellite sur orbite irrécupérable. Périgée à 1400 Km au lieu de 26000 Km.
1999-6 Lanceur	Juil. 1999	Khrunichev / Energiya	Proton	Proton K	Raduga	Défaut de fonctionnement du 2 ème étage et explosion en vol du 3 ème étage. Equippé du nouveau 4 ème étage Breeze M destiné à remplacer l'ancien, peu fiable. Perte du satellite. Remise en cause par le Kazakstan de l'utilisation de la base de lancement par la Russie.
Lanceur	Nov. 1999	Khrunichev / Energiya	Proton	Proton K	Express 1A	Explosion à faible altitude (2ème étage) Perte du satellite. Interdiction par le Kazakstan d'utiliser la base de lancement et donc remise en cause de tous les tirs à venir depuis Baïkonour.
Lanceur	15 Nov. 1999	Japon	H2	H2	Japan MOT-MTSat	Destruction volontaire du lanceur en vol du fait de l'arrêt des moteurs de propulsion. Plusieurs incidents satellite et lanceurs avaient repoussé le lancement de 3 mois. Perte du satellite (observation et navigation aviation civile) de 100 millions d'Euros.

A.2 Satellites

Matériel	Date	Compagnie	Famille	Nom	Charge utile	Défaillance
1999-01 Satellite	Fév. 1999	Boeing SS	Boeing 601 HP	Boeing - HGS 1	NA	Perte de capacité des batteries. Cellule. On se rappelera sa mise à poste passant par une orbite lunaire.
Satellite	Avril 1999	Lockeed Martin	A2100	Télésat - Nimiq	NA	Rapatrié du centre de lancement. Défaut d'interconnexion des émetteurs à tube.
Satellite	Avril 1999	Boeing SS	Boeing 601	SatMex - Solidaridad 1	NA	Coupure intempestive des émissions et perte de contrôle du satellite. Le satellite est déjà sur le calculateur de bord de secours. Remise en fonctionement quelques heures plus tard. Désorganisation complète des transmissions de données.
Satellite	Avril 1999	Alcatel Space Aérospatiale E&D	SB2000	Eutelsat II F4	NA	Interruption de service au cours de manjuvre de routine.
1999-05 Satellite	Mai 1999	Alcatel Space Aérosatiale E&D	SB2000	Nahuelsat - Nahuel 1	NA	Coupure intempestive des émissions
Satellite	Juin 1999	-	-	Wire	NA	Défaillance d'un composant commercial. Ejection prématurée du couvercle de l'objectif du téléscope pendant la prise de contrôle après lancement. Perte de l'hydrogène liquide. Téléscope inutilisable.
Satellite	Juin 1999	Space Systems Loral	-	Japan MOT - MTSAT	NA	Rapatriement du champ de tir. Défaillance du système d'alimentation électique de satellites de même génération. Par la suite une défaillance de lancement conduira à la perte du satellite
Satellite	Juin 1999	Astrium	ES3000	NSS - K TV	NA	Perte de puissance anormale des cellules solaires. Lancement repoussé d'un an. Satellite rapatrié du lieu de lancement sur le lieu de fabrication pour modification.
Satellite	Juin 1999	Astrium	ES3000	Worldspace - Asiastar	NA	Perte de puissance anormale du réseau de cellules solaires. Lancement repoussé. 1999-10
Satellite	Juin 1999	Astrium	ES3000	Worldspace - Ameristar	NA	Perte de puissance anormale des cellules solaires. Lancement repoussé.
Satellite	Juin 1999	Astrium	ES3000	SES - Astra 2B	NA	Perte de puissance anormale des cellules solaires. Lancement repoussé d'un an.
Satellite	Juil. 1999	Astrium	ES3000	Eutelsat - Hot Bird 5	NA	Après moins d'un an en orbite perte anormale de puissance des cellules solaires. Durée de vie se réduit.
Satellite	Août 1999	Lockeed Martin	A2100	Indonésie - Telkom 1	NA	Depuis son lancement en Juillet 1999 : mauvais fonctionnement du moteur double de rotation d'une aile de générateur solaire. Fonctionnement dégradé de la mission.
Satellite	Sept. 1999	Space Systems Loral	FS1300	Panamsat - PAS 8	NA	Indisponibilité temporaire de tous les canaux (2 à 3 heures) 1999-15
Satellite	Sept. 1999	Space Systems Loral	FS1300	Panamsat - PAS 7	NA	Perte d'un canal 1999-16
Satellite	Oct.1999	Brésil	NA	Brésil - SACI-1	NA	Perte de contact du satellite scientifique. Lancement par la Chine il y a 1 semaine.

A.3 Autres types

Matériel	Date	Compagnie	Famille	Nom	Charge utile	Défaillance
Sonde	24 Sept 1999	NASA	NA	Mars Climate Observer	NA	Disparition à l'approche de Mars.
Station de contrôle	24 Juin 1998	Aérospatiale-Matra Matra Marconi Space	-	ESA - SOHO	NA	Passage de la sonde spatiale en mode secours. Perte de la liaison. Erreur dans les commandes et mauvaise décision au centre de contrôle au sol.
Pas de tir	8 Mai 1999	Boeing	Delta	Delta II	GPS IIR 3	Pluie à l'intérieur de la salle blanche au sommet du lanceur. Satellite 'noyé'. Toit de protection imperméable mal accroché.
Téléscope	15 Nov.1999	NASA + Industrie US	NA	NA	NASA - Hubble	Mort en orbite d'un 3 ème gyroscope (sur 6) de stabilisation. Mise en sommeil du téléscope en attendant une réparation en orbite par la navette US. Fin décembre1999 le téléscope est réparé en orbite par l'équipage de la navette spatiale américaine lancée à cet effet. Il est de nouveau opérationnel.

Annexe B

Traces de l'automate *step1.tg*

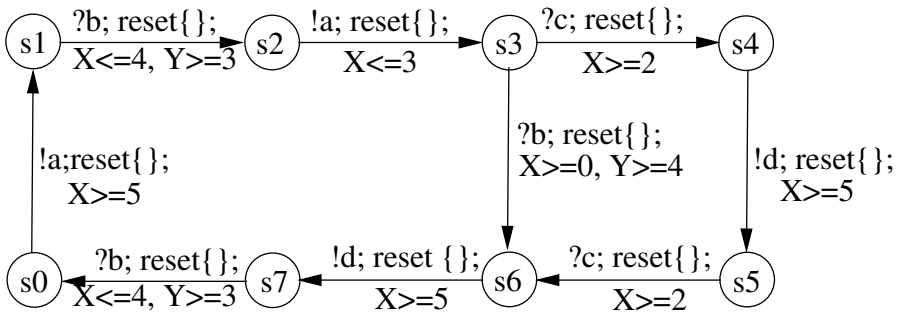


FIG. B.1 – Automate ne nécessitant que le passage à l'étape 1

B.1 Trace de l'exécution

```
+-----------------------------------------------------+
| Start of the analysis of the file step1.tg          |
+-----------------------------------------------------+

Numbers of clocks and variables : 2 -> Y X
States number : 8
State s_0 no0, transitions : 1
Destination state : 1, label : [E:!a G: -X <= -5 R:]
State s_1 no1, transitions : 1
Destination state : 2, label : [E:?b G: +X <= 4 and  -Y <= -3 R:]
State s_2 no2, transitions : 1
Destination state : 3, label : [E:!a G: +X <= 3 R:]
State s_3 no3, transitions : 2
Destination state : 4, label : [E:?c G: -X <= -2 R:]
Destination state : 6, label : [E:?b G: -X <= 0 and  -Y <= -4 R:]
State s_4 no4, transitions : 1
Destination state : 5, label : [E:!d G: -X <= -5 R:]
State s_5 no5, transitions : 1
Destination state : 6, label : [E:?c G: -X <= -2 R:]
State s_6 no6, transitions : 1
Destination state : 7, label : [E:!d G: -X <= -5 R:]
State s_7 no7, transitions : 1
Destination state : 0, label : [E:?b G: +X <= 4 and  -Y <= -3 R:]

+-----------------------------------------------------+
```

```
| End of the analysis of the file  step1.tg          |
+---------------------------------------------------+

+---------------------------------------------------+
|             Beginning of the algorithm            |
+---------------------------------------------------+

Begining of Step1
Controllable States :
  state 0 : 0
  state 1 : 1
  state 2 : 0
  state 3 : 1
  state 4 : 0
  state 5 : 1
  state 6 : 0
  state 7 : 1

OS!=0 ? : 0 (0=not empty    1=empty)
1: We begin a loop of while1 - d=1
1: CurrentOs : 1

  3 : New value of OtherOs : 3
  3: New value of CurrentSeq : [E:?b G: +X <= 4 and  -Y <= -3 R:], [E:!a G: +X <= 3 R:]
   4a: CurrentOs : 1 OtherOs : 3
   4a: d = 1 , current_seq = [E:?b G: +X <= 4 and  -Y <= -3 R:], [E:!a G: +X <= 3 R:]
   4a: New value of Recognized : FALSE : state 3 not recognized sequence [E:?b G: +X <= 4 and  -Y <= -3 R:],
       [E:!a G: +X <= 3 R:]
   4a: New value of other_os : 5
   4a: CurrentOs : 1 OtherOs : 5
   4a: d = 1 , current_seq = [E:?b G: +X <= 4 and  -Y <= -3 R:], [E:!a G: +X <= 3 R:]
   4a: New value of Recognized : FALSE : state 5 not recognized sequence [E:?b G: +X <= 4 and  -Y <= -3 R:],
       [E:!a G: +X <= 3 R:]
   4a: New value of other_os : 7
   4a: CurrentOs : 1 OtherOs : 7
   4a: d = 1 , current_seq = [E:?b G: +X <= 4 and  -Y <= -3 R:], [E:!a G: +X <= 3 R:]
   4a: New value of Recognized : FALSE : state 7 not recognized sequence [E:?b G: +X <= 4 and  -Y <= -3 R:],
       [E:!a G: +X <= 3 R:]
   4a: New value of other_os : -1
   4m: Value of Recognized : 0
   4b: We have identify the state 1 with the sequence [E:?b G: +X <= 4 and  -Y <= -3 R:],
       [E:!a G: +X <= 3 R:]
   4b: We have saved the sequence and changed status of state 1
 2: New value of current_os : 3
  3 : New value of OtherOs : 1
  3: New value of CurrentSeq : [E:?c G: -X <= -2 R:], [E:!d G: -X <= -5 R:]
   4a: CurrentOs : 3 OtherOs : 1
   4a: d = 1 , current_seq = [E:?c G: -X <= -2 R:], [E:!d G: -X <= -5 R:]
   4a: New value of Recognized : FALSE : state 1 not recognized sequence [E:?c G: -X <= -2 R:],
       [E:!d G: -X <= -5 R:]
   4a: New value of other_os : 5
   4a: CurrentOs : 3 OtherOs : 5
   4a: d = 1 , current_seq = [E:?c G: -X <= -2 R:], [E:!d G: -X <= -5 R:]
   4a: New value of Recognized : TRUE : state 5 recognized sequence [E:?c G: -X <= -2 R:],
       [E:!d G: -X <= -5 R:]
   4a: New value of other_os : 7
   4a: CurrentOs : 3 OtherOs : 7
   4a: d = 1 , current_seq = [E:?c G: -X <= -2 R:], [E:!d G: -X <= -5 R:]
   4a: New value of Recognized : FALSE : state 7 not recognized sequence [E:?c G: -X <= -2 R:],
       [E:!d G: -X <= -5 R:]
   4a: New value of other_os : -1
   4m: Value of Recognized : 1
   4b: We have not identify the state 3 with the sequence [E:?c G: -X <= -2 R:], [E:!d G: -X <= -5 R:]
   - It has been recognized by 1 state(s)
   4b: We didn't have any sequence for this state - We store it
   4b: sequence saved : [E:?c G: -X <= -2 R:], [E:!d G: -X <= -5 R:]  for state 3 with status 1
  3 : New value of OtherOs : 1
  3: New value of CurrentSeq : [E:?b G: -X <= 0 and  -Y <= -4 R:], [E:!d G: -X <= -5 R:]
   4a: CurrentOs : 3 OtherOs : 1
```

```
   4a: d = 1 , current_seq = [E:?b G: -X <= 0 and  -Y <= -4 R:], [E:!d G: -X <= -5 R:]
   4a: New value of Recognized : FALSE : state 1 not recognized sequence [E:?b G: -X <= 0 and  -Y <= -4 R:],
      [E:!d G: -X <= -5 R:]
   4a: New value of other_os : 5
   4a: CurrentOs : 3 OtherOs : 5
   4a: d = 1 , current_seq = [E:?b G: -X <= 0 and  -Y <= -4 R:], [E:!d G: -X <= -5 R:]
   4a: New value of Recognized : FALSE : state 5 not recognized sequence [E:?b G: -X <= 0 and  -Y <= -4 R:],
      [E:!d G: -X <= -5 R:]
   4a: New value of other_os : 7
   4a: CurrentOs : 3 OtherOs : 7
   4a: d = 1 , current_seq = [E:?b G: -X <= 0 and  -Y <= -4 R:], [E:!d G: -X <= -5 R:]
   4a: New value of Recognized : FALSE : state 7 not recognized sequence [E:?b G: -X <= 0 and  -Y <= -4 R:],
      [E:!d G: -X <= -5 R:]
   4a: New value of other_os : -1
   4m: Value of Recognized : 0
   4b: We have identify the state 3 with the sequence [E:?b G: -X <= 0 and  -Y <= -4 R:],
      [E:!d G: -X <= -5 R:]
   4b: We have saved the sequence and changed status of state 3
 2: New value of current_os : 5
  3 : New value of OtherOs : 1
  3: New value of CurrentSeq : [E:?c G: -X <= -2 R:], [E:!d G: -X <= -5 R:]
   4a: CurrentOs : 5 OtherOs : 1
   4a: d = 1 , current_seq = [E:?c G: -X <= -2 R:], [E:!d G: -X <= -5 R:]
   4a: New value of Recognized : FALSE : state 1 not recognized sequence [E:?c G: -X <= -2 R:],
      [E:!d G: -X <= -5 R:]
   4a: New value of other_os : 3
   4a: CurrentOs : 5 OtherOs : 3
   4a: d = 1 , current_seq = [E:?c G: -X <= -2 R:], [E:!d G: -X <= -5 R:]
   4a: New value of Recognized : TRUE : state 3 recognized sequence [E:?c G: -X <= -2 R:],
      [E:!d G: -X <= -5 R:]
   4a: New value of other_os : 7
   4a: CurrentOs : 5 OtherOs : 7
   4a: d = 1 , current_seq = [E:?c G: -X <= -2 R:], [E:!d G: -X <= -5 R:]
   4a: New value of Recognized : FALSE : state 7 not recognized sequence [E:?c G: -X <= -2 R:],
      [E:!d G: -X <= -5 R:]
   4a: New value of other_os : -1
   4m: Value of Recognized : 1
   4b: We have not identify the state 5 with the sequence [E:?c G: -X <= -2 R:], [E:!d G: -X <= -5 R:]
   - It has been recognized by 1 state(s)
   4b: We didn't have any sequence for this state - We store it
   4b: sequence saved : [E:?c G: -X <= -2 R:], [E:!d G: -X <= -5 R:]  for state 5 with status 1
 2: New value of current_os : 7
  3 : New value of OtherOs : 1
  3: New value of CurrentSeq : [E:?b G: +X <= 4 and  -Y <= -3 R:], [E:!a G: -X <= -5 R:]
   4a: CurrentOs : 7 OtherOs : 1
   4a: d = 1 , current_seq = [E:?b G: +X <= 4 and  -Y <= -3 R:], [E:!a G: -X <= -5 R:]
   4a: New value of Recognized : FALSE : state 1 not recognized sequence [E:?b G: +X <= 4 and  -Y <= -3 R:],
      [E:!a G: -X <= -5 R:]
   4a: New value of other_os : 3
   4a: CurrentOs : 7 OtherOs : 3
   4a: d = 1 , current_seq = [E:?b G: +X <= 4 and  -Y <= -3 R:], [E:!a G: -X <= -5 R:]
   4a: New value of Recognized : FALSE : state 3 not recognized sequence [E:?b G: +X <= 4 and  -Y <= -3 R:],
      [E:!a G: -X <= -5 R:]
   4a: New value of other_os : 5
   4a: CurrentOs : 7 OtherOs : 5
   4a: d = 1 , current_seq = [E:?b G: +X <= 4 and  -Y <= -3 R:], [E:!a G: -X <= -5 R:]
   4a: New value of Recognized : FALSE : state 5 not recognized sequence [E:?b G: +X <= 4 and  -Y <= -3 R:],
      [E:!a G: -X <= -5 R:]
   4a: New value of other_os : -1
   4m: Value of Recognized : 0
   4b: We have identify the state 7 with the sequence [E:?b G: +X <= 4 and  -Y <= -3 R:],
      [E:!a G: -X <= -5 R:]
   4b: We have saved the sequence and changed status of state 7
 2: New value of current_os : -1
1: New value of d : 2
1: We begin a loop of while1 - d=2
1: CurrentOs : 5

  3 : New value of OtherOs : 1
```

```
  3: New value of CurrentSeq : [E:?c G: -X <= -2 R:], [E:!d G: -X <= -5 R:],
     [E:?b G: +X <= 4 and  -Y <= -3 R:], [E:!a G: -X <= -5 R:]
   4a: CurrentOs : 5 OtherOs : 1
   4a: d = 2 , current_seq = [E:?c G: -X <= -2 R:], [E:!d G: -X <= -5 R:],
     [E:?b G: +X <= 4 and  -Y <= -3 R:], [E:!a G: -X <= -5 R:]
   4a: New value of Recognized : FALSE : state 1 not recognized sequence [E:?c G: -X <= -2 R:],
     [E:!d G: -X <= -5 R:], [E:?b G: +X <= 4 and  -Y <= -3 R:], [E:!a G: -X <= -5 R:]
   4a: New value of other_os : 3
   4a: CurrentOs : 5 OtherOs : 3
   4a: d = 2 , current_seq = [E:?c G: -X <= -2 R:], [E:!d G: -X <= -5 R:],
     [E:?b G: +X <= 4 and  -Y <= -3 R:], [E:!a G: -X <= -5 R:]
   4a: New value of Recognized : FALSE : state 3 not recognized sequence [E:?c G: -X <= -2 R:],
     [E:!d G: -X <= -5 R:], [E:?b G: +X <= 4 and  -Y <= -3 R:], [E:!a G: -X <= -5 R:]
   4a: New value of other_os : 7
   4a: CurrentOs : 5 OtherOs : 7
   4a: d = 2 , current_seq = [E:?c G: -X <= -2 R:], [E:!d G: -X <= -5 R:],
     [E:?b G: +X <= 4 and  -Y <= -3 R:], [E:!a G: -X <= -5 R:]
   4a: New value of Recognized : FALSE : state 7 not recognized sequence [E:?c G: -X <= -2 R:],
     [E:!d G: -X <= -5 R:], [E:?b G: +X <= 4 and  -Y <= -3 R:], [E:!a G: -X <= -5 R:]
   4a: New value of other_os : -1
   4m: Value of Recognized : 0
   4b: We have identify the state 5 with the sequence [E:?c G: -X <= -2 R:], [E:!d G: -X <= -5 R:],
     [E:?b G: +X <= 4 and  -Y <= -3 R:], [E:!a G: -X <= -5 R:]
   4b: We have saved the sequence and changed status of state 5
 2: New value of current_os : -1
1: New value of d : 3

End of the step1

Result of the step 1 :
  Number of recognized states : 4
  List of the identified states :
state 0 is not a controllable state
state 1 identified (step 1) with the sequence :
[E:?b G: +X <= 4 and  -Y <= -3 R:], [E:!a G: +X <= 3 R:]
state 2 is not a controllable state
state 3 identified (step 1) with the sequence :
[E:?b G: -X <= 0 and  -Y <= -4 R:], [E:!d G: -X <= -5 R:]
state 4 is not a controllable state
state 5 identified (step 1) with the sequence :
[E:?c G: -X <= -2 R:], [E:!d G: -X <= -5 R:], [E:?b G: +X <= 4 and  -Y <= -3 R:], [E:!a G: -X <= -5 R:]
state 6 is not a controllable state
state 7 identified (step 1) with the sequence :
[E:?b G: +X <= 4 and  -Y <= -3 R:], [E:!a G: -X <= -5 R:]

We do not need to process the step 2.

List of the remaining units of state : empty
We do not need to process the step 3.

+-----------------------------------------------------+
|                End of the algorithm                 |
+-----------------------------------------------------+

+-----------------------------------------------------+
| We have analysed the file step1.tg                  |
+-----------------------------------------------------+

Statistic (1 automata):
Number of controllable states : 4
States controllable at step 1 :     4 (100.00%)
States controllable at step 2 :     0 ( 0.00%)
States controllable at step 3 :     0 ( 0.00%)
```

B.2 Fichier de séquences associé

```
state 0 is not a controllable state
state 1 identified (step 1) with the sequence :
[E:?b G: +X <= 4 and  -Y <= -3 R:], [E:!a G: +X <= 3 R:]
state 2 is not a controllable state
state 3 identified (step 1) with the sequence :
[E:?b G: -X <= 0 and  -Y <= -4 R:], [E:!d G: -X <= -5 R:]
state 4 is not a controllable state
state 5 identified (step 1) with the sequence :
[E:?c G: -X <= -2 R:], [E:!d G: -X <= -5 R:], [E:?b G: +X <= 4 and  -Y <= -3 R:], [E:!a G: -X <= -5 R:]
state 6 is not a controllable state
state 7 identified (step 1) with the sequence :
[E:?b G: +X <= 4 and  -Y <= -3 R:], [E:!a G: -X <= -5 R:]
```

Annexe C

Traces de l'automate *step2.tg*

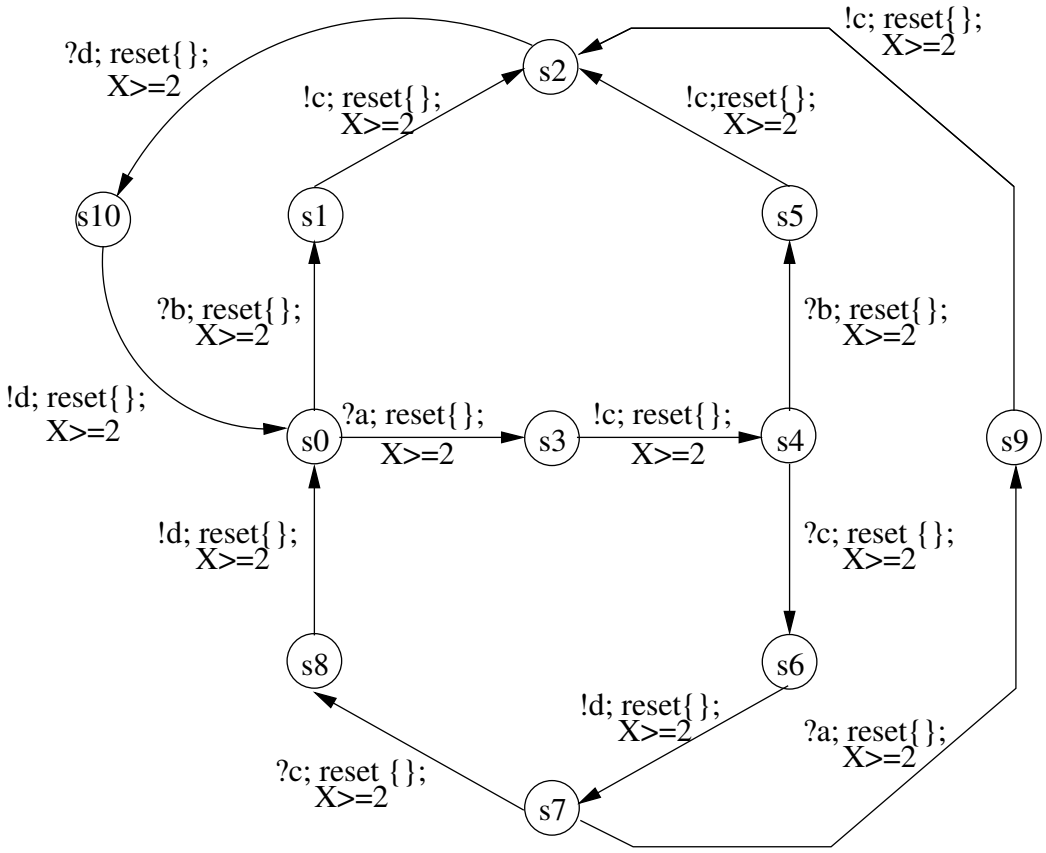


FIG. C.1 – Automate nécessitant le passage à l'étape 2

C.1 Trace de l'exécution

```
+-------------------------------------------------+
| Start of the analysis of the file step2.tg      |
+-------------------------------------------------+
```

```
Numbers of clocks and variables : 1 -> X
States number : 11
State s_0 no0, transitions : 2
Destination state : 1, label : [E:?b G: -X <= -2 R:]
Destination state : 3, label : [E:?a G: -X <= -2 R:]
State s_1 no1, transitions : 1
Destination state : 2, label : [E:!c G: -X <= -2 R:]
State s_2 no2, transitions : 1
Destination state : 10, label : [E:?d G: -X <= -2 R:]
State s_3 no3, transitions : 1
Destination state : 4, label : [E:!c G: -X <= -2 R:]
State s_4 no4, transitions : 2
Destination state : 6, label : [E:?c G: -X <= -2 R:]
Destination state : 5, label : [E:?b G: -X <= -2 R:]
State s_5 no5, transitions : 1
Destination state : 2, label : [E:!c G: -X <= -2 R:]
State s_6 no6, transitions : 1
Destination state : 7, label : [E:!d G: -X <= -2 R:]
State s_7 no7, transitions : 2
Destination state : 9, label : [E:?a G: -X <= -2 R:]
Destination state : 8, label : [E:?c G: -X <= -2 R:]
State s_8 no8, transitions : 1
Destination state : 0, label : [E:!d G: -X <= -2 R:]
State s_9 no9, transitions : 1
Destination state : 2, label : [E:!c G: -X <= -2 R:]
State s_10 no10, transitions : 1
Destination state : 0, label : [E:!d G: -X <= -2 R:]

+-----------------------------------------------------+
| End of the analysis of the file  step2.tg           |
+-----------------------------------------------------+

+-----------------------------------------------------+
|             Beginning of the algorithm              |
+-----------------------------------------------------+

Begining of Step1
Controllable States :
  state 0 : 1
  state 1 : 0
  state 2 : 1
  state 3 : 0
  state 4 : 1
  state 5 : 0
  state 6 : 0
  state 7 : 1
  state 8 : 0
  state 9 : 0
  state 10 : 0

OS!=0 ? : 0 (0=not empty    1=empty)
1: We begin a loop of while1 - d=1
1: CurrentOs : 0

  3 : New value of OtherOs : 2
  3: New value of CurrentSeq : [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
   4a: CurrentOs : 0 OtherOs : 2
   4a: d = 1 , current_seq = [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
   4a: New value of Recognized : FALSE : state 2 not recognized sequence [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
   4a: New value of other_os : 4
   4a: CurrentOs : 0 OtherOs : 4
   4a: d = 1 , current_seq = [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
   4a: New value of Recognized : TRUE : state 4 recognized sequence [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
   4a: New value of other_os : 7
   4a: CurrentOs : 0 OtherOs : 7
   4a: d = 1 , current_seq = [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
   4a: New value of Recognized : FALSE : state 7 not recognized sequence [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
   4a: New value of other_os : -1
   4m: Value of Recognized : 1
```

```
  4b: We have not identify the state 0 with the sequence [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  - It has been recognized by 1 state(s)
  4b: We didn't have any sequence for this state - We store it
  4b: sequence saved : [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]  for state 0 with status 1
 3 : New value of OtherOs : 2
 3: New value of CurrentSeq : [E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: CurrentOs : 0 OtherOs : 2
  4a: d = 1 , current_seq = [E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of Recognized : FALSE : state 2 not recognized sequence [E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of other_os : 4
  4a: CurrentOs : 0 OtherOs : 4
  4a: d = 1 , current_seq = [E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of Recognized : FALSE : state 4 not recognized sequence [E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of other_os : 7
  4a: CurrentOs : 0 OtherOs : 7
  4a: d = 1 , current_seq = [E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of Recognized : TRUE : state 7 recognized sequence [E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of other_os : -1
  4m: Value of Recognized : 1
  4b: We have not identify the state 0 with the sequence [E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  - It has been recognized by 1 state(s)
  4b: We have found a sequence for this state that is not the best one - We don't store it
2: New value of current_os : 2
 3 : New value of OtherOs : 0
 3: New value of CurrentSeq : [E:?d G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
  4a: CurrentOs : 2 OtherOs : 0
  4a: d = 1 , current_seq = [E:?d G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
  4a: New value of Recognized : FALSE : state 0 not recognized sequence [E:?d G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
  4a: New value of other_os : 4
  4a: CurrentOs : 2 OtherOs : 4
  4a: d = 1 , current_seq = [E:?d G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
  4a: New value of Recognized : FALSE : state 4 not recognized sequence [E:?d G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
  4a: New value of other_os : 7
  4a: CurrentOs : 2 OtherOs : 7
  4a: d = 1 , current_seq = [E:?d G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
  4a: New value of Recognized : FALSE : state 7 not recognized sequence [E:?d G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
  4a: New value of other_os : -1
  4m: Value of Recognized : 0
  4b: We have identify the state 2 with the sequence [E:?d G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
  4b: We have saved the sequence and changed status of state 2
2: New value of current_os : 4
 3 : New value of OtherOs : 0
 3: New value of CurrentSeq : [E:?c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
  4a: CurrentOs : 4 OtherOs : 0
  4a: d = 1 , current_seq = [E:?c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
  4a: New value of Recognized : FALSE : state 0 not recognized sequence [E:?c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
  4a: New value of other_os : 2
  4a: CurrentOs : 4 OtherOs : 2
  4a: d = 1 , current_seq = [E:?c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
  4a: New value of Recognized : FALSE : state 2 not recognized sequence [E:?c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
  4a: New value of other_os : 7
  4a: CurrentOs : 4 OtherOs : 7
  4a: d = 1 , current_seq = [E:?c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
  4a: New value of Recognized : TRUE : state 7 recognized sequence [E:?c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
  4a: New value of other_os : -1
  4m: Value of Recognized : 1
  4b: We have not identify the state 4 with the sequence [E:?c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
  - It has been recognized by 1 state(s)
  4b: We didn't have any sequence for this state - We store it
  4b: sequence saved : [E:?c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]  for state 4 with status 1
 3 : New value of OtherOs : 0
 3: New value of CurrentSeq : [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: CurrentOs : 4 OtherOs : 0
  4a: d = 1 , current_seq = [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of Recognized : TRUE : state 0 recognized sequence [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of other_os : 2
  4a: CurrentOs : 4 OtherOs : 2
  4a: d = 1 , current_seq = [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of Recognized : FALSE : state 2 not recognized sequence [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
```

```
   4a: New value of other_os : 7
   4a: CurrentOs : 4 OtherOs : 7
   4a: d = 1 , current_seq = [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
   4a: New value of Recognized : FALSE : state 7 not recognized sequence [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
   4a: New value of other_os : -1
   4m: Value of Recognized : 1
   4b: We have not identify the state 4 with the sequence [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
   - It has been recognized by 1 state(s)
   4b: We have found a sequence for this state that is not the best one - We don't store it
 2: New value of current_os : 7
  3 : New value of OtherOs : 0
  3: New value of CurrentSeq : [E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
   4a: CurrentOs : 7 OtherOs : 0
   4a: d = 1 , current_seq = [E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
   4a: New value of Recognized : TRUE : state 0 recognized sequence [E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
   4a: New value of other_os : 2
   4a: CurrentOs : 7 OtherOs : 2
   4a: d = 1 , current_seq = [E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
   4a: New value of Recognized : FALSE : state 2 not recognized sequence [E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
   4a: New value of other_os : 4
   4a: CurrentOs : 7 OtherOs : 4
   4a: d = 1 , current_seq = [E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
   4a: New value of Recognized : FALSE : state 4 not recognized sequence [E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
   4a: New value of other_os : -1
   4m: Value of Recognized : 1
   4b: We have not identify the state 7 with the sequence [E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
   - It has been recognized by 1 state(s)
   4b: We didn't have any sequence for this state - We store it
   4b: sequence saved : [E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]  for state 7 with status 1
  3 : New value of OtherOs : 0
  3: New value of CurrentSeq : [E:?c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
   4a: CurrentOs : 7 OtherOs : 0
   4a: d = 1 , current_seq = [E:?c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
   4a: New value of Recognized : FALSE : state 0 not recognized sequence [E:?c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
   4a: New value of other_os : 2
   4a: CurrentOs : 7 OtherOs : 2
   4a: d = 1 , current_seq = [E:?c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
   4a: New value of Recognized : FALSE : state 2 not recognized sequence [E:?c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
   4a: New value of other_os : 4
   4a: CurrentOs : 7 OtherOs : 4
   4a: d = 1 , current_seq = [E:?c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
   4a: New value of Recognized : TRUE : state 4 recognized sequence [E:?c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
   4a: New value of other_os : -1
   4m: Value of Recognized : 1
   4b: We have not identify the state 7 with the sequence [E:?c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
   - It has been recognized by 1 state(s)
   4b: We have found a sequence for this state that is not the best one - We don't store it
 2: New value of current_os : -1
1: New value of d : 2

End of the step1

Result of the step 1 :
  Number of recognized states : 1
  List of the identified states :
    state 0 has the sequence : [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:] recognized by 1 state(s)
state 1 is not a controllable state
state 2 identified (step 1) with the sequence :
[E:?d G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
state 3 is not a controllable state
    state 4 has the sequence : [E:?c G: -X <= -2 R:], [E:!d G: -X <= -2 R:] recognized by 1 state(s)
state 5 is not a controllable state
state 6 is not a controllable state
    state 7 has the sequence : [E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:] recognized by 1 state(s)
state 8 is not a controllable state
state 9 is not a controllable state
state 10 is not a controllable state

Begining of the Step2
```

```
States to control :
7 4 0
Starting function for states : 7 4 0
 Current State : 7 and current sequence : [E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
 We found a sequence which separate the unit
left unit : 0 7   ---- right unit : 4

Starting function for states : 0 7
 Current State : 0 and current sequence : [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
 We found a sequence which separate the unit
left unit : 0   ---- right unit : 7

Tree build in step2 : ( [ 7 4 0 [E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:] ]( [ 4 ][][])( [ 0 7 [E:?b G: -X <= -2 R:], [E:!c G

We visit the tree to get the results
  We found a leaf with a state alone
    State 4 with sequences :
[E:?c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
not([E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:] )

    We saved it
  We found a leaf with a state alone
    State 7 with sequences :
not([E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:] )
[E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]

    We saved it
  We found a leaf with a state alone
    State 0 with sequences :
[E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
[E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]

    We saved it

End of the step2

Result of the step 2 :
  Number of recognized states : 4
  List of the identified states :
state 0 identified (step 2) with the sequence :
[E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
[E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
state 1 is not a controllable state
state 2 identified (step 1) with the sequence :
[E:?d G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
state 3 is not a controllable state
state 4 identified (step 2) with the sequence :
[E:?c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
not([E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:] )
state 5 is not a controllable state
state 6 is not a controllable state
state 7 identified (step 2) with the sequence :
not([E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:] )
[E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
state 8 is not a controllable state
state 9 is not a controllable state
state 10 is not a controllable state

List of the remaining units of state : empty
We do not need to process the step 3.

+-----------------------------------------------------+
|                End of the algorithm                 |
+-----------------------------------------------------+

+-----------------------------------------------------+
| We have analysed the file step2.tg                  |
+-----------------------------------------------------+
```

```
Statistic (1 automata):
Number of controllable states : 4
States controllable at step 1 :     1 (25.00%)
States controllable at step 2 :     3 (75.00%)
States controllable at step 3 :     0 ( 0.00%)
```

C.2 Fichier de séquences associé

```
state 0 identified (step 2) with the sequence :
[E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
[E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
state 1 is not a controllable state
state 2 identified (step 1) with the sequence :
[E:?d G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
state 3 is not a controllable state
state 4 identified (step 2) with the sequence :
[E:?c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
not([E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:] )
state 5 is not a controllable state
state 6 is not a controllable state
state 7 identified (step 2) with the sequence :
not([E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:] )
[E:?a G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
state 8 is not a controllable state
state 9 is not a controllable state
state 10 is not a controllable state
```

Annexe D

Traces de l'automate *step3.tg*

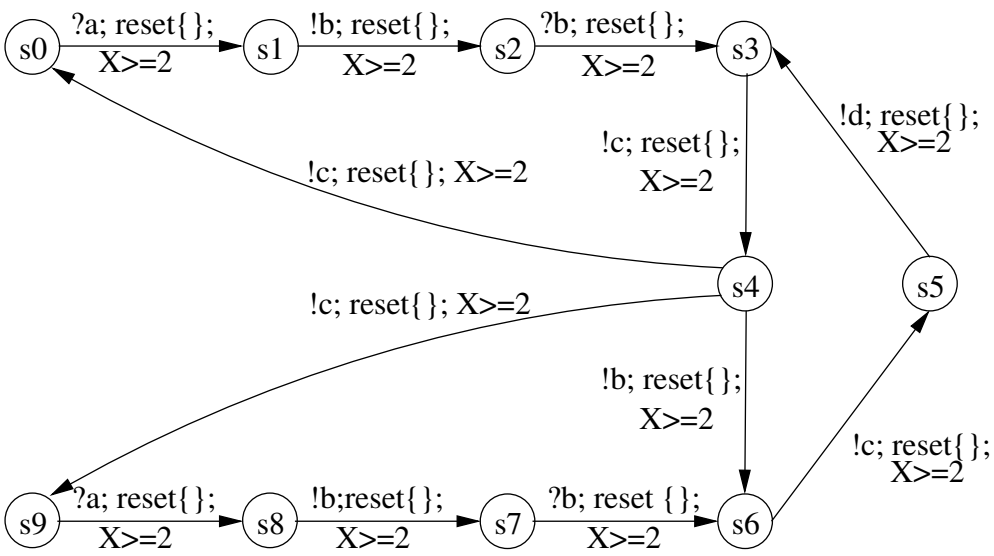


FIG. D.1 – Automate nécessitant le passage à l'étape 3

D.1 Trace de l'exécution

```
+-----------------------------------------------------+
| Start of the analysis of the file step3.tg          |
+-----------------------------------------------------+

Numbers of clocks and variables : 1 -> X
States number : 10
State s_0 no0, transitions : 1
Destination state : 1, label : [E:?a G: -X <= -2 R:]
State s_1 no1, transitions : 1
Destination state : 2, label : [E:!b G: -X <= -2 R:]
State s_2 no2, transitions : 1
Destination state : 3, label : [E:?b G: -X <= -2 R:]
State s_3 no3, transitions : 1
Destination state : 4, label : [E:!c G: -X <= -2 R:]
State s_4 no4, transitions : 3
Destination state : 6, label : [E:!b G: -X <= -2 R:]
```

```
Destination state : 0, label : [E:!c G: -X <= -2 R:]
Destination state : 9, label : [E:!c G: -X <= -2 R:]
State s_5 no5, transitions : 1
Destination state : 3, label : [E:!d G: -X <= -2 R:]
State s_6 no6, transitions : 1
Destination state : 5, label : [E:!c G: -X <= -2 R:]
State s_7 no7, transitions : 1
Destination state : 6, label : [E:?b G: -X <= -2 R:]
State s_8 no8, transitions : 1
Destination state : 7, label : [E:!b G: -X <= -2 R:]
State s_9 no9, transitions : 1
Destination state : 8, label : [E:?a G: -X <= -2 R:]

+--------------------------------------------------+
| End of the analysis of the file  step3.tg        |
+--------------------------------------------------+

Warning !!! The automaton isn't determinist

+--------------------------------------------------+
|             Beginning of the algorithm           |
+--------------------------------------------------+

Begining of Step1
Controllable States :
  state 0 : 1
  state 1 : 0
  state 2 : 1
  state 3 : 0
  state 4 : 0
  state 5 : 0
  state 6 : 0
  state 7 : 1
  state 8 : 0
  state 9 : 1

OS!=0 ? : 0 (0=not empty    1=empty)
1: We begin a loop of while1 - d=1
1: CurrentOs : 0

  3 : New value of OtherOs : 2
  3: New value of CurrentSeq : [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
   4a: CurrentOs : 0 OtherOs : 2
   4a: d = 1 , current_seq = [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
   4a: New value of Recognized : FALSE : state 2 not recognized sequence
       [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
   4a: New value of other_os : 7
   4a: CurrentOs : 0 OtherOs : 7
   4a: d = 1 , current_seq = [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
   4a: New value of Recognized : FALSE : state 7 not recognized sequence
       [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
   4a: New value of other_os : 9
   4a: CurrentOs : 0 OtherOs : 9
   4a: d = 1 , current_seq = [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
   4a: New value of Recognized : TRUE : state 9 recognized sequence
       [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
   4a: New value of other_os : -1
   4m: Value of Recognized : 1
   4b: We have not identify the state 0 with the sequence [E:?a G: -X <= -2 R:],
       [E:!b G: -X <= -2 R:]
   - It has been recognized by 1 state(s)
   4b: We didn't have any sequence for this state - We store it
   4b: sequence saved : [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
       for state 0 with status 1
 2: New value of current_os : 2
  3 : New value of OtherOs : 0
  3: New value of CurrentSeq : [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
   4a: CurrentOs : 2 OtherOs : 0
   4a: d = 1 , current_seq = [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
```

```
  4a: New value of Recognized : FALSE : state 0 not recognized sequence
      [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of other_os : 7
  4a: CurrentOs : 2 OtherOs : 7
  4a: d = 1 , current_seq = [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of Recognized : TRUE : state 7 recognized sequence
      [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of other_os : 9
  4a: CurrentOs : 2 OtherOs : 9
  4a: d = 1 , current_seq = [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of Recognized : FALSE : state 9 not recognized sequence
      [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of other_os : -1
  4m: Value of Recognized : 1
  4b: We have not identify the state 2 with the sequence [E:?b G: -X <= -2 R:],
      [E:!c G: -X <= -2 R:]
  - It has been recognized by 1 state(s)
  4b: We didn't have any sequence for this state - We store it
  4b: sequence saved : [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
      for state 2 with status 1
 3 : New value of OtherOs : 0
 3: New value of CurrentSeq : [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:],
    [E:!c G: -X <= -2 R:]
  4a: CurrentOs : 2 OtherOs : 0
  4a: d = 1 , current_seq = [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:],
     [E:!c G: -X <= -2 R:]
  4a: New value of Recognized : FALSE : state 0 not recognized sequence
      [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of other_os : 7
  4a: CurrentOs : 2 OtherOs : 7
  4a: d = 1 , current_seq = [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:],
      [E:!c G: -X <= -2 R:]
  4a: New value of Recognized : FALSE : state 7 not recognized sequence
      [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of other_os : 9
  4a: CurrentOs : 2 OtherOs : 9
  4a: d = 1 , current_seq = [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:],
      [E:!c G: -X <= -2 R:]
  4a: New value of Recognized : FALSE : state 9 not recognized sequence
      [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of other_os : -1
  4m: Value of Recognized : 0
  4b: We have identify the state 2 with the sequence [E:?b G: -X <= -2 R:],
      [E:!c G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4b: We have saved the sequence and changed status of state 2
2: New value of current_os : 7
 3 : New value of OtherOs : 0
 3: New value of CurrentSeq : [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: CurrentOs : 7 OtherOs : 0
  4a: d = 1 , current_seq = [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of Recognized : FALSE : state 0 not recognized sequence
      [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of other_os : 2
  4a: CurrentOs : 7 OtherOs : 2
  4a: d = 1 , current_seq = [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of Recognized : TRUE : state 2 recognized sequence
      [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of other_os : 9
  4a: CurrentOs : 7 OtherOs : 9
  4a: d = 1 , current_seq = [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of Recognized : FALSE : state 9 not recognized sequence
      [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  4a: New value of other_os : -1
  4m: Value of Recognized : 1
  4b: We have not identify the state 7 with the sequence [E:?b G: -X <= -2 R:],
      [E:!c G: -X <= -2 R:]
  - It has been recognized by 1 state(s)
  4b: We didn't have any sequence for this state - We store it
  4b: sequence saved : [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
```

```
      for state 7 with status 1
  3 : New value of OtherOs : 0
  3: New value of CurrentSeq : [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:],
     [E:!d G: -X <= -2 R:]
   4a: CurrentOs : 7 OtherOs : 0
   4a: d = 1 , current_seq = [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:],
     [E:!d G: -X <= -2 R:]
   4a: New value of Recognized : FALSE : state 0 not recognized sequence
     [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
   4a: New value of other_os : 2
   4a: CurrentOs : 7 OtherOs : 2
   4a: d = 1 , current_seq = [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:],
     [E:!d G: -X <= -2 R:]
   4a: New value of Recognized : FALSE : state 2 not recognized sequence
     [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
   4a: New value of other_os : 9
   4a: CurrentOs : 7 OtherOs : 9
   4a: d = 1 , current_seq = [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:],
     [E:!d G: -X <= -2 R:]
   4a: New value of Recognized : FALSE : state 9 not recognized sequence
     [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
   4a: New value of other_os : -1
   4m: Value of Recognized : 0
   4b: We have identify the state 7 with the sequence [E:?b G: -X <= -2 R:],
     [E:!c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
   4b: We have saved the sequence and changed status of state 7
 2: New value of current_os : 9
  3 : New value of OtherOs : 0
  3: New value of CurrentSeq : [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
   4a: CurrentOs : 9 OtherOs : 0
   4a: d = 1 , current_seq = [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
   4a: New value of Recognized : TRUE : state 0 recognized sequence
     [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
   4a: New value of other_os : 2
   4a: CurrentOs : 9 OtherOs : 2
   4a: d = 1 , current_seq = [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
   4a: New value of Recognized : FALSE : state 2 not recognized sequence
     [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
   4a: New value of other_os : 7
   4a: CurrentOs : 9 OtherOs : 7
   4a: d = 1 , current_seq = [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
   4a: New value of Recognized : FALSE : state 7 not recognized sequence
     [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
   4a: New value of other_os : -1
   4m: Value of Recognized : 1
   4b: We have not identify the state 9 with the sequence [E:?a G: -X <= -2 R:],
     [E:!b G: -X <= -2 R:]
   - It has been recognized by 1 state(s)
   4b: We didn't have any sequence for this state - We store it
   4b: sequence saved : [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
     for state 9 with status 1
 2: New value of current_os : -1
1: New value of d : 2

End of the step1

Result of the step 1 :
  Number of recognized states : 2
  List of the identified states :
    state 0 has the sequence : [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
    recognized by 1 state(s)
state 1 is not a controllable state
state 2 identified (step 1) with the sequence :
[E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
state 3 is not a controllable state
state 4 is not a controllable state
state 5 is not a controllable state
state 6 is not a controllable state
state 7 identified (step 1) with the sequence :
```

```
[E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
state 8 is not a controllable state
    state 9 has the sequence : [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
    recognized by 1 state(s)

Begining of the Step2
States to control :
9 0
Starting function for states : 9 0
 Current State : 9 and current sequence : [E:?a G: -X <= -2 R:],
     [E:!b G: -X <= -2 R:]
 The sequence was recognized by all state
 Current State : 0 and current sequence : [E:?a G: -X <= -2 R:],
     [E:!b G: -X <= -2 R:]
 The sequence was recognized by all state
No more sequence to test in the current node

Tree build in step2 : ( [ 9 0 ][][])

We visit the tree to get the results
  We found a leaf with a unit of state
    States 9 0 with sequences :
[E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
    We saved them

End of the step2

Result of the step 2 :
  Number of recognized states : 2
  List of the identified states :
    state 0 was not recognized at step 2 with the sequence :
[E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
state 1 is not a controllable state
state 2 identified (step 1) with the sequence :
[E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
state 3 is not a controllable state
state 4 is not a controllable state
state 5 is not a controllable state
state 6 is not a controllable state
state 7 identified (step 1) with the sequence :
[E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
state 8 is not a controllable state
    state 9 was not recognized at step 2 with the sequence :
[E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]

List of the remaining units of state : ( [ 0 9 ][][])
Begining of the Step3
Starting function with depth 1 for states : 0 9
 1: Current state : 0
  2: Current sequence : [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
  2: The sequence [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
     was recognized by all states
 1: No good sequence found for state : 0

 1: Current state : 9
  2: Current sequence : [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
  2: The sequence [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]$
     was recognized by all states
 1: No good sequence found for state : 9

At depth 1, no sequence found. New value of depth : 2
 1: Current state : 0
  2: Current sequence : [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:],
    [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  2: The sequence [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:],
    [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:]  was recognized by all states
  2: Current sequence : [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:],
    [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
  2: We found a sequence which separate the unit
```

```
The unit of state was separate by sequence [E:?a G: -X <= -2 R:],
    [E:!b G: -X <= -2 R:], [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:],
    [E:!c G: -X <= -2 R:]
Trees build in step3 :
Tree 1 : ( [ 0 9 [E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:],
    [E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:], [E:!c G: -X <= -2 R:] ]
    ( [ 9 ][][])( [ 0 ][][]))


We visit the trees to get the results
  We found a leaf with an alone state
    State 9 with sequences :
not([E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:], [E:!d G: -X <= -2 R:] )
not([E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:], [E:!c G: -X <= -2 R:] )
not([E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:], [E:?b G: -X <= -2 R:],
   [E:!c G: -X <= -2 R:], [E:!c G: -X <= -2 R:] )
[E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
    We saved it
  We found a leaf with an alone state
    State 0 with sequences :
[E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:], [E:?b G: -X <= -2 R:],
    [E:!c G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
[E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
    We saved it

End of the step3

Result of the step 3 :
  Number of recognized states : 4
state 0 identified (step 3) with the sequence :
[E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:], [E:?b G: -X <= -2 R:],
    [E:!c G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
[E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
state 1 is not a controllable state
state 2 identified (step 1) with the sequence :
[E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
state 3 is not a controllable state
state 4 is not a controllable state
state 5 is not a controllable state
state 6 is not a controllable state
state 7 identified (step 1) with the sequence :
[E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
state 8 is not a controllable state
state 9 identified (step 3) with the sequence :
not([E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:], [E:!d G: -X <= -2 R:] )
not([E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:], [E:!c G: -X <= -2 R:] )
not([E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:], [E:?b G: -X <= -2 R:],
    [E:!c G: -X <= -2 R:], [E:!c G: -X <= -2 R:] )
[E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]

+-----------------------------------------------------+
|                End of the algorithm                 |
+-----------------------------------------------------+

+-----------------------------------------------------+
| We have analysed the file step3.tg                  |
+-----------------------------------------------------+

Statistic (1 automata):
Number of controllable states : 4
States controllable at step 1 :     2 (50.00%)
States controllable at step 2 :     0 ( 0.00%)
States controllable at step 3 :     2 (50.00%)
```

D.2 Fichier de séquences associé

```
state 0 identified (step 3) with the sequence :
[E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:], [E:?b G: -X <= -2 R:],
   [E:!c G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
[E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
state 1 is not a controllable state
state 2 identified (step 1) with the sequence :
[E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:], [E:!c G: -X <= -2 R:]
state 3 is not a controllable state
state 4 is not a controllable state
state 5 is not a controllable state
state 6 is not a controllable state
state 7 identified (step 1) with the sequence :
[E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:], [E:!d G: -X <= -2 R:]
state 8 is not a controllable state
state 9 identified (step 3) with the sequence :
not([E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:], [E:!d G: -X <= -2 R:] )
not([E:?b G: -X <= -2 R:], [E:!c G: -X <= -2 R:], [E:!c G: -X <= -2 R:] )
not([E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:], [E:?b G: -X <= -2 R:],
   [E:!c G: -X <= -2 R:], [E:!c G: -X <= -2 R:] )
[E:?a G: -X <= -2 R:], [E:!b G: -X <= -2 R:]
```