université de BORDEAUX

Bordeaux INP
ENSEIRB
MATMECA

Manuscrit présenté à :

l'Université de Bordeaux

par :

Antoine Rollet

en vue de l'obtention d'une :

Habilitation à Diriger des Recherches

Voyage (dans le temps) autour de la vérification, du test, et au delà

Soutenue le 22 novembre 2018 devant le jury composé de :

Pascale Le Gall	Professeure à Centrale-Supélec Paris	Rapporteur
Stéphane Maag	Professeur à Télécom SudParis	Rapporteur
Hélène Waeselynck	Directrice de Recherche au CNRS, LAAS	Rapporteur
Dominique Méry	Professeur à l'Université de Lorraine	Président
Jérôme Leroux	Directeur de Recherche au CNRS, LaBRI	Examinateur
Richard Castanet	Professeur Emérite à Bordeaux INP	Invité

Remerciements

Cette Habilitation à Diriger des Recherches a été présentée devant le jury suivant :

— Pascale Le Gall, Professeure à Centrale-Supélec Paris, Rapporteur
— Stéphane Maag, Professeur à Télécom SudParis, Rapporteur
— Hélène Waeselynck, Directrice de Recherche au LAAS, Rapporteur
— Dominique Méry, Professeur à l'Université de Lorraine, Président
— Jérôme Leroux, Directeur de Recherche au LaBRI, Examinateur
— Richard Castanet, Professeur Emérite à Bordeaux INP, Invité

Je tiens à les remercier pour avoir accepté de participer à mon jury, le temps consacré à la lecture de ce manuscrit, et avoir pu se libérer pour assister à ma soutenance malgré des emplois du temps très chargés. Je les remercie aussi pour les retours riches et motivants qu'ils m'ont fait, et pour les questions et discussions très intéressantes qui ont suivi ma présentation.

Le LaBRI et l'Enseirb-Matmeca offrent un cadre de travail très privilégié. Merci à mes collègues, dont certains sont devenus des amis proches, pour les bons moments partagés durant ces années. Je remercie plus particulièrement Richard Castanet et Patrick Félix pour leur confiance lors de mon arrivée au LaBRI en 2005.

Enfin, je remercie mes proches pour leur soutien et pour tout le reste.

Table des matières

Chapitre 1

Introduction

Avant propos : *Ce document est une synthèse des travaux que j'ai effectués dans le domaine de la vérification et du test logiciel depuis ma soutenance de thèse de doctorat. Ils ont été réalisés au sein du LaBRI*[1]*, et durant mon affectation à Bordeaux INP*[2]*. L'objectif n'est pas de détailler l'ensemble des contributions, que le lecteur pourra trouver dans les articles associés, mais de donner une vue d'ensemble des travaux effectués. Le choix de rédaction est à la fois thématique, chronologique, et reprend le principe de la mémoire humaine : les contributions récentes sont bien plus détaillées que les anciennes. Ce choix assumé me permet à la fois de synthétiser un certain nombre de travaux plutôt anciens (travaux sur le test à base de modèles et la vérification de code des chapitres 2 et 3), mais aussi de décrire de façon détaillée mes travaux récents pour montrer au lecteur que derrière des idées qui peuvent paraître simples, il existe tout un travail de formalisation qui fait partie intégrante de ces contributions. Ainsi, le chapitre sur l'enforcement à l'exécution (chapitre 4), a été conçu pour être détaillé et autosuffisant d'un point de vue formel.*

Ces derniers mois, notre communauté scientifique s'est intéressée au cas de *PicSat*, un nanosatellite français dont on est sans nouvelles depuis le 20 mars 2018. Selon le directeur de mission [172] : *"La dernière hypothèse [...] penche pour un problème de logiciel. Sur les grandes missions spatiales, ces logiciels sont vérifiés et relus des milliers de fois, ce que nous n'avons pas pu faire avec notre petite équipe. Et le logiciel de PicSat est presque aussi complexe que celui d'un satellite de taille normale. S'il se met dans une boucle, par exemple, il est impossible de le réinitialiser à la main. Pour vérifier cette hypothèse, il va falloir repasser sur tout le code, ce qui va nous prendre du temps."*

Ce passage illustre bien les problématiques de la vérification et du test de logiciel. Vérifier proprement un logiciel, c'est coûteux. Mais ne pas le faire l'est généralement bien plus, comme le rappelle la multitude de bugs célèbres que l'on peut trouver dans la littérature ou dans la presse, et dont les conséquences financières ou humaines sont énormes. Le lecteur pourra trouver une analyse intéressante de certains bugs connus dans [90]. Une personne désireuse de consulter une liste très souvent mise à jour pourra aussi parcourir [217], avec toutes les précautions d'usage pour ce type de lien, qui permet de réaliser en temps-réel l'étendue du phénomène. On entend ici ou là des nombres astronomiques sur le coût total des bugs logiciels, mais en réalité, le coût d'un bug dépend essentiellement du moment où ce dernier est détecté. Par exemple,

1. Laboratoire Bordelais de Recherche en Informatique.
2. Successivement Enseirb, Enseirb-Matmeca, IPB, puis Bordeaux INP.

une erreur de spécification détectée avant d'implémenter le système aura un coût faible, alors que cette même erreur détectée en phase opérationnelle du produit aura un coût énorme (vies humaines, rappel de produit, etc...). Mais pourquoi est-ce si difficile de tester un programme ? Intuitivement, pour obtenir une confiance absolue dans un programme informatique, il faudrait tester toutes les situations possibles. Prenons un exemple simple : soit la fonction *C* suivante : `int addition (int i, int j);` dont l'objectif est de réaliser la somme de deux entiers quelconques. En supposant que les entiers sont codés sur 32 bits, tester l'ensemble des cas possibles donne 2^{64} cas. Si on estime qu'il est possible de produire un test par microseconde, alors il faudrait plus de 584000 ans pour réaliser un test exhaustif de cette fonction simple. Ajoutons à cela le fait que les programmes informatiques sont généralement bien plus compliqués que la simple addition de deux nombres entiers, et on comprend aisément qu'au final, tester un logiciel ne consiste pas à vérifier tous les cas possibles, mais plutôt à trouver ou appliquer des techniques permettant d'augmenter la confiance envers le logiciel en question. Les enjeux de recherche dans ces domaines consistent à diminuer au maximum l'effort de vérification et de test tout en maximisant leur degré de confiance. Pour cela, il est fréquent d'avoir recours notamment à des techniques d'automatisation de ces procédés (e.g. génération automatique de tests, automatisation du verdict), ou encore à des techniques d'abstraction pour diminuer le nombre de cas à explorer.

Avant de décrire nos travaux, nous allons les situer dans leur contexte, celui de la *vérification et du test de logiciel*. Il s'agit d'un domaine très vaste dans lequel énormément de travaux ont été faits depuis des décennies. Sans aucune prétention d'être exhaustif, nous allons tenter de fournir une vision globale de ce domaine, et en profiter pour fixer le vocabulaire associé. En effet, il arrive que des communautés différentes utilisent des méthodes similaires, et les nomment différemment. L'objectif est donc double : situer nos travaux au travers de ce vaste domaine de recherche, et définir certains termes que nous utiliserons par la suite. Nous proposons figure 1.1 une mise en perspective des principales méthodes de vérification et de test. Il s'agit de notre vision personnelle du domaine, probablement incomplète, et sûrement discutable. Une classification sous forme d'arbre s'étant avérée assez peu adaptée, nous avons opté pour une vision en rapport avec les étapes du cycle de développement. En partant d'une variante du modèle *Waterfall* que nous avons adaptée, notamment en séparant le code et l'implémentation d'un système, nous avons situé les méthodes de vérification et de test par rapport à ce modèle : ainsi, si une technique se situe à la verticale d'une phase de développement, cela signifie qu'elle est (essentiellement) mise en œuvre lors de cette phase même si elle peut utiliser des éléments des phases précédentes, comme par exemple une méthode de test qui utiliserait la spécification du système sous test (noté par la suite *SUT* pour *System Under Test*). Comme rien n'est simple, nous avons choisi une coloration en dégradé en cas de frontière floue, signifiant que la technique est (ou peut être) utilisée à cette phase de développement, mais de façon moins significative. Concernant les étapes de développement, la phase de *spécification* regroupe les étapes de rédaction des exigences, de design, et de modélisation du *SUT*. C'est au cours de cette phase que le cahier des charges initial, souvent décrit en langage naturel, va évoluer vers une spécification détaillée. Parfois cette dernière peut être décrite à l'aide d'un langage de description formel. Dans ce cas, il est possible de procéder à la validation de cette spécification. Ce procédé est communément appelé *vérification* : on s'assure que la spécification vérifie certaines propriétés, qui peuvent être décrites sous forme de logique par exemple. A la fin de cette étape, tous les éléments permettant le codage sont disponibles, notamment l'architecture, les interfaces, et les structures de données. La phase de *codage* correspondant à l'écriture du code source. Il s'agit généralement

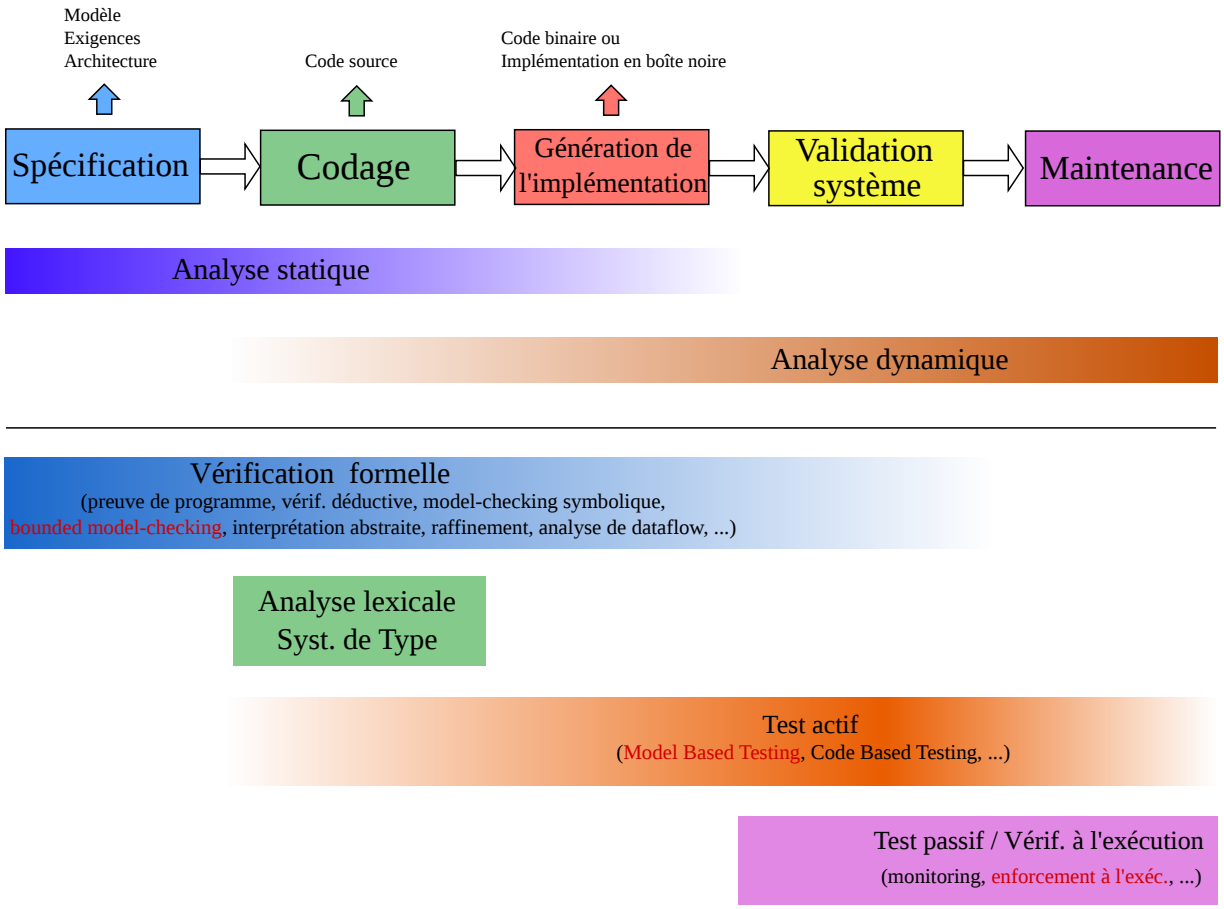


Figure 1.1 – Positionnement des principales méthodes de vérification / test (et de mes contributions (en caractères rouges sur la figure))

d'une écriture de code "à la main", mais il est aussi possible d'utiliser des outils de génération automatique de code. La phase de *génération de l'implémentation* correspond à la compilation du code source vers un code binaire. A ce stade, selon les méthodes de vérification considérées, ce dernier peut être vu comme un code (pouvant être analysé), ou comme une boîte noire, permettant notamment un test *Hardware In the Loop* (*HIL*, par opposition à *Software In the Loop (SIL)*). La phase de *validation système* correspond à la validation du *SUT* dans sa globalité. On l'appelle généralement phase de *test*, dans laquelle on s'assure que l'implémentation satisfait un certain nombre de propriétés de la spécification. Enfin, la phase de *maintenance* correspond à la vie opérationnelle du *SUT* après sa mise en service. La première grande distinction habituelle en vérification et test, c'est la séparation entre les méthodes d'*analyse statique* et *dynamique* (que la communauté de test appelle aussi respectivement *test statique* et *test dynamique*). Par dynamique, on entend que le programme est réellement exécuté, ce qui n'est pas le cas pour les méthodes statiques. Sur la figure 1.1, l'analyse statique déborde sur la phase d'implémentation, car il est possible d'analyser statiquement du code binaire. De même, l'analyse dynamique déborde sur la phase de codage, car la pratique de certains tests (e.g. unitaires) est habituelle lors de la programmation. Une méthode très connue d'analyse statique, généralement non formelle, est la revue de code par une équipe de développeurs (ou un outil pour certains aspects), mais on y trouve aussi toutes les méthodes classiques de *vérification formelle* (e.g. *model-checking symbolique, bounded model-checking, preuve de programme, vérification déductive, interprétation*

abstraite, raffinement, analyse de dataflow, etc...), dont certaines seront détaillées au chapitre 3. Les travaux sur les *systèmes de type* trouvent aussi leur place dans cette catégorie. Par ailleurs, le *test* logiciel est une technique dynamique très connue, mais on peut aussi citer le *monitoring* ou l'*enforcement à l'exécution* qui seront étudiés au chapitre 4. Les méthodes de test sont généralement séparées aussi en deux grandes familles, le *test actif* et le *test passif*. Dans le test actif, un testeur [3] interagit avec le *SUT*. Les problématiques de recherche liées à ces travaux consistent notamment à déterminer un échantillon des données d'entrée du programme permettant d'obtenir une confiance "suffisante" envers le *SUT* (e.g. remplir des *critères de couverture* sur le code ou la spécification, déterminer des *classes d'équivalence*, etc...), trouver des solutions pour générer automatiquement ces données, et automatiser le *verdict* du test. En test passif, le testeur n'interagit pas avec le *SUT*, et se contente d'observer l'exécution pour détecter des erreurs potentielles. Dans ce cas, on parle aussi de *vérification à l'exécution*. Le *monitoring* est une technique connue de vérification à l'exécution. En réalité, il existe une distinction essentiellement historique entre vérification à l'exécution et test passif. A l'origine, le test passif est plutôt conçu pour tester des systèmes pour lesquels un test actif s'avère compliqué à mettre en œuvre, alors que la philosophie en vérification à l'exécution est plutôt de s'assurer qu'un système déjà mis en service fonctionne correctement, i.e. scruter les exécutions du *SUT* (via un *moniteur*) pour remonter des alertes en cas de souci (e.g. une propriété de safety non vérifiée). En réalité, les techniques sont assez similaires (mais viennent de communautés différentes). Dans les deux cas, la problématique de l'obtention d'un verdict automatique se pose. Un autre enjeu dans ces travaux de vérification à l'exécution est la façon d'*instrumenter* cette observation. Précisons que ces observations peuvent se faire *online*, i.e. *à la volée* pendant l'exécution du système, ou *offline* i.e. après l'exécution. De plus, si on considère que le moniteur peut effectuer des corrections à la volée, alors on parle d'*enforcement à l'exécution*. En ce qui concerne les méthodes de test actif, il existe une classification célèbre proposée par Tretmans qui les classe selon trois axes : *niveau de détail*, *accessibilité*, et *caractéristique* du test. La figure 1.2 présente une variante proche de cette classification. On y retrouve l'accessibilité, correspon-

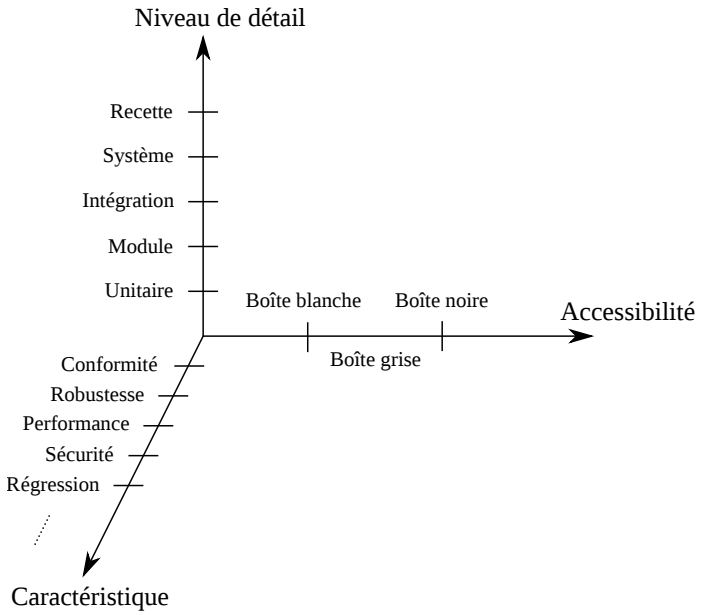


Figure 1.2 – Classification des différents types de test actif (inspirée de Tretmans)

3. Dans ce manuscrit, le terme "testeur" désigne un système informatique, et non une personne.

dant aux éléments auxquels il est possible d'avoir accès pour tester : *boîte noire* en cas d'accès uniquement aux interfaces, *boîte blanche* en cas d'accès sans restriction, et *boîte grise* en cas d'accès partiel (en général, on classe dans cette catégorie tout ce qui n'est ni blanc ni noir). Le niveau de détail correspond aux étapes progressives d'intégration des composants logiciels pour les tester. On y distingue entre autres le *test système*, où l'équipe de développement teste le projet dans son intégralité avant remise au client, et le *test de recette*, qui est réalisé par (ou en présence) du client. Enfin la caractéristique correspond au type de propriété ciblée par le test (*conformité, robustesse, régression,* etc...). En ce qui concerne le test, dans la suite de ce document, nous allons nous intéresser plus spécifiquement au *test de conformité*, qui consiste à vérifier qu'une implémentation est bien conforme à sa spécification (en conditions nominales), et au *test de robustesse*, qui peut être vu comme une extension du test de conformité, dans des conditions non nominales (voir chapitre 2). En test de conformité, on distingue deux approches principales, dépendant de la manière dont les données de test sont générées. La première est le *test structurel*, dont les données de test sont générées en utilisant la "structure" du *SUT*, généralement le code source. Dans ce dernier cas on le nomme *test à base de code*. La deuxième est le *test fonctionnel*, i.e. les données de test sont générées à partir de la spécification du *SUT*. Si cette spécification est décrite sous forme de modèle, alors on parle de *test à base de modèle* (ou *MBT* pour *Model Based Testing*), qui fera l'objet du chapitre 2. Précisons que pour ces deux catégories de méthodes de test (structurel ou fonctionnel), le résultat doit toujours être comparé à la spécification du *SUT* pour obtenir un verdict. Il est assez fréquent dans la littérature de voir une confusion entre test fonctionnel et test boîte noire, ou test structurel et test boîte blanche. En réalité, un test en boîte noire signifie qu'on fait l'hypothèse qu'il n'est pas possible d'accéder à la "structure" du *SUT* (en général le code), et que le testeur accède uniquement aux interfaces, contrairement au test boîte blanche. La confusion est compréhensible, car le fait de considérer le système comme une boîte noire implique généralement de devoir utiliser la spécification du *SUT* pour générer des tests, mais il s'agit bien de deux notions différentes. Le lecteur pourra trouver une classification des différentes méthodes de test actif, ainsi que des explications, dans la présentation que j'ai effectuée à ce sujet à l'école d'été *TAROT2016*[4] [I2]. Un lecteur intéressé par des éléments sur le test à base de modèle pourra aussi consulter la présentation que j'ai effectuée à l'école d'été temps-réel *ETR2011* [I3]. Enfin, le lecteur pourra trouver des éléments concernant la vérification à l'exécution dans le livre tutoriel à ce sujet [97], livre à la rédaction duquel j'ai participé. Les contributions qui seront présentées dans la suite de ce document se rapportent au test à base de modèle, à la vérification formelle de code, et en particulier au bounded model-checking (BMC), et enfin à l'enforcement à l'exécution. Le lecteur pourra les repérer en caractères rouges sur la figure 1.1.

Le chapitre 2 présente nos contributions concernant le test à base de modèle. Ce sont nos travaux les plus anciens. Après une brève présentation concernant les principales méthodes, nous exposons dans un premier temps nos contributions concernant le test de robustesse de systèmes réactifs. Nous y discutons la notion d'aléa, particulière à ce type de test, et nous présentons un framework de test basé sur le principe de spécification augmentée : les aléas décrits sous forme de graphes sont ajoutés de façon incrémentale à la spécification nominale du *SUT*. Cette spécification augmentée est finalement utilisée pour générer des tests et une relation de conformité adhoc est proposée, et enfin quelques études de cas sont brièvement présentées. Dans un deuxième temps, nous présentons nos contributions concernant le test de systèmes temporisés à flot de données. En partant d'un exemple commun à l'ensemble des partenaires (y compris

4. Training And Research On Testing, école issue de projet européen de même nom.

industriels) du projet ANR TESTEC [5], nous proposons un nouveau modèle formel adapté pour décrire ce type de systèmes, le *VDTA (Variable Driven Timed Automaton)* et discutons dans quelle mesure il est possible de définir un framework de test pour ce type de modèle. Ensuite nous présentons brièvement quelques résultats sur le test à distance de systèmes temporisés. En nous basant sur des travaux liés à ce domaine mais non temporisés, nous décrivons comment les adapter tout en utilisant le pouvoir expressif des automates temporisés pour définir une condition suffisante pour garantir la testabilité à distance. *Les travaux décrits dans ce chapitre ont été financés par les projets ANR TESTEC (projet dont j'ai été coordinateur au LaBRI), WEBMOV [6] et VACSIM [7] (projet dont j'ai été coordinateur au LaBRI), et le projet Européen TAROT [8]. Ils sont les résultats de collaborations avec Ismail Berrada, Richard Castanet, Alexandre David, Kim Larsen, Hervé Marchand, Marius Mikucionis, Omer Nguena-Timo (Postdoc au LaBRI à cette période, dont j'ai assuré la direction), Fares Saad-Khorchef (en thèse au LaBRI à cette période, dont j'ai assuré le co-encadrement), Sébastien Salva et ont engendré les publications [C22, J4, C20, C18, C16, C17, C15, C10, C9].*

Le chapitre 3, nettement plus court que le précédent et le suivant, présente nos contributions autour du model-checking de code *C*, en utilisant notamment la programmation par contraintes. Après quelques rappels sur le (bounded) model-checking, nous décrivons en particulier la stratégie de recherche dynamique en *backjump* permettant d'optimiser la vérification de propriétés dans le code, et discutons son efficacité, en comparaison à l'outil *CBMC*, sur une étude de cas avec des propriétés temps-réel fournie par un industriel : le *FlashManager*. Notons que nous avons choisi d'isoler ce chapitre pour des raisons essentiellement thématiques. *Ce travail a été financé par le projet ANR TESTEC. Il est le résultat d'une collaboration avec Hélène Collavizza, Le Vinh Nguyen, Olivier Ponsini et Michel Rueher et a généré la publication [J5].*

Le chapitre 4 synthétise nos contributions autour de l'enforcement à l'exécution. Il s'agit de nos travaux les plus récents. Comme évoqué précédemment, ce chapitre est nettement plus détaillé que les autres afin de montrer au lecteur la difficulté inhérente à ces problèmes de modélisation, et les choix qu'il a fallu faire pour les résoudre. Il a été rédigé à la manière d'un article tutoriel détaillant nos contributions sur le sujet. Après un rappel sur les principes liés à l'enforcement à l'exécution, nous y décrivons progressivement comment synthétiser un moniteur d'enforcement de propriétés régulières temporisées, en partant de propriétés de *safety*, pour arriver finalement aux propriétés régulières quelconques, et ce à différents niveaux d'abstraction. Ensuite, nous nous intéressons au cas particulier des événements incontrôlables, i.e. des événements qu'il est possible uniquement d'observer, mais pas de modifier. Dans un premier temps nous expliquons dans un contexte non temporisé comment l'apport de la théorie des jeux permet de simplifier le problème, puis nous présentons les grandes lignes pour étendre ces résultats dans un cadre temporisé. *Les travaux décrits dans ce chapitre ont été financés par les projets ANR VACSIM, et un projet co-financé par la région Aquitaine, Bordeaux INP, et le GIS Albatros. Ils sont le résultat de collaborations avec Yliès Falcone, Thiérry Jéron, Hervé Marchand, Omer Nguena-Timo (à cette période postdoc au LaBRI dont j'ai assuré la direction, puis chercheur au CRIM à Montréal), Srinivas Pinisetty, et Matthieu Renard (en thèse au LaBRI à cette période, dont j'ai assuré la co-direction) et ont engendré les publications suivantes : [C25, C24, C23, J6, C21]. De plus je suis impliqué dans l'action Européenne*

5. http://www.agence-nationale-recherche.fr/Projet-ANR-07-TLOG-0022
6. http://webmov.lri.fr/
7. http://vacsim.inria.fr
8. http://tarot.it-sudparis.eu/

COST ARVI[9] *(Runtime Verification beyond Monitoring) regroupant 27 pays, dont l'objectif est de mettre en place une expertise dans le domaine de la vérification à l'exécution, de réfléchir aux possibilités et challenges dans ce domaine, et de le promouvoir. J'y représente la France au Management Committee. Une des contributions de cette action est un livre tutoriel sur la vérification à l'exécution. J'ai participé à la rédaction d'un chapitre de ce livre [B1], en collaboration avec Yliès Falcone, Leonardo Mariani et Saikat Saha. Dans la suite de ce document, je ferai parfois référence à cet ouvrage.*

Chaque chapitre commence par une brève justification de l'intérêt de la thématique abordée, situe nos travaux par rapport à d'autres (précédents ou postérieurs), et fournit quelques perspectives liées à cette thématique. Le chapitre 5 résume brièvement les contributions de ce document, et fournit quelques pistes de projets de recherche.

9. `http://www.cost.eu/COST_Actions/ict/IC1402?management`

Chapitre 2

Test à base de modèle

Dans la grande majorité des systèmes informatiques, les spécifications sont décrites sous forme textuelle, en utilisant notamment des *cas d'utilisation*. Le défaut essentiel de cette description, c'est que le langage est intrinsèquement ambigu, pouvant mener à des interprétations différentes. Pour remédier à cela, notamment dans le cas d'applications critiques, il est nécessaire d'utiliser un modèle ou un langage de description plus formel pour décrire la spécification. Ce langage doit être fondé sur des règles sémantiques précises (idéalement être basé sur une sémantique formelle), et peut être normalisé pour faciliter sa diffusion. Concernant les systèmes à entrées-sorties (qui nous intéresserons par la suite), il existe un grand nombre de langages normalisés allant de ceux conçus pour décrire les modèles avec un niveau élevé d'abstraction, comme LOTOS, ESTELLE, SDL, ou encore UML, ou de plus bas niveau, dont l'objectif est plutôt de décrire un système de façon détaillée, notamment sous forme de sémantique formelle, comme les machines à états finis (FSM) ou les systèmes à transitions étiquetées (LTS). Citons aussi le langage B, associé à la méthode B [45], permettant de modéliser de façon abstraite le système, puis de raffiner successivement vers un modèle concret, et sa version événementielle event-B [44]. L'utilisation de modèles pour tester un système permet en général de faciliter l'automatisation du test, notamment son exécution et l'obtention du verdict. En effet, il est généralement possible pour un testeur d'analyser automatiquement le modèle de la spécification pour anticiper le résultat attendu. Comme nous l'avons vu précédemment, le *test à base de modèle* (*Model Based Testing*, noté *MBT* par la suite) est une variante du test fonctionnel (i.e. les données de test sont générées en se basant sur la spécification), mais où la spécification est décrite via un modèle pouvant être formel. Les données de test étant générées à partir de la spécification, la connaissance du code de l'application n'est pas nécessaire. C'est donc un test en *boîte noire*. Par la suite, nous nous focaliserons sur le test à base de modèles *formels*. Cette pratique est assez répandue dans le domaine des applications critiques. Toutefois, dans les autres domaines, elle progresse lentement. La raison, c'est que l'effort de modélisation d'une spécification vers un modèle formel nécessite une grande expertise, et beaucoup de travail. En effet, il est nécessaire de trouver un modèle de description adapté. Si ce dernier est trop abstrait, la spécification ne sera pas réaliste, et aura peu d'intérêt. S'il est trop concret, la modèle sera difficile à lire, et le nombre de cas à vérifier risque d'être trop grand. Ensuite, il faut bien entendu construire le modèle en lui-même, ce qui peut prendre un certain temps. Cependant, généralement les entreprises qui ont effectué cette démarche admettent que l'investissement initial de la modélisation permet un gain significatif sur l'effort de test, et donne au final un bilan très positif. On peut citer notamment l'entreprise Microsoft qui a décidé de modéliser la plupart

de ses protocoles de communication il y a quelques années, et dont le bénéfice est discuté par Grieskamp dans [117]. Un autre intérêt d'utiliser une spécification formelle, c'est qu'il est alors possible d'utiliser des techniques de vérification comme le model-checking directement sur le modèle. Ainsi, il est possible de détecter certaines erreurs (e.g. des deadlocks) avant même de commencer à implémenter le système.

Ce chapitre présente les travaux auxquels j'ai participé autour du *MBT*. Nous y présentons une proposition de framework de test de robustesse pour des systèmes modélisés par des systèmes de transitions à entrées-sorties (ioLTS). Nous décrivons comment intégrer les différents aléas dans la technique de test en enrichissant une *spécification nominale* pour obtenir une *spécification dégradée*, et comment générer des cas de tests à partir de ce nouveau modèle, permettant ainsi de se focaliser sur la robustesse, et nous discutons brièvement les études de cas associées. Nous proposons aussi dans ce chapitre une approche de test basée sur un nouveau modèle temporisé, adapté pour décrire les systèmes temporisés à flot de données qui sont des systèmes ayant des contraintes de temps et interagissant de façon continue avec leur environnement. Partant de ce nouveau modèle, nous décrivons comment adapter les techniques de test temporisées existantes à ce dernier, d'une part en générant des tests de façon non déterministe (*à la TorX* [212]), ou à base d'objectif de test (*à la TGV* [132]). Nous abordons aussi le problème du test à distance, i.e. de la prise en compte de la latence, dans le cadre d'un test temporisé. Nous discutons comment le fait d'utiliser un modèle temporisé peut être utilisé pour intégrer la latence dans le processus de test, et proposons une sémantique à base de FIFO permettant de faciliter l'écriture de la relation de conformité. Nous présentons aussi une condition suffisante sur un automate temporisé, la Δ-*testabilité*, permettant de garantir l'absence d'entrelacement dû aux problèmes d'observabilité lors d'une session de test.

Ces travaux ont été effectués entre autres dans le cadre des projets ANR TESTEC, WEB-MOV et VACSIM et du projet Européen TAROT. Ils sont le résultat de collaborations avec Ismail Berrada, Richard Castanet, Alexandre David, Kim Larsen, Hervé Marchand, Marius Mikucionis, Omer Nguena-Timo, Fares Saad-Khorchef, Sébastien Salva et ont généré les publications suivantes : [C22, J4, C20, C18, C16, C17, C15, C10, C9] [1].

L'organisation de ce chapitre est la suivante :

— La section 2.1 rappelle quelques généralités concernant le *MBT*.
— La section 2.2 présente nos résultats concernant le test de robustesse de systèmes réactifs. Cette section est essentiellement issue des travaux de thèse de Fares Saad-Khorchef [197], dont j'ai assuré le co-encadrement.
— La section 2.3 décrit nos contributions à propos de test de systèmes temporisés à flots de données. Cette section est essentiellement issue des travaux de postdoc d'Omer Nguena-Timo, dont j'ai assuré la direction.
— La section 2.4 présente nos contributions concernant le test à distance de systèmes temporisés. Cette section est aussi essentiellement issue des travaux de postdoc d'Omer Nguena-Timo, dont j'ai assuré la direction.

1. Noter que [C17] est une version réduite pour respecter les restrictions de pages de la conférence. Quand nous le citerons, nous ferons en réalité allusion à la version longue correspondante, et disponible à l'adresse `https://hal.archives-ouvertes.fr/hal-00503000/`.

2.1 Généralités sur le *MBT*

Le *MBT* est donc une technique qui consiste à utiliser un *modèle* de la spécification (que l'on appellera par la suite simplement la *spécification*) comme base pour générer des données de test. Comme dans toutes les méthodes de test décrites au chapitre 1, ces données sont appliquées sur l'implémentation, et le résultat est comparé à la spécification. C'est un test actif. De nos jours, il existe un grand nombre de méthodes et d'outils plus ou moins anciens qui se basent sur ce principe [132, 212, 34, 191, 157, 147, 62, 152, 143, 115, 65, 7, 136, 130] [2]. Une étude très complète (mais un peu ancienne) des différents modèles formels et des méthodes de test associées pourra être trouvée dans Hierons et al. [126]. Le livre tutoriel sur le MBT de systèmes réactifs de Broy et al. [40] constitue aussi un bon point de départ pour obtenir des connaissances sur le sujet. Si on se place d'un point de vue test de conformité, l'objectif est de détecter des *fautes* de l'implémentation qui ne sont pas conformes à la spécification. Ainsi, deux entités interviennent dans le raisonnement : la *spécification* et l'*implémentation* ou *IUT* pour *Implementation Under Test* , comme décrit figure 2.1. Ensuite, il faut définir ce que "spécifier" et

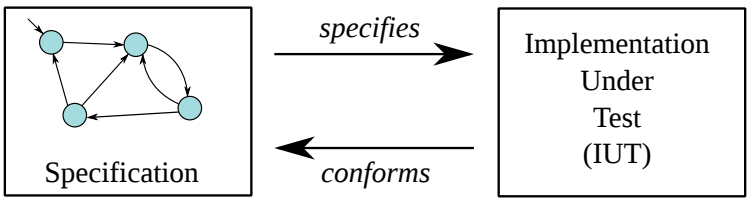


Figure 2.1 – Lien entre une spécification et une implémentation

"être conforme" signifient. Pour cela, on fait l'hypothèse que l'*IUT* est inconnue, mais modélisable par un modèle formel, c'est la fameuse "hypothèse de test", et pour chaque méthode de test, il faut définir une *relation de conformité* entre la spécification et l'*IUT*. Il peut s'agir par exemple d'une relation d'équivalence (e.g. bisimilarité), ou encore d'un préordre comme utilisé dans TorX [212], ou dans TGV [132]. Dans la suite, nous allons nous concentrer sur des systèmes réactifs, i.e. des systèmes réagissant à des stimuli de l'environnement en mode "réflexe". Le principe de ce test boîte noire est résumé figure 2.2. Un *testeur* observe les sorties de l'*IUT*, et envoie des stimuli via des *points de contrôle et d'observation* (PCO), et utilise ces informations pour fournir un verdict. Nous allons rappeler ici quelques approches possibles pour

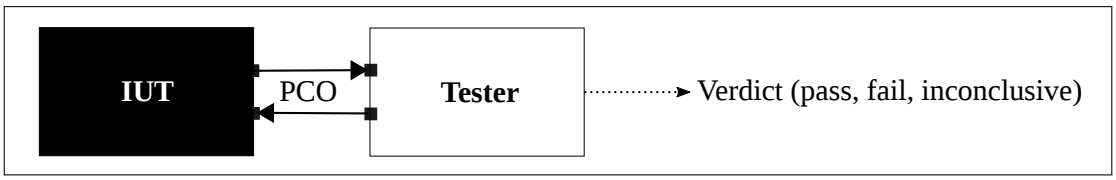


Figure 2.2 – Principe du test boîte noire

tester formellement des systèmes réactifs, mais il en existe bien d'autres. L'objectif ici n'est pas de faire un état des lieux des méthodes de test, mais plutôt de se concentrer sur quelques unes afin d'identifier les problématiques sous-jacentes. Le lecteur intéressé pourra consulter

2. Le lecteur intéressé pourra trouver une liste bien fournie mais non exhaustive d'outils à l'URL `http://mit.bme.hu/~micskeiz/pages/modelbased_testing.html`.

le livre de Broy et al. [40] pour une description plus large. Dans le domaine du *MBT*, différentes tendances se sont développées (provenant de communautés différentes) comme le test à base d'automates [173, 114, 198, 58, 109, 213, 153] qui utilise essentiellement les machines à états finis (ou FSM pour *Finite State Machine*), dont une synthèse est disponible dans [146] et [182], et plus récemment dans [84], ou encore le test fondé sur les systèmes de transitions [211, 132, 210, 38, 2, 174, 184] qui dérive essentiellement des algèbres de processus. Des travaux de test se sont aussi focalisés sur des systèmes décrits par des langages synchrones comme Lustre [191, 34, 201, 157]. L'intérêt de ce type d'approches est que le langage de programmation utilisé possède une sémantique, ce qui permet de le vérifier formellement, et de l'utiliser comme base pour des tests sur des critères aussi bien structurels que fonctionnels. Il permet aussi de prendre en compte certains aspects temporisés dans le modèle. On peut aussi citer aussi quelques travaux de test basés sur UML. En effet, depuis la version UML 2.0, des efforts ont été faits pour définir une sémantique adaptée aux différents diagrammes UML (une étude est disponible dans [165]), permettant ainsi de modéliser la spécification d'un système dans la perspective de le tester [152, 170, 136], ou de décrire les cas de test [214, 1].

Les approches de test à base de machines à états finis utilisent généralement une spécification modélisée par une machine de Mealy [160], i.e. une machine déterministe ayant un ensemble d'états, des alphabets d'entrée et de sortie, et pour un état et une entrée donnés, une fonction de *transition* identifiant l'état suivant, et une fonction de *sortie* identifiant la sortie correspondante de la machine. De nombreuses méthodes ont été développées sur ce principe durant ces quarante dernières années. Le lecteur pourra trouver une description précise de ces différentes méthodes dans [146, 31, 182, 84]. Nous allons rappeler brièvement le principe des plus connues, en nous inspirant de la présentation que j'ai effectuée à l'école d'été *Tarot2016* [I2]. Ces dernières considèrent généralement des machines de Mealy complètes, fortement connexes (si besoin, existence d'un *reset*) et minimales au sens usuel du terme (cf [40]). Ce problème du test revient en réalité pour une spécification sous forme de machine de Mealy M_S connue, et une *IUT* M_I sous forme de machine de Mealy non connue mais observable via des entrées-sorties, à générer une séquence pour vérifier si M_I est conforme à M_S, i.e. équivalente (plus précisément isomorphe) à M_S. Selon les méthodes, certaines hypothèses supplémentaires peuvent être nécessaires, comme l'égalité du nombre d'états entre M_I et M_S. Vérifier l'équivalence entre M_I et M_S revient à vérifier que M_I ne comporte pas de *fautes*, selon un modèle de fautes précis. Les plus classiques sont les *fautes de sortie* (pour une entrée donnée la sortie n'est pas celle attendue), ou les *fautes de transfert* (pour une entrée donnée l'état d'arrivée n'est pas celui attendu). Ces méthodes utilisent généralement le même algorithme de base :

pour chaque *état s et chaque entrée i (de la spécification)* **faire**
aller en s
appliquer i, et vérifier la sortie o (en comparaison à M_S)
vérifier l'état d'arrivée
fin pour

Pour vérifier l'état d'arrivée, il est possible d'utiliser des séquences particulières, notamment :

— une séquence de *distinction* (*DS*) : si on applique cette séquence sur deux états distincts, les sorties obtenues doivent être différentes (méthode DS [114])

— une séquence *UIO* : pour un état donné s, il existe une séquence UIO_s permettant de le distinguer de tous les autres (méthode UIO [198]).

— un ensemble de séquences *W*, dans lequel il existe nécessairement une séquence x_{ij} permettant de distinguer deux états s_i et s_j (méthode W de Chow [58]).

Précisons que les séquences *DS* et *UIO* n'existent pas toujours. En revanche l'ensemble *W* existe toujours pour une machine minimale. Les méthodes utilisant (plus ou moins) ce principe sont nombreuses. Leur objectif est d'obtenir la séquence de test la plus courte possible. On peut citer la méthode *TT* dont le but est de trouver une séquence minimale parcourant l'ensemble des transitions de la spécification [3] ce qui revient à résoudre le problème dit du "postier chinois" ; la méthode *UIO*, consistant à trouver une séquence *UIO* pour chaque état, puis à trouver un circuit minimal atteignant chaque état s_i, testant chaque entrée partant de cet état, et appliquant la séquence UIO_{s_j} pour vérifier que l'état d'arrivée est bien l'état s_j attendu. Cela revient à résoudre le problème dit du "postier rural chinois". La méthode *DS* [114] peut être vue comme un cas particulier de la méthode UIO, avec la même séquence *UIO* pour tous les états. Enfin, la méthode *W* combine deux ensembles de séquences pour atteindre chaque état, appliquer chaque entrée par état, et vérifier l'état d'arrivée grâce à l'ensemble *W* que nous avons évoqué ci-dessus. De nombreuses autres méthodes permettant de diminuer la longueur des séquences ou d'obtenir des hypothèses moins contraignantes ont été développées, notamment les méthodes *Wp* [109], *UIOp* [60], *UIOv* [213]. Des extensions sur des modèles plus expressifs utilisant notamment des variables [183, 135, 216] ou encore le temps [107] ou les deux [177] ont aussi été développées.

La littérature concernant le test à base de systèmes de transitions étiquetés est riche aussi. Le fait qu'il existe deux approches distinctes s'explique plus par des raisons historiques que scientifiques, les similitudes entre les deux étant nombreuses [4]. Toutefois, l'approche à base de systèmes de transitions considère à l'origine des hypothèses moins restrictives sur les modèles. Par exemple, ces dernières sont généralement adaptées pour des modèles non déterministes, ou encore le nombre d'états de l'implémentation n'a pas d'incidence sur la correction du test. Encore une fois, nous n'allons pas décrire l'ensemble de ces méthodes, mais nous allons nous focaliser sur celles qui nous ont inspirés dans nos travaux. Le lecteur pourra trouver une description de ces approches dans [37, 40]. Les premières approches de test sur ce genre de modèles cherchaient surtout à définir des relations de conformité adaptées entre le spécification et l'*IUT*. L'idée sous-jacente était de définir une relation de conformité moins restrictive que l'équivalence, en utilisant notamment des préordres [5]. Ainsi, des relations comme les *préordres de test* [2, 174] ou la relation **conf** de Brinskma [38] ou encore les *préordres de refus* [142] ont été proposées. Comme il s'agit ici d'un test actif, il est utile de distinguer les actions provenant du testeur d'une part et celles provenant de l'*IUT*. C'est ce que proposent notamment Phalippou [184] et surtout Tretmans [210] qui définit la relation **ioco**, qui servira de référence pour de nombreux travaux par la suite. C'est sur cette base que Jard et Jéron proposeront dans TGV [132] une méthode de génération de test à base d'objectifs de test et d'une spécification modélisés sous forme d'ioLTS et qui s'appuie sur la relation **ioco**. Ces travaux ont fortement inspiré les nôtres, c'est pourquoi nous allons donner plus de détails ci-dessous concernant cette

3. Il est possible de considérer que cette méthode se base sur le schéma général décrit précédemment si on considère l'hypothèse que chaque état peut être identifié via une sortie spéciale *status*. C'est l'hypothèse faite dans [40]. Cette hypothèse n'est pas réaliste mais permet une classification plus intuitive des méthodes de test. En réalité, la méthode *TT* ne permet simplement pas de détecter les fautes de transfert.

4. Si on considère les travaux récents, il devient même assez difficile de distinguer les deux approches.

5. Relation binaire réflexive et transitive.

relation, et la génération de cas de test associée. Comme dans les cas du test à base de FSM, ces approches ont fait l'objet de nombreuses extensions, notamment en y ajoutant la gestion des variables grâce à une approche symbolique [62] ou le temps, en utilisant des modèles dérivés des automates temporisés [143, 24, 141, 175, 39, 166, 125, 26, 32]. Le lecteur pourra trouver une comparaison de certaines de ces différentes méthodes dans l'article de Schmaltz et Tretmans [199].

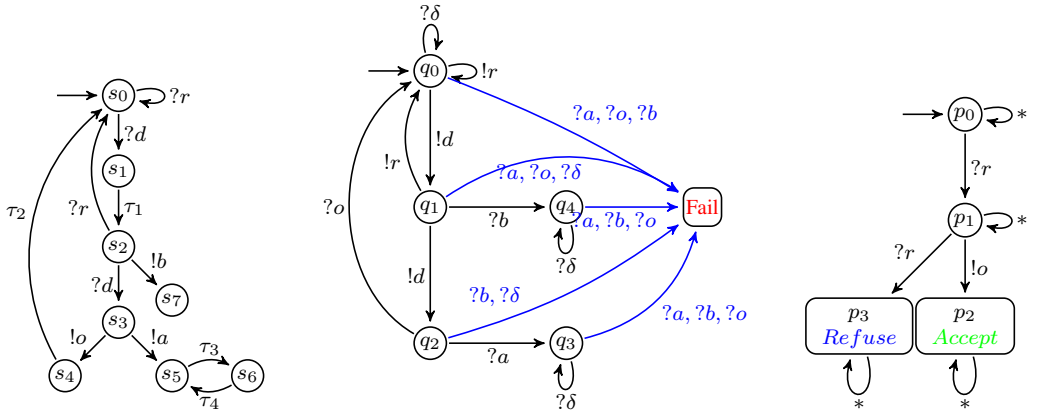
Nous rappelons quelques notations usuelles. Le lecteur habitué à ces formalismes pourra aisément passer cette partie.

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

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

On définit $\Sigma_{VIS}^M = \Sigma_I^M \cup \Sigma_O^M$ *l'ensemble des actions visibles.*

La figure 2.3a donne un exemple d'ioLTS. Ce dernier nous servira de spécification, notée S par la suite.



(a) Une spécification S sous forme d'ioLTS

(b) testeur canonique $Can(S)$

(c) Un exemple d'objectif de test OT

Figure 2.3 – Quelques éléments utilisés dans les méthodes de test à base d'ioLTS

Les notations utilisées ici sont définies dans [132]. Pour ne pas alourdir, nous ne les redéfinissons pas toutes ici. Le lecteur pourra aussi trouver un tutoriel complet dans le cours que j'ai effectué à l'école *ETR11* [I3]. On considère de façon usuelle un *Run* comme une alternance d'états et d'actions tirables entre ces états, une *Trace* comme la projection d'un *Run* sur les actions visibles, et pour un état P, P *after* σ l'ensemble des états atteints à partir de P en observant

σ (cette notion s'étend naturellement à un ensemble d'états). Sur la figure 2.3a, nous obtenons ainsi : $s_0 \xrightarrow{?d} s_1 \xrightarrow{\tau_1} s_2 \xrightarrow{?d} s_3 \xrightarrow{!o} s_4 \in Runs(S)$; $Traces(S) = \{\varepsilon, ?d, ?r, ?d.?r, ?r.?d, ?d.!b, ...\}$; et $\{s_2\}$ *after* $?d.!o = \{s_0, s_4\}$, $\{s_0\}$ *after* $?d, !a = \emptyset$. On définit aussi S *after* $\sigma = \{q_0\}$ *after* σ. Pour la suite, pour un ioLTS M, on notera $det(M)$ la déterminisation de M au sens de [132]. Parfois il est possible que l'*IUT* se bloque. Dans ce cas, il ne s'agit pas forcément d'une erreur car il est possible que ce blocage soit spécifié. La théorie de Tretmans prend en compte ces aspects. En réalité il existe plusieurs sortes de blocages, qui correspondent à des états d'un ioLTS. Il peut s'agir d'un *deadlock* (un état puits), d'un *outputlock* (le système attend une action de l'environnement), ou d'un *livelock* (boucle d'actions internes)[6]. Il s'agit des états s_0, s_5, s_6 et s_7 sur la figure 2.3a. De façon générale, un état ayant une de ces caractéristiques est appelé *quiescent*. Souvent en pratique, le testeur les repère par un *timeout*. D'un point de vue formel, les blocages sont considérés comme une sortie particulière, notée δ, qui sera rajoutée à la spécification. Le fait de les exprimer explicitement permet notamment de les conserver en cas de déterminisation. Pour un ioLTS S, l'ioLTS obtenu après ajout des états quiescents est appelé *automate de suspension*, et noté $\Delta(S)$. Il consiste simplement à ajouter une boucle étiquetée δ sur chaque état quiescent. Finalement, étant donnés un ioLTS S connu correspondant à la spécification, et un autre ioLTS IUT complet en entrée et non connu, représentant une *IUT*, la relation **ioco** est définie ainsi [211] :

$$IUT \textbf{ ioco } S \;=\; \forall \sigma \in Traces(\Delta(S)), Out(\Delta(IUT) \;\; after \;\; \sigma) \subseteq Out(\Delta(S) \;\; after \;\; \sigma)$$

où $Out(P)$ représente l'ensemble des sorties (y compris δ) partant d'un état P. L'idée sous-jacente, c'est qu'une *IUT* est conforme à sa spécification S si après toute trace de suspension de S, les sorties de l'*IUT* (blocages compris) sont incluses dans celles de la spécification. L'objectif de ces travaux est de générer des cas de test que le testeur utilisera ensuite pour guider les tests. Il s'agit d'un ioLTS équipé d'états particuliers, appelés *verdicts*, qui peuvent être **Fail** (le test est un échec), **Pass** (le test est un succès) ou encore **Inconc** (le test n'est pas concluant). Ainsi, d'un point de vue théorique, exécuter un test revient à effectuer la composition parallèle avec synchronisation sur les actions visibles communes entre le cas de test et l'*IUT* : si une exécution de cette composition mène à **Fail**, alors le test est un échec. Les propriétés usuelles recherchées pour un ensemble TS de cas de tests sont : le *non-biais* (*soundness*) : si une *IUT* est conforme, il n'existe pas de cas de test de TS menant à **Fail** ; l'*exhaustivité* : si une *IUT* est non conforme, il existe au moins un cas de test de TS menant à **Fail** ; et la *complétude* : TS est non-biaisée et exhaustive. Un ensemble de cas de test doit toujours être non biaisé. En revanche, il est généralement impossible d'obtenir un ensemble de cas de tests qui soit exhaustif. On cherche donc à définir une méthode de sélection de test permettant d'obtenir un suite de tests *exhaustive à la limite*, i.e. l'ensemble de tous les cas de test qu'il est possible de générer à partir de cette méthode est exhaustif.

Tretmans propose dans [211] un algorithme récursif simple pour générer des cas de test à partir de la spécification S. C'est l'approche utilisée dans TorX [212]. L'idée générale de cette fonction est la suivante (on se placera du point de vue du testeur, i.e. une sortie (resp. entrée) de la spécification sera considérée comme une entrée (resp. sortie) pour le testeur) :
A chaque étape, le testeur choisit (aléatoirement) entre les trois possibilités suivantes, en supposant que S soit à l'état P :

— *s'arrêter et émettre le verdict* **Pass**

6. Cette notion n'existait pas initialement dans les travaux de Tretmans, elle a été ajoutée par Jard et Jéron dans [132].

— *émettre une entrée x vers l'IUT (si possible), et rappeler récursivement cette fonction sur P after x*

— *écouter les sorties de l'IUT et exprimer un verdict* **Fail** *en cas de sortie non autorisée. En cas de sortie autorisée y, rappeler récursivement cette fonction sur P after y*

Pour caractériser de façon simple les comportements autorisés ou non, il est possible d'utiliser un *testeur canonique* : en partant de S, on construit l'ioLTS le plus général possible permettant de détecter la non conformité d'une implémentation par rapport à S. Le testeur canonique correspondant à S est donné figure 2.3b (de façon usuelle, on se place du point de vue du testeur, ainsi, les entrées précédées d'un "?" deviennent des sorties précédées d'un "!" et réciproquement). Cette méthode de génération bien que très simple, permet d'obtenir une suite de test *complète*, i.e. *non-biaisée* et *exhaustive*. Cependant, il est aisé de constater que cette méthode permet difficilement de cibler un comportement particulier de la spécification. C'est pour cette raison que Jard et Jéron proposent dans TGV [132] une méthode de génération à partir d'un *objectif de test*, lui-même décrit sous forme d'ioLTS. Pour cela, il faut d'abord exprimer les comportements ciblés à l'aide d'un objectif de test (OT), afin de limiter l'exploration de la spécification. L'OT est un ioLTS complet équipé d'états Accept décrivant les comportements que l'on souhaite privilégier, et d'états Refuse servant à décrire les comportements que l'on ne souhaite pas tester [7]. Un exemple d'OT est donné figure 2.3c. Celui-ci stipule que l'on souhaite vérifier qu'après l'entrée d'un $?r$, le système va bien émettre un $!o$. Toutefois, nous ne souhaitons pas tester les comportements où une deuxième entrée $?r$ serait appliquée avant l'émission de $!o$. Les étapes successives pour générer une suite de test sont les suivantes :

1. construction de l'automate de suspension de S puis déterminisation $(det(\Delta(S)))$
2. construction du produit synchronisé visible $PS^{VIS} = Can(S) \times OT$, permettant d'intégrer à la fois les états Accept et de donner un verdict *Fail* en cas de comportement erroné.
3. élagage
4. contrôlabilité (i.e. retirer les conflits entre entrées/sorties), et sélection de test

Remarquons qu'avec cette approche, il est possible d'obtenir un verdict **Inconc**. C'est comme cela que les branches élaguées sont marquées, et cela correspond en réalité aux exécutions que l'on ne souhaite pas (continuer à) tester. Comme dans la méthode précédente, les suites de test obtenues par cette méthode sont aussi *non-biaisées* et *exhaustives* (à la limite).

2.2 Test de robustesse de systèmes réactifs

Généralement, la spécification d'un système décrit son comportement dans les situations normales. Cependant, dans le cas d'un système critique, il arrive que cette spécification soit complétée avec des éléments permettant d'éviter des défaillances en cas de situation stressante. Il s'agit donc d'informations spécifiques liées à la robustesse du système. En pratique, ces informations peuvent être ajoutées bien après la phase initiale de spécification du système, et fournies séparément. C'est en utilisant cette vision que nous allons résumer nos contributions relatives au test de robustesse de systèmes réactifs. Ce domaine a été assez peu étudié dans la littérature

7. Et non les comportements erronés, comme pourrait le laisser penser le mot "Refuse".

en comparaison au test de conformité. Comme nous l'avons vu au chapitre 1, le test de robustesse peut être vu comme une extension du test de conformité mais en ciblant ce dernier sur les entrées non prévues dans une utilisation nominale du système. Si cette définition peut paraître assez claire dans un cadre de test unitaire structurel (par exemple, tester une valeur négative pour une fonction définie sur des entiers positifs, amenant généralement à la levée d'une exception), elle est nettement plus floue s'agissant de *MBT*. Par la suite, nous nous focaliserons sur ce type de test. En réalité, même le terme *robustesse* a fait l'objet d'interprétations assez diverses. Notons que la définition IEEE correspondante laisse une grande liberté d'interprétation : *"degré selon lequel un système ou un composant peut fonctionner correctement en présence d'entrées invalides ou de conditions environnementales stressantes"* [179]. Considérant un réel besoin pratique d'augmenter et de tester la robustesse des systèmes, et devant l'absence de taxonomie commune, un action spécifique du CNRS [49] a proposé la définition suivante de test de robustesse : *"vérifier la capacité d'un système ou d'un composant à conserver un comportement acceptable malgré la présence d'aléas"*. En se basant sur cette idée, nous avons proposé un framework de test de robustesse pour des systèmes modélisés par ioLTS, et plus particulièrement des protocoles de communications. Nous avons notamment proposé une définition formelle de la robustesse, et une méthode de génération de tests. Le principe consiste à enrichir une *spécification nominale* (e.g. la spécification standard d'un protocole) avec des aléas représentables, afin d'obtenir une *spécification augmentée*. Nous avons proposé différentes approches de test de robustesse selon le type d'aléas à considérer, et une relation binaire de robustesse étendant la fameuse relation **ioco** [211]. Ces approches ont été implémentées dans un prototype, ce qui a permis de fournir deux études de cas sur deux protocoles classiques : TCP [C10] and SSL-TLS [C16, C9].

Nous allons tout d'abord rappeler quelques approches liées au test de robustesse. Il s'agit de quelques travaux marquants mais en aucun cas d'une liste exhaustive, d'autant plus que la définition même de test de robustesse est sujette à (large) interprétation. Le lecteur intéressé trouvera un chapitre consacré à ce sujet dans [163]. Une première famille de travaux se concentre sur l'injection de fautes dans le système afin de l'amener dans une situation de stress. C'est notamment le cas de l'outil *Ballista* [79], qui se spécialise dans le test de robustesse pour des systèmes d'exploitation (noté SE par la suite) en utilisant une base de données d'entrées par type, considérées comme particulièrement stressantes. Ce principe aussi utilisé dans *JCrasher* [71], ou encore dans l'outil *Mafalda* [196], qui se spécialise dans l'injection de fautes sur un micro-noyau, soit en interceptant des appels et y inversant des bits de façon aléatoire, soit en simulant des fautes physiques au niveau des données. Sur ce principe d'injection de fautes, on peut aussi citer l'outil *Fuzz* [167] qui comme *Ballista* se focalise sur les SE en intégrant dans les appels systèmes des caractères aléatoires. Plus récemment (travaux postérieurs à nos contributions), des travaux du LAAS se sont concentrés sur l'évaluation de la robustesse d'un système de contrôle de robot, focalisant soit sur la corruption de messages [59], soit sur le robustesse temporelle [188]. On peut citer aussi l'outil récent ASTAA [129] conçu pour tester la robustesse des systèmes autonomes. A l'instar de Ballista, il injecte des valeurs exceptionnelles issues d'une base de données aux interfaces des composants, mais il permet en plus de monitorer des invariants de safety lors de tests. Ces méthodes sont généralement automatisables et plutôt efficaces. Elles sont basées sur des approches non formelles. Cependant, elles se contentent généralement de vérifier que le système ne "crashe" pas, ou qu'il ne répond pas des choses aberrantes, mais elles peuvent difficilement vérifier des comportements fins. Les approches formelles cherchent à définir précisément la (relation de) robustesse pour appliquer une méthode spécifique. C'est

dans cet état d'esprit que se situent nos contributions sur ce sujet. On peut citer notamment l'approche *Stress* [93] qui utilise un modèle de fautes orienté robustesse modélisé par FSM. Ce dernier permet d'identifier des états d'erreur et d'orienter le test vers ces états. L'approche proposée par Fernandez et al. dans [104] suggère de muter la spécification en se basant sur un modèle de fautes, permettant ainsi d'obtenir une *spécification dégradée* intégrant les entrées incorrectes et erreurs éventuelles définies dans ce modèle. Ensuite, les cas de test sont générés à l'aide d'un observateur sous forme d'automate de Rabin permettant d'identifier les traces ne respectant pas la propriété de robustesse souhaitée. Ces approches considèrent implicitement que si un système est robuste, alors il est conforme, ce qui n'était pas le cas dans nos travaux précédents [C2, C4]. Les contributions présentées ici considèrent aussi cette vision de la robustesse. On peut citer aussi quelques travaux plus récents (apparus après nos contributions sur ce sujet) qui utilisent UML pour décrire explicitement les informations liées à la gestion de la robustesse [164, 170] et du test associé.

2.2.1 Classification des aléas

La différence majeure entre le test de conformité et le test de robustesse réside dans la gestion des *aléas*. Dans la littérature sur le sujet, la définition communément utilisée d'un aléa est *"tout événement non attendu par la spécification nominale du système"*, puis elle a été précisée par une *Action Spécifique* du CNRS dans [49] en classant les aléas en fonction de leur situation par rapport aux frontières du système. Ils peuvent donc être *internes* (e.g. buffer overflow, défaillance du processeur, etc...), *externes* (e.g. intrusion, conditions stressantes, etc...) ou *au delà des frontières* du système. Nous nous sommes intéressés à la possibilité pratique d'intégrer des aléas dans le test, et pour cette raison, nous avons étendu cette classification en nous plaçant du point de vue du testeur en boîte noire. Ainsi un aléa peut être (ou non) :

— *observable* : certains aléas peuvent se produire à l'intérieur de l'*IUT*, mais sans être détectés par le testeur. Ainsi, un aléa observable est un événement détectable (e.g. mauvaise entrée, température élevée, etc...)

— *contrôlable* : la possibilité pour le testeur d'activer ou désactiver cet aléa dans le processus de test

— *représentable* : l'aléa est représentable dans le modèle formel utilisé pour spécifier le système. Cette notion est liée à la fameuse *hypothèse de test* évoquée précédemment.

Notons que ces propriétés associées à des aléas dépendent de l'instrumentation du processus de test. Ainsi, selon les outils à disposition, un testeur aura par exemple la possibilité ou non de bombarder une *IUT* d'ondes électromagnétiques, et/ou la possibilité ou pas de mesurer la température à l'aide du capteur approprié. Nous avons choisi de séparer l'observabilité et la représentabilité. En effet, lors d'une session de test, si le testeur décide d'augmenter la température du système, cet aléa est observable mais a priori pas représentable dans un modèle formel classique. Par la suite, pour faciliter la formalisation du processus de test, nous avons considéré un aléa observable et non représentable de façon abstraite, comme un événement binaire (i.e. activé ou pas). Finalement, nous avons analysé les huit combinaisons d'aléas possibles et extrait celles qui ont du sens d'un point de vue test de robustesse. Dans un cadre de test actif, nous éliminons les aléas non contrôlables, car le testeur doit mener l'*IUT* à une situation stressante. Les aléas non observables mais représentables sont aussi éliminés car non pertinents. Au final, nous obtenons le tableau suivant résumant les cas pertinents pour un test de robustesse :

Observabilité	Controlabilité	Représentabilité	Notation
False	True	False	$\overline{O}C\overline{R}$
True	True	False	$OC\overline{R}$
True	True	True	OCR

Cette classification donne malgré tout un panel de possibilités d'aléas très vaste. Cependant, dans nos travaux sur le test de robustesse, nous nous sommes concentrés sur les protocoles de communications, et nous avons choisi d'utiliser une spécification (nominale) sous forme d'ioLTS avec une architecture classique en boîte noire correspondant à la figure 2.2. Nous avons donc proposé une classification d'aléas plus spécifique à ce type de systèmes, sur les entrées et les sorties de l'*IUT*. Il peut donc s'agir (en se plaçant du point de vue de l'*IUT*) :

— d'*entrées invalides* : correspondant à des entrées non connues ou erronées dans la spécification nominale du système. Il peut s'agir par exemple d'une erreur de transmission provoquant l'arrivée d'un message inconnu ou contenant des erreurs. Formellement, on considérera comme entrée invalide toute action d'entrée n'appartenant pas à l'alphabet de la spécification nominale (en réalité, il s'agit d'actions appartenant à un alphabet augmenté privé de l'alphabet nominal).

— d'*entrées inopportunes* : correspondant à des entrées connues, mais pas attendues à cet instant. Il peut s'agir par exemple d'un message dupliqué arrivant finalement alors qu'il n'était plus attendu (phénomène classique sous TCP). Formellement, pour un état q de la spécification, il s'agit de l'alphabet des actions d'entrée de la spécification nominale privé des actions d'entrée apparaissant sur les transitions partant de q

— de *sorties inattendues* : toute sortie non spécifiée dans la spécification nominale. Certaines peuvent être malgré tout acceptables.

Les perturbations physiques ou internes au système sont généralement non représentables. Dans nos travaux, nous avons considéré que si on souhaite appliquer ce type d'aléa, alors il faut ajouter un *contrôleur d'aléas* permettant de simuler ce type de situations, en supposant qu'il existe. D'un point de vue formel, il s'agit simplement d'actions *?start* et *?stop* ajoutées à l'algorithme de test.

2.2.2 Présentation du framework de test de robustesse

Nous décrivons ci-dessous le principe général de notre approche de test de robustesse. Il s'agit ici d'une description intuitive non détaillée formellement. Le lecteur intéressé pourra trouver l'ensemble des définitions et détails dans [C10]. De façon similaire à Fernandez et al. dans [104], nous considérons ici que le comportement à adopter en cas de situation imprévue est décrit séparément du comportement nominal. En pratique, cela peut se traduire par le fait qu'un concepteur souhaite séparer explicitement les aspects liés à la robustesse, ou simplement que ces aspects auraient été omis lors de la conception initiale, et rajoutés après coup. Le principe de cette approche, c'est de partir d'une spécification nominale et d'y intégrer progressivement les aléas (correspondant aux différentes catégories décrites ci-dessus) par composition pour finalement obtenir une *spécification augmentée*, et d'utiliser cette dernière pour générer les cas de test. Dans la suite, nous reprenons les notions habituelles sur les ioLTS décrites précédemment. Pour le test de robustesse, nous considérons une spécification nominale $S = (Q, q_0, \Sigma, \longrightarrow)$ et n ioLTS $HG_k = (Q^k, q_0^k, \Sigma^k, \longrightarrow^k), k \in \{1, ..., n\}$ précisant le comportement du système en

présence d'aléas, appelés *graphes d'aléas*. Ces ioLTS sont en réalité ce que nous avons appelé des *méta-graphes* : ces derniers contiennent des *méta-états* associés aux états de S, et peuvent contenir en plus des états *dégradés* permettant de définir un comportement spécifique en cas de situation inattendue. Ce type de description permet simplement de regrouper des actions similaires sur plusieurs états. Le méthode se décompose en deux étapes distinctes que nous allons détailler par la suite :

1. construction de la *spécification augmentée* S_A intégrant les aléas dans la spécification,
2. génération des cas de test.

La construction de la spécification augmentée est résumée par le schéma de la figure 2.4. Ainsi,

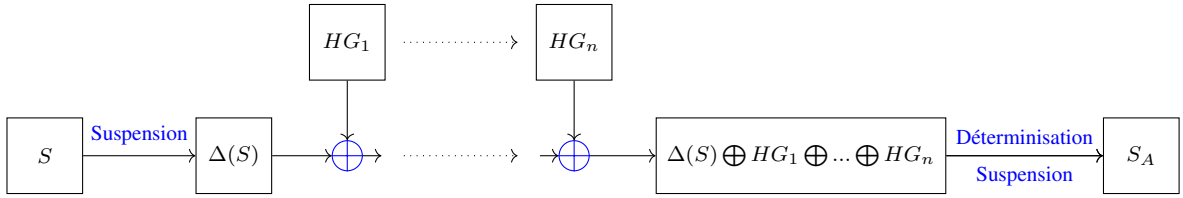


Figure 2.4 – Construction de la spécification augmentée

de façon similaire à l'approche TGV [132], on commence par intégrer les blocages à la spécification pour obtenir l'automate de suspension $\Delta(S)$. Ensuite, on applique une sorte de composition avec chacun des graphes d'aléas fournis (en pratique, on manipule en général un graphe par famille d'aléas). Finalement, il reste simplement à ajouter une nouvelle fois les traces de suspension au modèle (des blocages pourraient avoir été ajoutés lors de la phase de composition), et à déterminiser le tout pour obtenir la spécification augmentée S_A. Au final, S_A contient l'ensemble des comportements à adopter pour les aléas décrits dans les graphes HG_k.

Exemple 1. *La figure 2.5 présente un exemple complet de construction d'une spécification augmentée à partir d'une spécification nominale S définie sur l'alphabet $\{?a, ?b, !x, !y\}$, d'un graphe d'aléas HG_1 défini sur l'alphabet $\{?a, !x, ?a', ?b'\}$, et d'un autre graphe HG_2 défini sur l'alphabet $\{?a, ?b, ?a', ?b'\}$. Les détails concernant cet exemple sont disponibles dans [C10].*

La génération des cas de test reprend le même principe que TGV [132]. A partir de la spécification augmentée S_A et d'un objectif de test de robustesse RTP (pour *Robustness Test Purpose*), suivant les mêmes définitions que dans [132] (notamment le produit synchronisé) et que nous avons décrites précédemment, la génération se décompose selon les étapes suivantes :

1. construction du produit synchronisé $S_A \times RTP$: cette étape permet d'intégrer les états Accept et de donner un verdict **Fail** en cas de comportement erroné ;
2. élagage : supprimer les branches inutiles
3. contrôlabilité et sélection de test : résolution des problèmes de contrôlabilité (le testeur ne peut envoyer qu'une entrée à la fois), et sélection d'un cas de test.

Exemple 2. *La figure 2.6c donne un exemple de cas de test obtenu à partir d'une spécification augmentée S_A (figure 2.6a*[8]*) et de l'objectif de test RTP de la figure 2.6b. Le lecteur trouvera une version détaillée de cet exemple dans [C10].*

8. Spécification différente de celle de la figure 2.5 pour obtenir un nombre d'états plus petit.

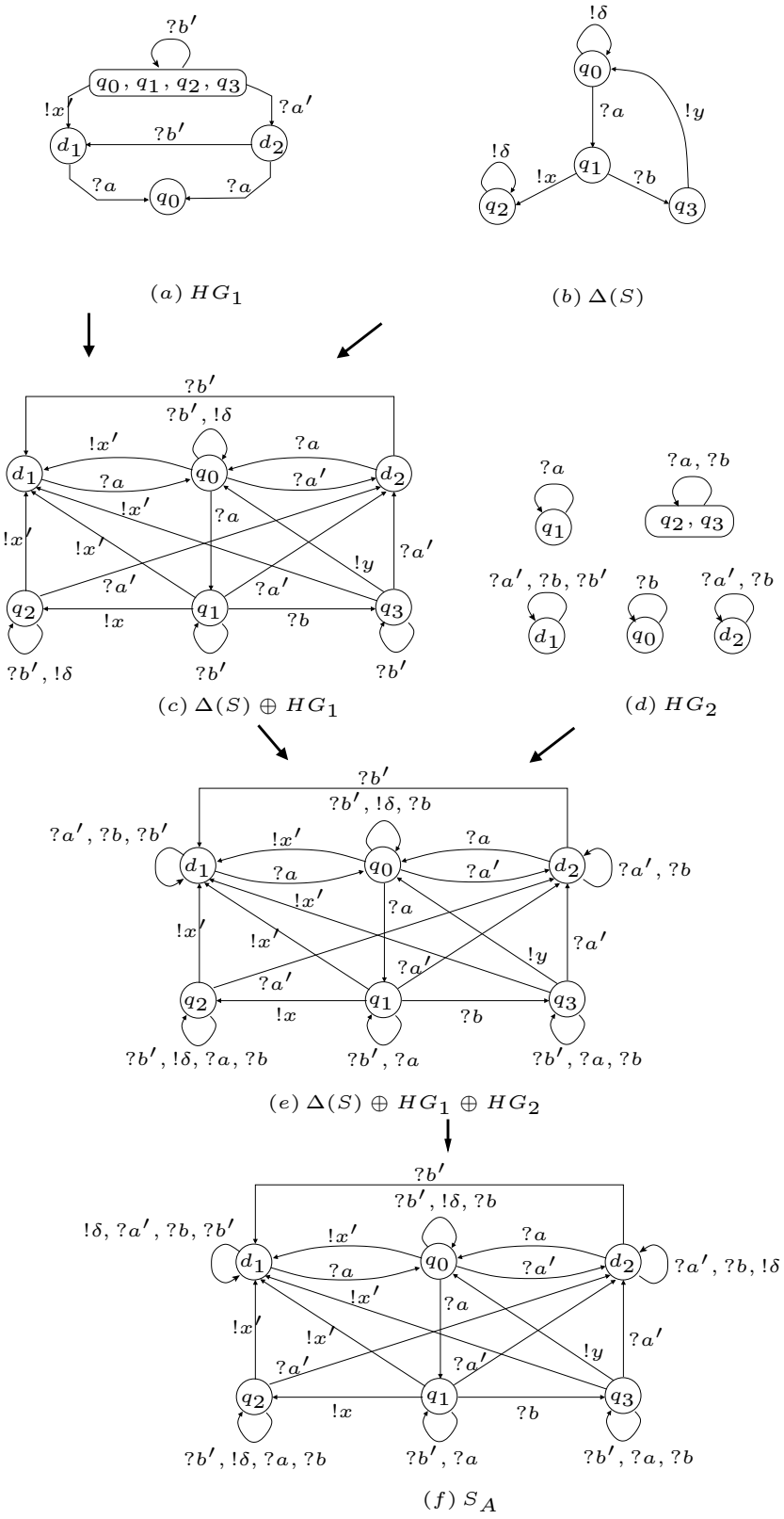
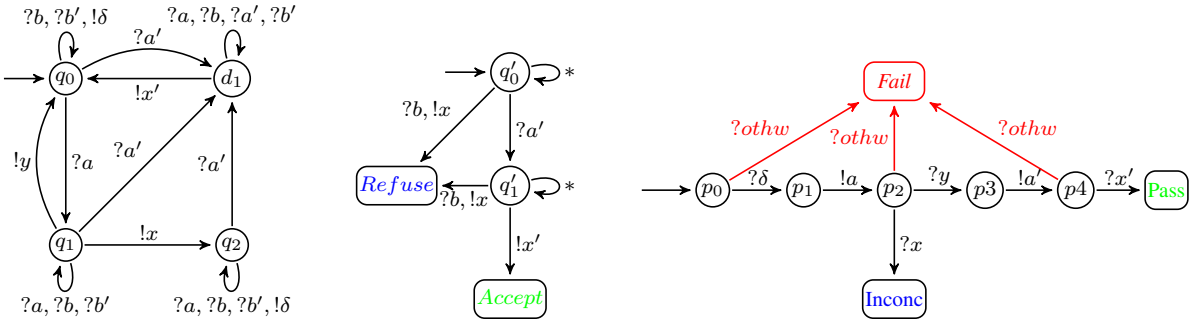


Figure 2.5 – Exemple de construction de spécification augmentée (extrait de [C10])



(a) Spéc. aug. S_A

(b) Un exemple d'objectif de test de robustesse RTP

(c) Exemple de cas de test de robustesse

Figure 2.6 – Génération de cas de test de robustesse

L'algorithme de sélection de cas de test peut être identique à TGV, ou utiliser une approche à base de coloration. Dans ce dernier cas, l'idée est de colorier les transitions ajoutées lors du processus d'intégration des aléas, et de favoriser ces transitions (si possible) lors de la génération des cas de test. Le lecteur intéressé pourra trouver plus de détails dans [C10, C9, 197].

La relation de robustesse entre l'implémentation et la spécification augmentée s'inspire fortement de la relation **ioco** de Tretmans [211], mais cible les aléas. Considérant deux ioLTS S_A et *IUT*, correspondant respectivement à la spécification augmentée et à l'*IUT* supposée complète en entrée et définie sur le même alphabet d'actions que S_A, nous avons défini la relation de robustesse suivante (toujours en reprenant les notations décrites précédemment) :

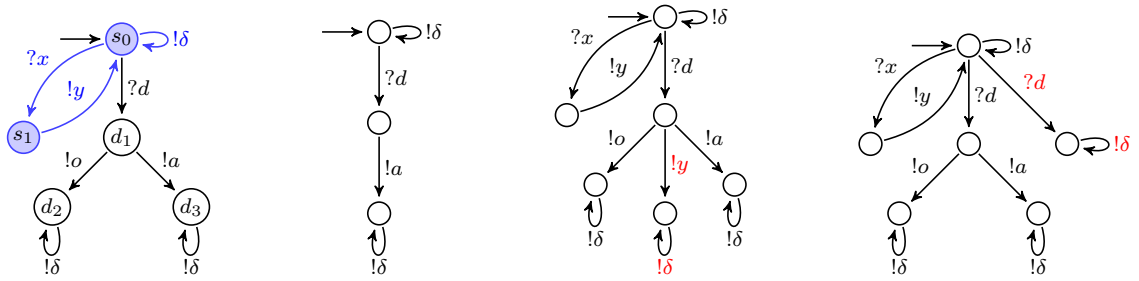
$$IUT \textbf{ robust } S_A \triangleq \forall \sigma \in Traces(S_A) \backslash Traces(\Delta(S)), Out(\Delta(IUT) \textit{ after } \sigma) \subseteq Out(S_A \textit{ after } \sigma).$$

Cette relation se focalise uniquement sur les comportements liés aux aléas. Elle permet de détecter les erreurs de robustesse, même si l'*IUT* n'est pas conforme à sa spécification nominale. Dans nos travaux, nous avons généralement considéré que l'*IUT* était conforme à sa spécification nominale avant d'appliquer le test de robustesse. Dans ce cas, la propriété suivante est vérifiée : IUT **robust** $S_A \wedge IUT$ **ioco** $S \Longrightarrow IUT$ **ioco** S_A. Remarquons que le fait d'utiliser une relation de robustesse spécifique permet de focaliser la session de test sur ces aspects, et ainsi de générer des cas de test adaptés pour détecter des comportements non robustes. Cette approche garantit des cas de test non-biaisés. D'un point de vue théorique, elle assure aussi l'exhaustivité (à la limite) si on considère que l'alphabet de la spécification augmentée inclut bien toutes les entrées et sorties possibles.

Exemple 3. *La figure 2.7 illustre des cas d'implémentations conformes ou non. Les éléments en bleu dans la figure 2.7a représentent la spécification nominale du système.*

- IUT_1 **robust** S_A *car toutes les traces de* IUT_1 *se retrouvent dans celles la spécification augmentée sans la partie bleue, et par exemple* $Out(IUT_1 \textit{ after } ?d) = \{!a\} \subseteq Out(S_A \textit{ after } ?d) = \{!a, !o\}$
- $\neg(IUT_2$ **robust** $S_A)$ *car par exemple* $Out(IUT_2 \textit{ after } ?d) = \{!a, !o, !y\} \not\subseteq Out(S_A \textit{ after } ?d) = \{!a, !o\}$
- $\neg(IUT_3$ **robust** $S_A)$ *car par exemple* $Out(IUT_3 \textit{ after } ?d) = \{!a, !o, !\delta\} \not\subseteq Out(S_A \textit{ after } ?d) = \{!a, !o\}$

Cette technique de génération est directement utilisable pour les aléas OCR, mais aussi pour les aléas $\overline{OCR}$ et $O\overline{CR}$, à condition d'ajouter un contrôleur d'aléas (comme évoqué précédem-



(a) Spéc. aug. S_A (b) IUT_1 (c) IUT_2 (d) IUT_3

Figure 2.7 – relation **robust** par l'exemple.

ment), et d'utiliser un ou plusieurs graphes d'aléas pour vérifier que les sorties obtenues sont acceptables. Par exemple, cette technique s'adapte parfaitement pour tester un système soumis à des perturbations physiques, comme utilisé dans le projet *Homere* [94]).

2.2.3 Outil et études de cas

Dans un premier temps, cette approche a été implémentée à l'aide de l'outil TGSE [24] développé au LaBRI, qui est un outil de test de conformité. Il fallait pour cela construire la spécification augmentée "à la main", et ensuite nous utilisions cette dernière comme spécification entrée à TGSE. Cette approche est détaillée dans [C9, C16]. Ensuite, nous avons développé le prototype *RTCG*, spécialisé pour le test de robustesse, permettant de construire une spécification augmentée à partir d'une spécification nominale et d'un ensemble de graphes d'aléas, et de générer des cas de test de robustesse à partir d'un objectif de test, selon la méthode décrite ci-dessus. Les formats SDL ou DOT sont utilisés pour représenter les modèles en entrée, et les cas de test sont fournis aux formats TTCN-3 [220, 116], XML (utilisant le format défini dans le projet RNRT Calife) ou DOT.

Nous avons proposé deux études de cas. La première a été effectuée sur le protocole SSL-TLS [80] conçu pour la sécurité des communications entre deux applications sur Internet. Lors du SSL handshake, la spécification standard identifie un certain nombre de fautes et leur message d'erreur associé. Cependant, Bradley et al. identifient dans [36] deux messages d'erreur usuellement implémentés, mais ne figurant pas dans la description du protocole. Il s'agit des messages `Unsupported-Authentication-Type-Error` et `Unexpected-Message-Error`. Ce type de message pouvant être à la fois une entrée ou une sortie, nous les avons donc considérés comme des entrées invalides et des sorties acceptables, et construit les graphes d'aléas en conséquence. Nous avons considéré trois objectifs de test : (1) maintien de connexion entre le serveur et le client après la détection d'un certificat non approprié, (2) fermeture de la connexion après la détection d'un problème de chiffrement, (3) maintien de la connexion lors d'un message d'erreur non attendu. Nous avons aussi proposé une étude de cas sur le protocole TCP [187], protocole fiable de la couche transport orienté connexion. Nous avons considéré comme spécification nominale la FSM fournie directement dans la RFC de TCP. Ensuite, nous avons considéré les aléas suivants : message avec des erreurs détectées (e.g. mauvais numéro de séquence), message non attendu à cet état, et émission d'un message *reset* après réception d'un message non spécifié. Ensuite, nous avons proposé des objectifs de test vérifiant que les phases de connexion se déroulaient correctement. Dans ces deux cas d'étude, la spécification augmentée obtenue contenait respectivement 19 et 21 états, et le temps nécessaire à la génération des

cas de test était de l'ordre de quelques milli-secondes [9]. Tous les détails concernant ces études de cas sont disponibles dans [C9, C16, 197, C10].

2.2.4 D'autres contributions, en bref

Au cours du projet ANR WEBMOV, nous nous sommes brièvement intéressés à la possibilité de tester le conformité et la robustesse d'applications de services Web. Nous avons étendu les travaux de Frantzen et al. [108] sur le test de conformité de services Web modélisés par des STS (Symbolic Transition System) en les adaptant dans un contexte de robustesse. Cette vision du test de robustesse se rapproche plus de celle de Ballista [79] en ciblant les erreurs d'interface (e.g. type du paramètre, nombre de paramètres, etc...). Ainsi, nous avons proposé dans [J4] de tester la robustesse de services Web décrits par des ioSTS, une extension des ioLTS enrichie avec des variables internes, et des gardes sur ces variables. Les actions d'entrée sont des opérations typées, et les actions de sortie sont des valeurs typées. Nous avons construit une spécification augmentée considérant des aléas comme des appels d'opérations avec des valeurs inhabituelles, ou encore le remplacement/ajout d'une opération par une autre. Nous avons implémenté cette approche et effectué une étude de cas sur un service d'E-commerce, mettant ainsi en évidence quelques problèmes de robustesse. Les détails de ces travaux sont disponibles dans [J4].

2.3 Test de conformité de systèmes temporisés à flot de données

Dans le domaine du test formel, le choix du modèle de spécification est très important, car il a des conséquences directes sur la stratégie de test à adopter. Généralement, un changement de modèle nécessite de réadapter, voire reconcevoir, la méthode de test utilisée. Certains modèles peuvent ne pas être adaptés pour spécifier un type de systèmes donné. Plus précisément, il existe des systèmes où la description du comportement sous forme de machines à états ou à transitions étiquetées n'est pas assez expressive. C'est le cas notamment des systèmes temps-réel, i.e. des systèmes où la durée nécessaire pour effectuer une action a de l'importance et doit être prise en compte dès la conception. Par exemple, la spécification d'une barrière de passage à niveau devra préciser que cette dernière doit être baissée en un temps inférieur à celui que met le train pour atteindre le passage. Il existe un grand nombre de modèles permettant de décrire des aspects liés au temps. Comme pour les modèles non temporisés, on trouve tout un panel de modèles allant d'un haut niveau d'abstraction, à des modèles plus détaillés, comme les extensions temporisées des modèles LTS ou FSM vus précédemment, souvent enrichis par des horloges. Citons aussi Event-B et ses patterns décrivant les contraintes de temps [46], permettant une description à haut niveau du système, puis de le raffiner progressivement. De nombreux travaux concernant le test de ce type de systèmes ont été proposés dans la littérature. On trouve généralement trois grands types d'approches. Tout d'abord, celles qui considèrent comme spécification des modèles utilisant une représentation discrète du temps, notamment les modèles synchrones [118, 25]. C'est le cas des outils Lutess et Lurette [34, 191] qui permettent de générer des cas de test à partir d'une spécification Lustre [118], un langage synchrone à flot de

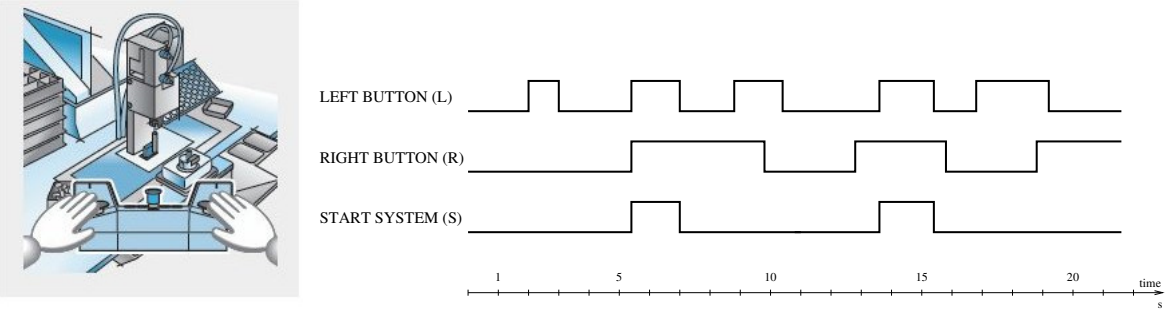
9. Résultat obtenu avec un CPU à 2,80 GHz, et 256 Mo RAM, sous WindowsXP©.

données. Pour cela il est nécessaire de décrire les contraintes sur l'environnement et les propriétés attendues (pour l'oracle). Ensuite, à l'aide d'une description sous forme de BDD [4] et d'un solveur de contraintes, ces outils génèrent des cas de test. Etant donné que le générateur sélectionne aléatoirement les vecteurs d'entrée du test, ces approches sont non-déterministes. Plus récemment, mais dans le même esprit, Mnad et al. proposent dans [169] l'outil *Testium* conçu pour tester les contrôleurs synchrones (connus sous le nom de *Programmable Logic Controllers* ou *PLC*). Il permet de traduire une spécification au format SPTL [168], langage permettant de spécifier des invariants ou des objectifs de test, en un système de contraintes. Ainsi, à l'aide d'un solveur, il choisit aléatoirement des entrées de test à appliquer au contrôleur. L'outil Gatel [157] utilise un principe similaire à Lutess et Lurette (descriptions en Lustre, utilisation d'un solveur de contraintes), mais permet d'utiliser des objectifs de test à la manière de TGV. Ce type de modèle permet généralement une description compacte et intuitive des systèmes, en revanche, il gère le temps de façon discrète, et non de façon continue. Les deux autres catégories utilisent des modèles à temps continu pour décrire la spécification. Il s'agit de variantes des automates temporisés d'Alur et Dill (notés TA pour *Timed Automata* [6] par la suite), i.e. des automates enrichis par des horloges dont la valeur évolue avec le temps. Les transitions sont gardées par des contraintes sur les horloges, et il est possible de remettre à zéro une horloge en passant une transition, ce qui permet de décrire bon nombre de comportements ayant des contraintes de temps. Le deuxième type d'approche étend la théorie des tests à base de FSM. On peut citer notamment [47, 138, 92, 208] qui proposent d'adapter les méthodes à base de caractérisation d'état (décrites à la section 2.1 de ce chapitre) soit sur des TA à la *Uppaal* [20, 6], comme décrit par Cardell-Oliver dans [47], soit sur une représentation finie des TA (avec entrées-sorties) comme le graphe des régions ou de zones [6, 22, 83, 219, 124], comme décrit dans [92, 208]. Le dernier type d'approches consiste généralement à étendre la théorie **ioco** de Tretmans. Ainsi, [143, 24, 141, 175, 39, 166, 125, 26, 32, 177] définissent une nouvelle relation de conformité inspirée de **ioco** sur les TA à entrées-sorties, et proposent une méthode de test adaptée. De façon similaire au cas non temporisé, ces approches peuvent être non déterministes [141, 166, 125, 39, 143] avec un algorithme inspiré de TorX [211, 212], ou encore basées sur un objectif de test qui peut être un critère de couverture [175, 125] ou un automate (temporisé) permettant comme dans TGV de cibler les comportements à privilégier lors du test [26, 24]. C'est ce type d'approche que nous allons utiliser dans la suite de ce chapitre. Des extensions de ces travaux à base de théorie des jeux ont été proposées par David et al. [75] : l'idée est de considérer le test comme un jeu à deux joueurs : le testeur d'un côté, et l'*IUT* de l'autre. Ensuite, guider le test consiste à trouver une stratégie gagnante (pour cela les auteurs utilisent l'outil *Uppaal-Tiga* [48, 76]) pour le testeur, qui peut, soit envoyer une action à l'*IUT*, soit ne rien faire. Nous retrouverons cette idée dans nos stratégies d'enforcement en présence d'événements incontrôlables au chapitre 4, et c'est ce framework de test qui a servi de base dans notre étude de cas de test à distance, que nous présenterons à la section 2.4. Plus récemment, Henry et al. ont aussi proposé dans [121] une approche à base de jeux pour résoudre les problèmes de contrôlabilité dans le test. Là aussi, les cas de test sont définis comme des stratégies de jeu entre le testeur et l'*IUT*, avec pour objectif de réduire la *distance* entre le procédé de test et la satisfaction de son objectif. On peut aussi citer dans cette catégorie les travaux de Nuñez et al. [177] qui proposent une approche assez similaire mais à base de FSM étendue temporisée, i.e. une FSM enrichie non seulement par des horloges, mais aussi par des variables. Notons que lorsque ces approches doivent atteindre un état particulier du TA (ce qui est le cas pour des approches à base d'objectif de test notamment), il est nécessaire d'utiliser une représentation symbolique finie du TA, comme le graphe de régions [6] ou de zones [22, 83, 219, 124],

pour garantir la terminaison du calcul. Par ailleurs, les TA n'étant pas déterminisables dans le cas général, les approches décrites ici considèrent soit que la spécification est déterministe, soit manipulent une sous-classe déterminisable des TA [175, 26]. Le lecteur intéressé pourra trouver une comparaison de certaines de ces différentes approches dans l'article de Schmaltz et Tretmans [199]. Un tutoriel permettant de découvrir les problématiques liées à la gestion du temps dans le test formel pourra être trouvé dans la présentation et son support de cours que j'ai rédigés pour l'école ETR11 [I3].

Lors du projet ANR TESTEC, nous avons cherché à étudier les possibilités de test formel pour des systèmes temporisés réactifs à flot de données, i.e. des systèmes à contraintes de temps et intéragissant de façon continue avec leur environnement sur les entrées-sorties, avec la possibilité à tout moment qu'une ou plusieurs entrées reçoivent un signal, potentiellement simultané. Il est fréquent de trouver ce genre d'applications dans les systèmes à contrôle-commande dans l'industrie. L'exemple ci-dessous donne un exemple de spécification de ce type de modèle [10]. Il s'agit d'une commande *bi-manuelle* qui est un dispositif pour démarrer une machine dangereuse, dont le cahier des charges est :

Exemple 4. *La commande est constituée de deux boutons (gauche (L) et droit (R)) et d'un contrôleur du démarrage et de l'arrêt de la machine (S). Initialement, la machine est arrêtée ; elle démarre lorsque les deux boutons sont appuyés dans un intervalle de temps inférieur à* 1 *seconde. Lorsque qu'un seul bouton est appuyé et que le second n'est pas appuyé* 1 *seconde après le premier, la machine ne démarre pas tant que les deux boutons ne sont pas relâchés. Quand la machine est activée, elle s'arrête dès qu'un bouton est relâché. Le processus de démarrage est réinitialisé lorsque les deux boutons sont relâchés. La figure 2.8a fournit une illustration de ce type de système. Le chronogramme de la figure 2.8b donne un exemple des comportements possibles, en notant* S *la sortie.*



(a) Une machine à commande "bi-manuelle"

(b) Chronogramme illustrant des comportements autorisés

Figure 2.8 – La commande "bi-manuelle".

L'utilisation de TA à entrées-sorties (TAIO, TIOA, TIOTS selon les travaux), comme dans beaucoup de travaux sur le test de systèmes temporisés s'inspirant des travaux de Tretmans, est possible, mais ce modèle n'est pas adapté pour ce genre de systèmes. En effet, dans l'exemple précédent, il serait nécessaire d'utiliser une transition pour le cas ou L et R sont appuyés simultanément, une autre pour le cas où L est appuyé mais pas R, etc... Ainsi, le nombre de transitions nécessaires augmente de façon exponentielle par rapport aux nombres d'entrées dans le système.

10. Il s'agit d'un exemple "jouet" fourni par un partenaire du projet ANR TESTEC.

Nous avons finalement proposé [C15] un modèle formel, le *Variable Driven Timed Automaton (VDTA)* particulièrement adapté pour décrire ce genre de systèmes. Ce dernier s'inspire notamment des modèles synchrones [118, 25], mais surtout du langage à commandes gardées de Dijkstra [82], qui est un modèle à base de variables. Le VDTA reprend aussi le principe d'urgence existant dans *Uppaal* [20], en le rendant systématique, et aussi des aspects du langage IF [35], comme la communication à travers des variables. Cependant, notre sémantique diffère légèrement, notamment en ce qui concerne la notion d'*état stable*. Le VDTA est donc un modèle permettant de décrire de façon concise les systèmes temporisés réactifs à flot de données. Le principe est de placer les variables au centre des communications entre le système et son environnement. Ainsi, les entrées et les sorties du modèle sont des mises à jour de variables : l'environnement peut à tout moment affecter de nouvelles valeurs aux variables d'entrée, et observer les variables de sortie, et le système évolue en fonction de l'état des variables à chaque localité de façon urgente (i.e. dès que les contraintes sont satisfaites). Les contributions présentées ici décrivent le modèle VDTA et sa sémantique. Nous discutons ensuite dans quelle mesure ce modèle peut être utilisé pour générer des cas de tests de conformité en s'inspirant de l'approche de TGV [132] ou de TGSE [24] dans le cas temporisé. Il s'agit de contributions théoriques. Cette partie est un résumé des travaux publiés dans [C15, C17, C18, C20].

2.3.1 Les Variable Driven Timed Automata (VDTA)

D'un point de vue formel, le VDTA est une adaptation des TA, dans laquelle les transitions sont gardées par des contraintes sur les variables d'entrée, les variables de sortie, et les horloges. L'environnement modifie les variables d'entrée, alors que le système est autorisé à modifier les variables de sortie, qui sont toujours observables. Lorsque les contraintes sur une transition deviennent vraies, celle-ci doit être tirée immédiatement. Cela permet potentiellement de franchir plusieurs transitions en temps zéro en cas de mise à jour d'une variable, ce qui n'est en général pas possible dans les modèles basés sur les événements.

Nous donnons la définition exacte de VDTA ci-dessous. Celle-ci utilise certaines notations classiques utilisées dans les TA, et notamment par Alur et Dill dans [6]. Nous considérons ici le lecteur familier avec les TA, cependant ce dernier pourra trouver l'ensemble des notions détaillées dans [C15, C17]. Dans cette définition, étant donné un ensemble fini de variables ou d'horloges Y, on notera $\mathcal{G}(Y)$ l'ensemble des contraintes défini comme la combinaison booléenne de contraintes simples de la forme $Y_i \bowtie c$ avec $Y_i \in Y$, $c \in \mathbb{N}$ et $\bowtie \in \{<, \leq, =, \geq, >\}$, et on étend cette notation sur n ensembles de contraintes : $\mathcal{G}(Y_1, ..., Y_n)$ correspond à la combinaison booléenne d'éléments de $\mathcal{G}(Y_1)$, ... , $\mathcal{G}(Y_n)$. Par ailleurs, pour un ensemble fini de variables V, on notera $A(V)$ les affectations sur V. Voici la définition :

Définition 2 (VDTA)**.** *Un* Variable Driven Timed Automaton *(VDTA) est un 7-uplet* $\mathcal{A} = (L, X, I, O, l^0, G^0, \Delta_{\mathcal{A}})$*, tel que :*

- L *est un ensemble fini de* localités*,* $l^0 \in L$ *est la* localité initiale*,*
- $X = \{X_1, X_2, \ldots, X_k\}$ *est un ensemble fini d'*horloges*,*
- $I = \{I_1, I_2, \ldots, I_n\}$ *est un ensemble fini de* variables d'entrée*,*
- $O = \{O_1, O_2, \ldots, O_m\}$ *est un ensemble fini de* variables de sortie*,*
- $G^0 \in \mathcal{G}(I, O)$ *est la* condition initiale*, une contrainte avec des variables de* $I \cup O$*.*

— $\Delta_{\mathcal{A}} \subseteq L \times \mathcal{G}(I,O,X) \times A(O) \times 2^X \times L$ *est la relation de transition :*
$(l, G, A, \mathcal{X}, l') \in \Delta_{\mathcal{A}}$ *est une* transition *telle que*
  — l *et* l' *sont les localités* source *et* destination *de la transition.*
  — G *est une combinaison booléenne d'éléments de* $\mathcal{G}(I)$, $\mathcal{G}(O)$ *et* $\mathcal{G}(X)$.
  — *Une* affectation de sortie $A \in A(O)$.
  — $\mathcal{X} \in 2^X$ *est l'ensemble des horloges réinitialisées au passage de la transition*

Un exemple de VDTA décrivant la commande bi-manuelle est fourni figure 2.9 (dans cet exemple, toutes les variables sont initialisées à 0, par la suite nous considérons toujours les variables initialisées à 0). Des explications concernant cette figure sont données à l'exemple 5. Un état d'un VDTA est défini par un tuple contenant une localité, les valeurs des variables

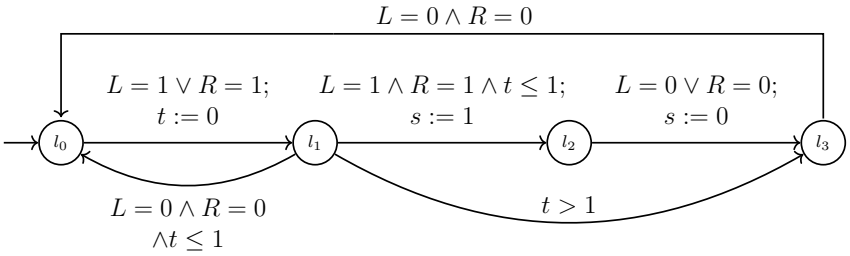


Figure 2.9 – VDTA modélisant la commande bi-manuelle

d'entrée et de sortie, et les valeurs des horloges. Dans un état donné, on peut trouver trois types de situations :

— une transition est franchie de façon urgente car la garde est satisfaite ; alors les variables de sortie sont mises à jour instantanément
— aucune garde sur les transitions partant de la localité n'est satisfaite. Alors, deux situations peuvent se produire : (1) le temps passe, (2) l'environnement modifie la valeur d'une (ou plusieurs) variable(s).

La sémantique d'un VDTA est décrite en termes de *système de transitions temporisé* (*TTS*, modèle fréquemment utilisé pour définir une sémantique dans un modèle temporisé). De façon classique, on y trouve des *transitions discrètes*, et des *transitions de délai* symbolisant l'écoulement du temps. Les transitions discrètes peuvent être de deux types : *transition urgente* ou *transition de mise-à-jour (MAJ) d'entrée*. Les transitions urgentes sont franchies dès que les contraintes sont satisfaites dans la configuration courante du système, ce qui permet de mettre à jour les variables de sortie. Les transitions de MAJ d'entrée permettent seulement de changer les valeurs des variables d'entrée. Elles sont franchies lorsque l'environnement décide de mettre à jour ces variables et lorsque les gardes ne sont pas satisfaites. Les transitions de MAJ d'entrée et de délai ne sont franchies que si aucune transition urgente ne peut être tirée, i.e. aucune garde n'est vraie sur les transitions sortantes de la localité actuelle. Dans notre modèle, les événements sont en réalité les mises à jour sur les entrées, mais comparé à [13], les événements venant de l'environnement ne sont pas explicitement décrits dans le modèle, ce qui le rend plus concis dans le cas de systèmes à flot de données. Par ailleurs, on définit un *Run* comme une séquence alternée d'états et de transitions de la sémantique du VDTA.

Exemple 5. *La figure 2.9 montre un exemple de VDTA $\mathcal{A}$ modélisant la commande bi-manuelle. Il est composé de deux variables booléennes d'entrée L et R (symbolisant l'appui des boutons gauche et droit respectivement), d'une variable booléenne de sortie s (symbolisant la mise en marche ou l'arrêt de la machine), et d'une horloge t. Nous illustrons le modèle à l'aide de deux cas d'utilisation :*

1. *Depuis l'état initial l_0, si L et R sont mis simultanément à 1 par l'environnement, le système va aller instantanément en l_2 après avoir mis en marche la machine ($s := 1$). Cela est du au fait que deux transitions urgentes sont franchies en temps nul.*
2. *A l'état initial, l'environnement écrit la valeur 1 dans l'entrée R, mais L reste à 0. Le système va donc en l_1. Pour atteindre l_0 ou l_2, le système doit quitter l_1 avant une unité de temps. Pour cela, il peut soit remettre R à 0, et le système retourne en l_0, soit mettre L à 1 (en maintenant R à 1), ce qui fera aller le système en l_2 et démarrer la machine. En l_1, si aucune valeur n'est modifiée avant une unité de temps, le système va en l_3, nécessitant ensuite une remise à zéro des deux entrées pour démarrer la machine.*

Ces deux cas illustrent l'intérêt de ce modèle. Notons que ce genre de comportement n'est pas facile à décrire dans un modèle à événements classique, y compris Uppaal[11] *[20]. Dans cet exemple, un état de $\mathcal{A}$ est décrit par un couple (localité, vars) où "localité" est la localité du VDTA, et "vars" est un quadruplet correspondant dans l'ordre à L, R, s et t. On note Id_O lorsque qu'aucune variable n'est affectée. Un exemple possible de* Run *pour $\mathcal{A}$ est :*
$r_1 = (l_0,(0,0,0,0)) \xrightarrow{L:=1} (l_0,(1,0,0,0)) \xrightarrow{Id_O} (l_1,(1,0,0,0)) \xrightarrow{0.3} (l_1,(1,0,0,0.3)) \xrightarrow{R:=1} (l_1,(1,1,0,0.3)) \xrightarrow{s:=1} (l_2,(1,1,1,0.3))$

Par ailleurs, nous considérons un VDTA comme déterministe si pour chaque localité, l'intersection des gardes de deux transitions sortantes est insatisfiable. La définition exacte se trouve dans [C15, C17]. Par la suite, nous ne considérerons que des VDTA déterministes.

2.3.2 Test de conformité à base de VDTA

Comme nous l'avons vu précédemment, le *MBT* consiste à vérifier que les comportements d'une *IUT* (non connue, mais modélisable par un modèle formel connu) sont autorisés par la spécification formelle. Nous avons proposé dans [C17, C15] une adaptation de la relation **ioco** [211], et plus précisément **tioco** [141] dans le contexte des VDTA. Rappelons ici que le principe de **tioco** , c'est d'ajouter comme sortie possible du système l'écoulement du temps. Ainsi, pour qu'une *IUT* soit conforme à sa spécification, il faut non seulement que toutes les sorties possibles de l'*IUT* soient autorisées par la spécification, mais aussi que l'écoulement du temps soit aussi autorisé par cette dernière. Nous avons donc formalisé une relation de conformité entre une implémentation (un VDTA non connu *IUT*) et le VDTA $\mathcal{A}$ décrivant sa spécification. Comme pour **tioco**, tous les comportements de l'implémentation doivent être prévus dans la spécification, ce qui dans notre cas signifie que l'*IUT* ne doit pas faire de mises à jour de variables de sortie à un instant non autorisé par la spécification, et que ces mises à jour doivent avoir lieu uniquement lorsque la spécification le précise. Comme il s'agit d'un test en

11. Notre tentative de modéliser ce système avec *Uppaal* 4.0 a échoué du fait que les invariants sont des contraintes bornées, et que le contraintes de temps ne sont pas autorisées sur les transitions urgentes, ce qui rend difficile la description de la transition urgente sortant de l_1.

boîte noire, la notion de trace est importante. Nous avons donc défini la relation de conformité à partir de traces observables, mais dans le cas d'un VDTA, cette notion n'est pas triviale, car il est possible qu'une variable de sortie change plusieurs fois de valeur en temps nul, ce que nous considérons comme non observable. Ainsi, il est nécessaire d'attendre une *stabilisation* du système pour pouvoir l'observer, ce qui différencie notre relation de **tioco** par exemple (notons qu'il aurait été possible de considérer tous les changement de valeur sur les sorties, ce qui aurait été assez similaire à **tioco**). Dans nos travaux, nous définissons une trace simplement comme un élément de $(A(I) \cup A(O) \cup \mathbb{R}_+)^*$. Nous avons donc défini la notion d'*état stable*. Intuitivement, il s'agit d'un état dans lequel la système va rester un certain temps (même très court), autrement dit, il s'agit simplement d'un état à partir duquel aucune transition urgente ne peut être franchie, i.e. pour quitter ce type d'état, il est nécessaire de mettre à jour une entrée, ou d'attendre. Cette notion est définie sur la sémantique du VDTA. Finalement, on considère dans la relation de conformité des traces stables, i.e. qui se terminent sur un état stable. Nous avons défini la notion de trace observable sur les entrées, comme la projection d'une trace sur les entrées observables de $\mathcal{A}$, i.e. les éléments de $(A(I) \cup \mathbb{R}_+)$.

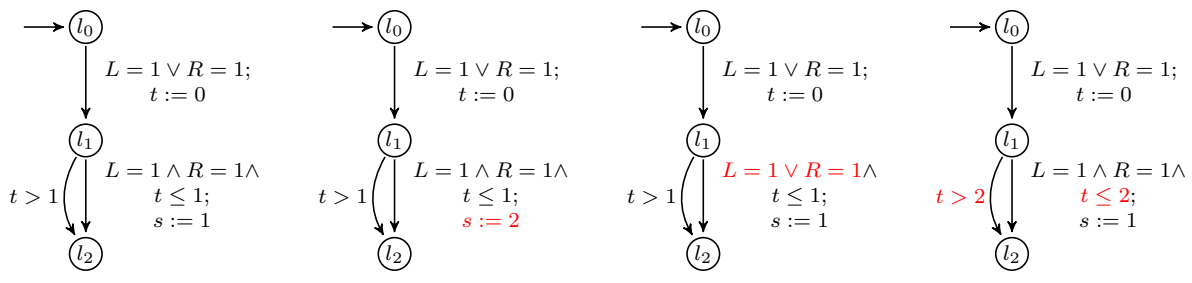
Exemple 6. *Considérons la figure 2.9. Par exemple, les états* $(l_0, (1,0,0,0))$ *et* $(l_0, (1,1,0,0))$ *ne sont pas stables, mais l'état* $(l_2, (1,1,1,0))$ *est stable. Par ailleurs, la trace observable sur les entrées correspondant au Run* r_1 *de l'exemple précédent est* $L := 1; 0.3; R := 1$.

Dans [C17, C15], nous avons proposé la relation de conformité suivante en considérant deux VDTA déterministes $\mathcal{A}$ et IUT correspondant respectivement à la spécification et à l'implémentation :

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

où $ObsTr_I(.)$ correspond à toutes les traces sur les entrées observables depuis l'état initial de $\mathcal{A}$ (comme vu précédemment), $Out(.)$ donne accès aux valeurs des variables de sortie, et *Safter* correspond aux traces après stabilisation. Cette relation de conformité stipule que pour toute séquence d'entrées, les valeurs des variables de sortie de l'implémentation sont égales à celles de la spécification. Ces entrées peuvent être des affectations de variables d'entrées ou le passage du temps. Rappelons qu'on considère ici que le testeur peut lire à tout moment le contenu des variables de sortie.

Exemple 7. *La figure 2.10 illustre des cas d'implémentations non conformes (*L *et* R *sont des entrées booléennes,* s *une sortie entière, et* t *une horloge).*



(a) Spécification $\mathcal{A}$ (b) IUT_1 (c) IUT_2 (d) IUT_3

Figure 2.10 – **tvco** par l'exemple.

— $\neg(IUT_1$ **tvco** $\mathcal{A})$ *car par exemple :* $\mathcal{A}$ *Safter* $(L := 1; 0.3; R := 1)$ *vaut* $s = 1$ *alors que* IUT_1 *Safter* $(L := 1; 0.3; R := 1)$ *vaut* $s = 2$

— $\neg(IUT_2$ **tvco** $\mathcal{A})$ *car par exemple :* $\mathcal{A}$ *Safter* $(L := 1; 0.3)$ *vaut* $s = 0$ *alors que* IUT_2 *Safter* $(L := 1; 0.3)$ *vaut* $s = 1$

— $\neg(IUT_3$ **tvco** $\mathcal{A})$ *car par exemple :* $\mathcal{A}$ *Safter* $(L := 1; 1.5; R := 1)$ *vaut* $s = 0$ *alors que* IUT_3 *Safter* $(L := 1; 1.5; R := 1)$ *vaut* $s = 1$.

Ensuite, nous avons proposé dans [C15] une méthode de génération de test à la volée et non déterministe. Le principe décrit est très similaire à celui de TorX [212] ou Tron [143] pour le cas temporisé. Le testeur connaît le VDTA $\mathcal{A}$ spécifiant le système, et va interagir avec l'*IUT* comme dans le schéma de la figure 2.2. Il va aussi maintenir l'état symbolique courant de la spécification en fonction des entrées qu'il écrit ou des sorties qu'il lit. Rappelons que la correction de l'*IUT* est vérifiée dans des états stables, ce qui signifie qu'en cas de changements multiples d'une variable de sortie en temps nul, seule la dernière valeur sera prise en compte. Nous rappelons brièvement le principe de cet algorithme de génération ci-dessous ; le lecteur intéressé pourra trouver l'algorithme complet (et les propriétés associées) dans [C15] :

Le testeur choisit aléatoirement (si possible) parmi les trois règles suivantes par rapport à la spécification $\mathcal{A}$ *:*

— *Affecter une ou plusieurs variables d'entrée et vérifier les sorties de l'implémentation. En cas de valeur de sortie non spécifiée dans* $\mathcal{A}$*, émettre* **Fail**

— *Le testeur laisse passer le temps tout en observant d'éventuels changements de valeurs de sortie de l'IUT. Si une valeur de sortie non spécifiée dans* $\mathcal{A}$ *est observée, émettre* **Fail**

— *Le testeur décide de stopper la session de test. Un verdict* **Pass** *est alors émis, stipulant qu'aucune erreur n'a été détectée jusqu'à maintenant.*

De façon similaire à Tron, cet algorithme permet d'obtenir des cas de test non-biaisés et exhaustifs à la limite (ces points sont discutés dans [C15]).

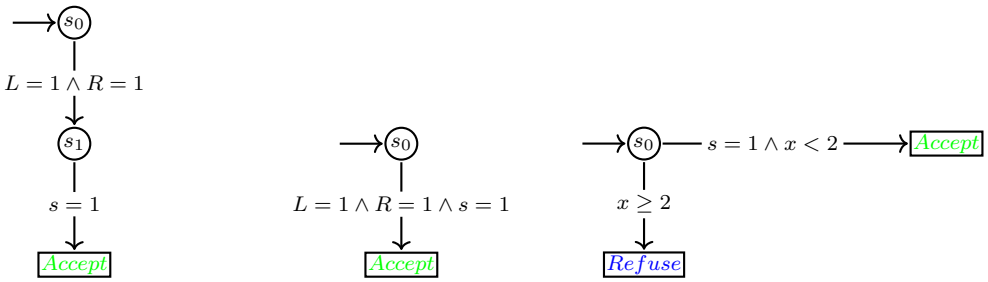
Notons toutefois que la multiplication des aspects aléatoires de cet algorithme (trois règles et le temps d'attente de la deuxième règle) rendent la probabilité d'atteindre un objectif donné très faible. Nous avons donc réfléchi à la possibilité d'étendre cette approche pour atteindre un objectif de test, à la manière de TGV, mais adapté pour des VDTA. Pour cela, nous nous sommes inspirés de l'approche proposée par Bertrand et al. dans [26], qui résout le même problème mais pour des TA à entrées-sorties. Notons que contrairement à [26], nous ne considérons que des modèles déterministes, ce qui simplifie nettement l'approche. Finalement, pour générer des cas de test "à la TGV", nous retrouvons les étapes suivantes, déjà présentes dans le cas non temporisé :

1. *construction d'un produit synchronisé entre la spécification et l'objectif de test,*
2. *sélection de cas de test.*

Nous allons résumer ci-dessous les problèmes liés à ces deux étapes. Le lecteur intéressé trouvera plus de détails dans [C17, C18].

Construction du produit synchronisé Dans un premier temps, il est nécessaire d'exprimer un *objectif de test*, i.e. des comportements à cibler. Pour cela, nous considérons un VDTA particulier TP, équipé d'états Accept, identifiant les états que l'on souhaite cibler, et éventuellement

d'états Refuse identifiant les comportements que l'on ne souhaite pas tester. Un objectif de test est *complet*, i.e. toute variable d'entrée peut être modifiée pendant le processus de test sans blocage, et *non intrusif*, i.e. l'objectif de test ne peut qu'observer les variables et les horloges, mais ne peut pas les affecter. Si besoin, il peut contenir ses propres variables et horloges privées, qu'il peut affecter, afin de décrire des comportements spécifiques. La figure 2.11 montre quelques exemples d'objectifs de test qu'il est possible de décrire.



(a) L et R sont simultanément à 1, et s vaut 1 dans le futur

(b) s vaut 1 en même temps que L et R

(c) s prend la valeur 1 au bout d'au moins 2 unités temps

Figure 2.11 – Exemples d'objectifs de test

Sélection de cas de test Afin de pouvoir sélectionner des cas de test, il est nécessaire de définir un produit synchronisé entre deux VDTA. Nous avons proposé dans [C17] une construction entre deux VDTA $\mathcal{A}$ et TP, basée sur les trois règles suivantes :

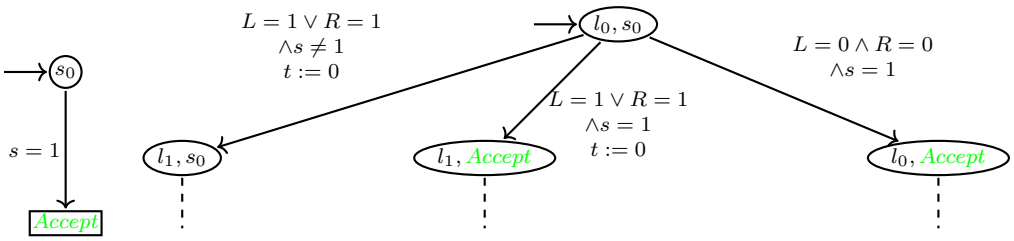
— *une transition urgente est tirable dans $\mathcal{A}$ mais pas dans TP, alors $\mathcal{A}$ change sa localité, mais pas TP,*

— *une transition urgente est tirable dans TP mais pas dans $\mathcal{A}$, alors TP change sa localité, mais pas $\mathcal{A}$,*

— *une transition urgente est tirable à la fois dans $\mathcal{A}$ et TP, et ainsi les deux changent de localité.*

Le lecteur pourra trouver la définition détaillée du produit synchronisé dans [C17]. Cette construction permet d'ajouter des états Accept tout en préservant les traces de $\mathcal{A}$.

Exemple 8. *La figure 2.12b montre un exemple partiel (4 localités) de produit synchronisé entre le VDTA $\mathcal{A}$ de la figure 2.9 et le VDTA TP de la figure 2.12a.*



(a) TP

(b) $\mathcal{A} \times TP$

Figure 2.12 – Exemple partiel de produit synchronisé entre deux VDTA

Comme dans TGV, pour pouvoir sélectionner un cas de test, il faut pouvoir extraire du produit synchronisé les traces acceptées ne conservant que les états "utiles". Pour cela, il est nécessaire d'identifier l'ensemble des états *co-accessibles* depuis un état Accept. Ainsi, pour un état donné P, et en notant Pre l'ensemble des états prédécesseurs de P, il faut être capable de calculer $Pre^*(P)$. Cependant, même si cette opération est simple sur un ioLTS, ça n'est pas le cas pour un VDTA. Sa sémantique étant infinie, il est possible que ce calcul ne termine pas. A la manière d'Alur et Dill dans [6], nous avons donc proposé dans [C17] une abstraction du VDTA, le *TAG (Timed Abstract Graph)* permettant de calculer l'accessibilité et la co-accessibilité. Nous allons détailler cette construction et son analyse d'accessibilité ci-dessous. Finalement, en notant, $\mathcal{A}_{TP}$ le produit synchronisé, et $RG(\mathcal{A}_{TP})$ le TAG correspondant, il suffit d'appliquer l'algorithme de co-accessibilité sur les états de $RG(\mathcal{A}_{TP})$ marqués Accept, et de conserver les chemins qui contiennent l'état initial. La démarche complète ainsi qu'un algorithme d'exécution des tests permettant de guider l'*IUT* vers les états acceptants est disponible dans [C17].

2.3.3 Accessibilité dans un VDTA

Pour pouvoir assurer la terminaison des algorithmes d'accessibilité, nous avons proposé dans [C17, C18] une transformation du VDTA vers un modèle abstrait, le *TAG*, inspiré du *graphe de régions* d'Alur et Dill [6], qui partitionne l'espace des valeurs des horloges en des classes d'équivalence appelées *régions*. L'intérêt de cette approche, c'est qu'elle préserve l'accessibilité des états. Nous donnons ici les grands principes de cette construction. Le lecteur intéressé trouvera toutes les définitions détaillées dans [C17]. La notion de *région* est similaire à celle d'Alur et al. dans [6]. Comme dans le graphe de régions, les informations liées au temps sont aussi contenues dans les états du TAG. Les localités symboliques du TAG sont composées d'un couple correspondant à la localité du VDTA initial, et une région d'horloge. Il y a deux types de transitions : des transitions permettant d'affecter les variables sortie et les horloges ($U1$), et des transitions décrivant l'écoulement du temps ($U2$). Toute les transitions sont urgentes. Le système évolue d'un état vers son successeur temporel lorsque les valeurs des horloges atteignent cette région (suite à écoulement du temps). Comme toutes les transitions sont urgentes, afin de donner priorité aux transitions de type $U1$, il est nécessaire d'ajouter sur les transitions de type $U2$ une garde supplémentaire correspondant au complémentaire des variables d'entrées sur les autres transitions sortantes (de type $U1$).

Exemple 9. *La figure 2.13 montre le TAG correspondant au VDTA décrivant la commande bi-manuelle de la figure 2.9. Dans ce graphe, les transitions horizontales (et en bleu) sont les transitions d'écoulement du temps ($U2$). Si on reprend la séquence d'entrées $L := 1; 0.3; R := 1$ vue précédemment, voici le chemin correspondant dans le TAG : depuis l'état initial $(l_0, t = 0)$, l'affectation $L := 1$ fait passer immédiatement en $(l_1, t = 0)$. Puis l'attente de 0.3 unités de temps fait évoluer le TAG en $(l_1, 0 < t < 1)$ (transition horizontale d'écoulement du temps). Finalement, l'affectation $R := 1$ mène en $(l_2, 0 < t < 1)$, permettant de constater le démarrage de la machine ($s := 1$).*

Nous avons proposé dans [C17] un algorithme de co-accessibilité basé sur un TAG. L'idée est de sélectionner un ensemble d'états cibles dans le TAG, et de calculer successivement l'ensemble des états prédécesseurs, sur un principe de point fixe. Ainsi, le prédécesseur abstrait $aPred$ d'un état P du TAG, est défini par $aPred(P) = aPred_o(P) \cup aPred_i(P) \cup aPred_t(P)$,

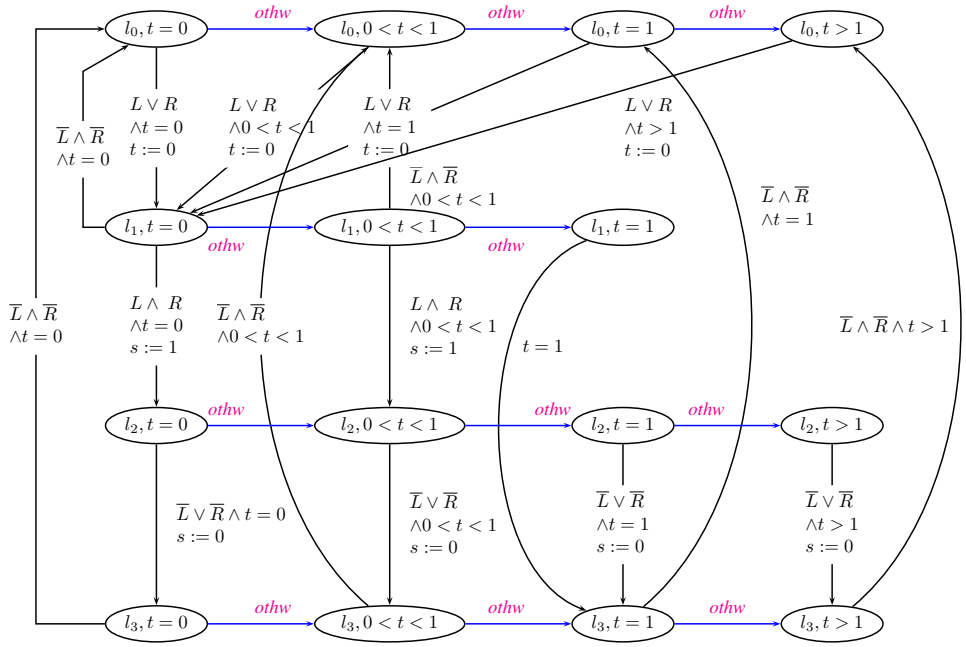


Figure 2.13 – Le TAG correspondant au VDTA de la figure 2.9

.

correspondant respectivement aux prédécesseurs abstraits en cas d'affectation de sortie, d'affectation d'entrée, ou d'écoulement du temps. Finalement, la fonction de co-accessibilité $coreach_a$ est définie comme le plus petit point fixe de la fonction $\lambda X.X \cup aPred(X)$. Le lecteur intéressé trouvera toutes les définitions ainsi que les preuves de correction dans [C17]. Par ailleurs, l'utilisation de régions dans le TAG risque de provoquer un nombre exponentiel d'états. Ainsi, nous avons proposé dans [C20] une approche pour l'accessibilité à base de zones [22, 83, 219, 124], qui est une représentation reconnue comme plus efficace et compacte des contraintes de temps. Cette approche étant relativement technique, nous ne la développerons pas ici. La description détaillée de cette approche pourra être trouvée dans [C20].

2.4 Test à distance de systèmes temporisés

Dans les méthodes de test décrites précédemment, nous avons utilisé l'architecture de test décrite figure 2.2, et nous avons fait l'hypothèse que le temps de propagation entre le testeur et l'IUT était négligeable. Il s'agit de *test synchrone*. Cependant, cette hypothèse est parfois abusive, comme discuté dans [133, 204], notamment dans des situations où le testeur est assez éloigné de l'IUT (par exemple, dans le test de services web et/ou des systèmes ayant des architectures complexes) ou encore dans des systèmes réactifs temps-réels ayant un temps de réponse (souhaité) très court. Ainsi, l'ordre des actions observées par le testeur peut différer de l'ordre réel observé par l'*IUT*, ou simplement les dates observées par l'*IUT* peuvent ne pas correspondre aux dates observées par le testeur, et provoquer un verdict inapproprié, notamment dans un contexte temporisé (cf Exemple 10). Ces problèmes sont connus comme étant des problèmes d'*observabilité* du test. Nous avons donc décidé de lever cette hypothèse et cherché à intégrer

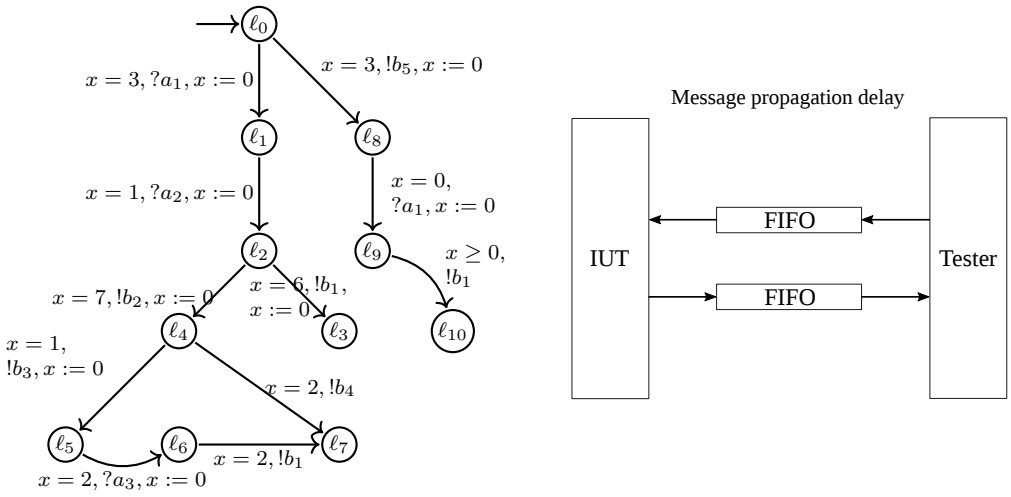
le temps de propagation dans la théorie de test temporisé. Il s'agit donc de test *asynchrone* mais dans un cadre temporisé, ce qui différencie cette approche de la plupart des travaux précédents sur la même thématique.

Avant nos contributions sur le sujet, le test à distance asynchrone a fait l'objet d'un certain nombre de travaux [133, 204, 176, 127, 137], mais essentiellement dans un cadre non temporisé (ou du moins, le modèle de spécification utilisé n'est pas temporisé). Ils pointent généralement le fait qu'il est nécessaire soit d'ajouter des informations dans le procédé de test, soit de faire des hypothèses sur le modèle, pour se ramener à des techniques de test synchrone. Par exemple, [133, 127] montrent qu'il est possible d'utiliser les techniques de test synchrone simplement en ajoutant des informations d'horodatage dans les messages, soit par un principe de *piggyback* [133], ou en associant à une action dans les séquences de test son temps d'observation par une horloge locale associée à chaque port [127], alors que Simao et Petrenko définissent dans [204] une condition suffisante pour que les tests asynchrones et synchrones fournissent le même verdict : il suffit que les entrées ne soient autorisées par la spécification que sur des état quiescents. Jard et al. proposent dans [133] un framework de test à distance asynchrone en équipant la spécification (sous forme d'ioLTS) de FIFOs, définissant ainsi une sémantique asynchrone globale permettant d'utiliser une relation de conformité directement sur cette sémantique. Ensuite, le fait d'ajouter les informations d'horodatage permet d'utiliser les techniques de test synchrone. Dans [204], Simao et al. utilisent aussi une architecture à base de FIFOs, et un modèle proche de l'ioLTS (un ioTS). Ils proposent une approche à base d'objectif de test (OT) : l'OT est transformé pour y intégrer la distorsion induite par les FIFOs (en réalité représentées par des ioLTS), et ce avant d'effectuer la composition avec la spécification, ce qui permet d'éviter l'explosion d'états (généralement provoquée par les FIFOs). On peut aussi citer les travaux de Noroozi et al. [176] qui revisitent les approches précédentes de test asynchrone, et montrent dans quelle mesure il est possible (ou pas) de synchroniser le test, en utilisant des canaux en FIFO et des mécanismes de test synchrone pour obtenir un verdict cohérent. On y trouve aussi la définition d'une sémantique asynchrone intégrant des FIFOs, et la proposition d'une condition suffisante sur un ioTS pour que le test à distance ne dépende pas du contexte. Parmi les approches citées, seule celle de Khoumsi [137] n'utilise pas de FIFO dans le modèle. En considérant un tuple de FSM, il montre que les problème de contrôlabilité et d'observabilité peuvent être résolus en ajoutant des messages de coordination en lien avec les temps de réaction et d'attente des testeurs locaux. Enfin, d'autres travaux [120, 77] ciblant spécifiquement les systèmes concurrents, proposent de générer des tests à partir de modèles *Event Structure*, qui ont l'avantage de spécifier l'ordre des événements de façon explicite. Ce domaine a continué à intéresser la communauté, plus généralement dans le cadre du test de systèmes distribués (temporisés), et d'autres résultats postérieurs (ou simultanés [12]) aux nôtres ont été proposés par Gaston et al. [112] qui proposent un framework de test distribué temporisé pour un tuple de TA à entrées/sorties avec variables (TIOSTS) et une architecture adaptée. Ils définissent une relation de conformité **dtioco** qui est vraie si et seulement si la projection locale vue par le testeur vérifie la relation de conformité **tioco** (que nous allons décrire par la suite), et que les messages échangés par les différentes entités vérifient des règles de communication précises. Récemment, une extension de ces travaux a été proposée par Benharrat et al. [23], qui améliorent la vérification des règles de communication en les transformant en système de contraintes qu'il est ensuite possible de résoudre à l'aide d'un solveur.

12. Les travaux de Gaston et al. évoqués par la suite ont été présentés à la conférence ICTSS13, conférence dans laquelle nous avons aussi présenté nos résultats.

Comme nous souhaitions utiliser un cadre et des modèles temporisés, nous sommes partis des travaux de Krichen et Tripakis décrits dans [141] qui modélisent la spécification du système à tester à l'aide d'un TA à entrées-sorties (TIOA ou TAIO) et qui définissent une relation de conformité **tioco** adaptée de la relation **ioco** comme vu auparavant. Notons que nous aurions pu sans problème nous baser sur les travaux de [166, 125] dans lesquels la relation **rtioco** est assez similaire. En s'inspirant des travaux de Jard et al. [133], nous nous concentrons sur deux entités distantes, le testeur et l'IUT, et nous utilisons le pouvoir expressif des modèles temporisés pour décrire un lien explicite intégrant le temps de latence entre ces différents éléments. Ainsi, dans [C22], nous avons proposé une sémantique permettant d'intégrer explicitement les délais de latence dans le framework de test, et une relation de conformité basée sur cette sémantique. Ensuite, nous avons proposé une condition suffisante simple pour garantir l'absence d'entrelacements dus aux problèmes d'observabilité dans le processus de test (idée que l'on retrouve dans [176]), et enfin, nous avons proposé une étude de cas utilisant *Uppaal-Tiga* [48, 76] pour générer des cas de test à distance. Nous allons évoquer brièvement ces contributions ci-dessous. Le lecteur intéressé trouvera tous les détails dans [C22].

Nous considérons donc une spécification $\mathcal{A}$ sous forme de TIOA défini de façon similaire au TAIO de Krichen et Tripakis dans [141]. Ce chapitre ayant été pensé dans l'optique de rester à un niveau intuitif, il a été décidé de ne pas fournir la définition précise de TIOA. Noter qu'elle se rapproche de la définition 3 de TA fournie au chapitre 4, à ceci près que le TIOA est défini sur un alphabet comportant des entrées (marquées par un $?$) et des sorties (marquées par un !), et qu'il n'y a pas d'états acceptants. La figure 2.14a montre un exemple de TIOA. Par exemple, à la localité initiale l_0 de $\mathcal{A}$, le système peut soit recevoir l'action $?a_1$, soit émettre l'action $!b_5$, dans le deux cas au bout de 3 unités de temps. Dans les deux situations, l'horloge x est remise à zéro. Par la suite, nous allons nous intéresser particulièrement à la notion de trace temporisée, définie comme une alternance de dates et d'événements liés à ces dates. Par exemple, une trace temporisée possible pour $\mathcal{A}$ est : $\sigma_1 = (3{\cdot}?a_1) \cdot (4{\cdot}?a_2) \cdot (11{\cdot}!b_2) \cdot (12{\cdot}!b_3) \cdot (14{\cdot}?a_3)$, qui signifie que a_1 est reçu au temps (global) 3, puis que a_2 est reçu au temps 4, b_2 est émis au temps 11, etc... L'architecture du test est illustrée figure 2.14b. A la manière de [133, 204], nous avons



(a) Un exemple de TIOA $\mathcal{A}$ (b) Architecture de test à distance

Figure 2.14 – Test à distance dans un cadre temporisé

intégré deux FIFOs [13] pour prendre en compte les délais de propagation. Plus précisément, le modèle consiste en une architecture $2FIFO(\bowtie, \Delta)$ avec : (1) une FIFO pour chaque direction de communication entre l'*IUT* et le testeur, (2) une latence de communication bornée par Δ. Le symbole $\bowtie$ signifie $\leq$ ou $=$.

Dans cette architecture, il existe des situations où le temps de propagation d'un événement d'un cas de test ne peut être négligé. C'est le cas par exemple lorsque l'instant de réception d'une entrée par l'IUT a des conséquences sur le reste de l'exécution.

Exemple 10. *Considérons la spécification $\mathcal{A}$ de la figure 2.14a et un délai de propagation de 2 unités de temps. Si le testeur souhaite que l'IUT reçoive a_1 à la date $t = 3$, il devrait l'envoyer à la date $t = 1$, de même, lorsque l'IUT envoie b_2 à la date $t = 11$, le testeur la reçoit à la date $t = 13$, et ainsi de suite. Ainsi, le testeur devrait observer la trace $\sigma'_1 = (1{\cdot}?a_1) \cdot (2{\cdot}?a_2) \cdot (12{\cdot}?a_3) \cdot (13{\cdot}!b_2) \cdot (14{\cdot}!b_3)$ alors que l'IUT exécute en réalité la trace $\sigma_1 = (3{\cdot}?a_1) \cdot (4{\cdot}?a_2) \cdot (11{\cdot}!b_2) \cdot (12{\cdot}!b_3) \cdot (14{\cdot}?a_3)$. Précisons que σ_1 est bien une trace temporisée de $\mathcal{A}$, mais pas σ'_1. Par conséquent, l'observation de la trace σ'_1 par le testeur pourrait être considérée comme non conforme, alors qu'elle correspond bien à une trace temporisée de $\mathcal{A}$. De plus, l'ordre (apparent) des événements diffère selon le point de vue du testeur ou de l'IUT. Il s'agit de problèmes d'observabilité du test.*

Ainsi, le test à distance amène de nouveaux challenges : prendre en compte le délai de propagation entre le testeur et l'IUT, et gérer l'entrelacement induit par les problèmes d'observabilité.

Dans un premier temps, nous avons proposé dans [C22] une sémantique asynchrone sous forme d'*Input Output Transition Timed System (IOTTS)* (un modèle équivalent au TIOTS utilisé dans [125]) permettant d'intégrer les délais de propagation bornés par Δ dans le modèle, et ainsi de décrire globalement les évolutions du système. Le principe, repris de [133] et schématisé figure 2.15, consiste à considérer le TIOA et les FIFOs comme un modèle unique global. Ainsi,

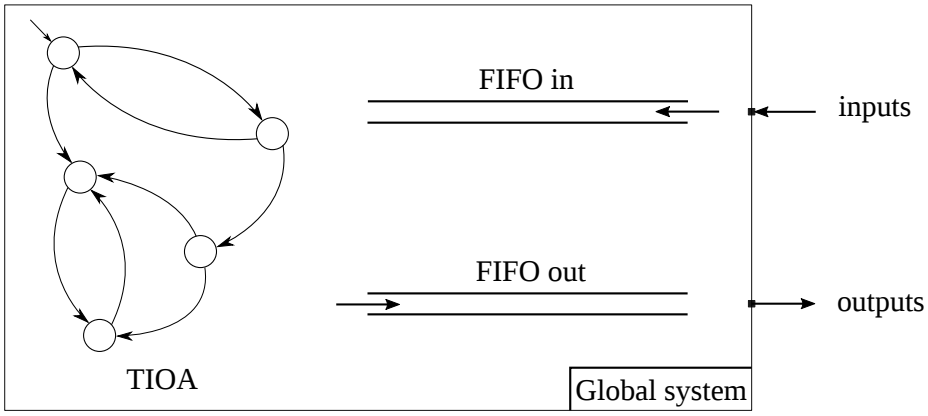


Figure 2.15 – Principe de la sémantique asynchrone

les observations réelles par le testeur correspondent aux entrées-sorties de cette sémantique. On retrouve dans cette sémantique les évolutions possibles du système qui sont :

13. Modèle fréquemment utilisé. On le trouve notamment dans la sémantique d'Estelle, SDL et TTCN.

— occurrence d'un événement d'entrée $?a$ au niveau du système global. L'événement est inséré dans la FIFO d'entrée, et un timer δ est armé.

— lorsque le timer de l'événement $?a$ en tête de la FIFO d'entrée arrive à expiration (au temps $\delta \leq \Delta$), la transition correspondant à la réception de $?a$ par le TIOA est franchie, l'état courant du TIOA est mis à jour. Rien n'est observé à l'extérieur du système global (τ transition)

— écoulement du temps

— occurrence d'un événement de sortie $!b$ au niveau du TIOA local. L'événement est inséré dans la FIFO de sortie, et un timer δ est armé. Rien n'est observé à l'extérieur du système global (τ transition).

— envoi de l'événement $!b$ en tête de la FIFO de sortie (au temps $\delta \leq \Delta$).

Grâce à cette sémantique, il est possible de définir la relation de conformité directement sur le système global, ce que nous avons décrit dans [C22].

Sur le même principe que [176] qui propose une condition suffisante sur un ioLTS pour que le test à distance ne dépende pas du contexte, nous avons cherché dans quelle mesure il était possible de se prémunir des problèmes d'observabilité dus au modèle asynchrone dans une session de test. Pour simplifier le problème, nous considérons ici que le délai de propagation exact Δ entre le testeur et l'IUT est connu. Dans ce contexte, nous avons proposé dans [C22] un critère mesurable sur un TIOA permettant de garantir qu'il n'y aura pas ce genre d'entrelacement : la $\Delta-$*testabilité*. Ainsi, pour qu'un TIOA soit $\Delta-$*testable*, il faut que toutes ses traces le soient, i.e. pour chaque trace temporisée du TIOA, l'occurrence d'une entrée (venant du testeur) ne peut être précédée par une sortie à moins qu'un temps minimum de 2Δ ne se soit écoulé entre les deux. Intuitivement, ce principe inspiré du *Round Time Trip (RTT)* de certains protocoles réseau, permet de garantir que le testeur puisse réagir à un stimulus précédent venant de l'IUT [14]. Cette idée est aussi inspirée de travaux de Khoumsi [137] qui définit des contraintes de temps suffisantes pour garantir la contrôlabilité et l'observabilité, mais dans pour un ensemble de FSM (modèle non temporisé, et avec une sortie pour une entrée). Le lecteur intéressé trouvera la définition précise de $\Delta-$*testabilité* dans [C22], ainsi que les propriétés associées. Notons qu'il s'agit ici d'une condition suffisante, mais pas nécessaire, et qu'il est possible que des critères plus fins existent. La $\Delta-$testabilité d'un TIOA $\mathcal{A}$ est décidable. Il faut placer $\mathcal{A}$ en parallèle avec un TIOA observateur O_Δ, qui mène dans un état d'erreur si et seulement si le temps séparant toute action de sortie et toute action d'entrée de $\mathcal{A}$ dépasse 2Δ. La $\Delta-$testabilité devient alors équivalente à un problème d'accessibilité dans un TA, i.e. vérifier si aucune trace de $\mathcal{A}|O_\Delta$ ne mène à la localité d'erreur de O_Δ.

Enfin, nous avons étudié dans [C22] la possibilité d'utiliser le logiciel *Uppaal-Tiga* [48, 76] pour réaliser un test à distance. Avec la collaboration de l'équipe de Kim Larsen, nous avons donc repris les approches de test et l'étude de cas *Light Controller* proposées dans [75, 76], en y ajoutant notamment des FIFOs dans le modèle, comme décrit figure 2.16. L'originalité réside dans la manière d'encoder les FIFOs : chaque *cellule* de FIFO est modélisée par un automate ayant son propre identifiant. Chaque automate est équipé de sa propre horloge permettant de retarder (et dans notre cas de le renommer) les événements entrants et sortants pour simuler un délai de latence. Le lecteur pourra trouver tous les détails de cette étude dans [C22].

14. Nous avons considéré ici le temps de calcul du testeur comme nul, dans le cas contraire, il suffirait de l'ajouter au temps de propagation.

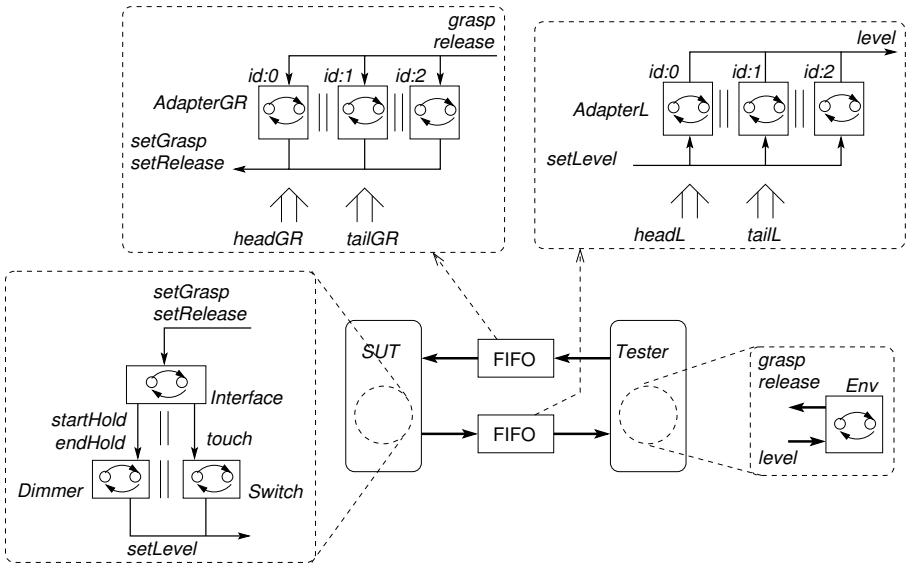


Figure 2.16 – Schéma de test pour l'exemple du "Light Controller"

2.5 Conclusion et perspectives

Les contributions présentées dans ce chapitre sont maintenant assez anciennes, mais les domaines concernés restent actifs. Les travaux sur le test de robustesse ont plutôt évolué vers des travaux sur l'injection de fautes (qui a même fait récemment l'objet d'une journée spécifique dans le cadre du GDR GPL) [15], et dans des domaines d'application ciblés, notamment les systèmes autonomes ou la (cyber-)sécurité. C'est le cas des travaux du LAAS [188], mais aussi des travaux récents de l'équipe de Koopman [129] qui restent très actifs sur le sujet. Simultanément à Fernandez et al. [104] nous avons présenté les premières approches de test de robustesse à base de modèle, ce qui fait que ces travaux sont généralement mentionnés dans les papiers récents sur le sujet (comme [129]) ainsi que dans les articles états de l'art (e.g. [203]). Dans le même esprit, le modèle VDTA que nous avons décrit dans ce chapitre, apparaît dans des articles états de l'art (récents) sur le test de systèmes temps-réels. C'est le cas par exemple de [86, 215]. Il faut admettre que le fait de proposer un outil abouti implémentant cette approche aurait probablement favorisé son adoption plus importante auprès de la communauté scientifique. Enfin, nos contributions (plus récentes) concernant le test à distance ont été explicitement reprises dans le récent framework de test distribué *DTRON* [9], comme stipulé dans la thèse d'Aivo Anier [8] qui écrit : *"Therefore the thesis focuses on the notion of weaker conformance relation - delta-testability* (citation de [C22]) *that is one of the main design consideration for execution environment DTRON is designed for."*

Les perspectives concernant le *MBT* sont nombreuses, ne serait-ce que parce qu'il s'agit d'un domaine très vaste. Généralement, dès qu'un nouveau formalisme de spécification apparaît, ce qui est très fréquent, il est nécessaire de mettre au point une nouvelle méthode de génération de cas de test adaptée à ce dernier. Dans le meilleur des cas, il peut s'agir d'une tentative de réutiliser une méthode de test existante, mais il peut parfois être nécessaire de définir une nouvelle méthode. Concernant plus précisément les contributions qui viennent d'être présentées, certains aspects mériteraient d'être approfondis. Par exemple, le test de robustesse reste

15. Détails à l'adresse `https://wp-systeme.lip6.fr/jaif/`.

un enjeu de recherche majeur, notamment avec l'augmentation du nombre de (petits) systèmes autonomes. Il est donc nécessaire de mettre en œuvre des techniques efficaces d'injection de fautes pour s'assurer que le *SUT* résiste à des conditions stressantes. En ce qui concerne les systèmes autonomes, il nous semble pertinent d'orienter les travaux sur l'injection de fautes dans un environnement simulé, sur le principe des travaux de Sotiropoulos et al. [206, 207]. L'idée serait d'identifier des événements inattendus mais réalistes provenant de l'environnement, et de les intégrer à la simulation de façon accélérée pour permettre un nombre plus important de tests. Concernant le test de systèmes à flots de données, la priorité serait de développer un outil de test permettant à des industriels de modéliser facilement leur produit. En s'inspirant de l'approche *Altarica* [10], ou plus récemment de *Stimulus* [134], l'idée serait de fournir un environnement graphique permettant de décrire simplement des modèles, et de les simuler, et bien entendu de les tester. Précisons que ce besoin d'outils ne se limite pas au test de systèmes à base de VDTA, mais s'applique globalement au *MBT*. D'un point de vue plus théorique, nous avons considéré dans notre approche uniquement des modèles déterministes. La levée de cette hypothèse compléterait ces travaux. Concernant le test asynchrone, il serait intéressant d'étudier globalement l'existence de stratégies de jeu gagnantes dans le cas asynchrone et de fournir leur sémantique. Notamment, en supposant un temps de propagation Δ constant et connu, étudier dans quelle mesure la stratégie de jeu dans le cas asynchrone est décidable. Plus généralement, on peut se demander sous quelles conditions l'existence de stratégies asynchrones est décidable, voire étudier l'équivalence générale de l'existence entre les stratégies synchrones et asynchrones. La résolution de ces problèmes (théoriques) ne se limite pas au test à distance entre deux entités, et il est naturel d'envisager d'étendre ces questions dans le cadre plus large du test distribué, i.e. dans dans le cas ou de multiples entités distribuées communiqueraient avec des temps de propagation explicites. Il faut noter que le test de systèmes distribués, et notamment temporisés, demeure un enjeu de recherche majeur du *MBT*. Cela s'explique assez simplement par le fait qu'il cumule plusieurs difficultés, comme la gestion du test en lui-même, la prise en compte de plusieurs entités distantes, et les soucis de contrôlabilité et d'observabilité. Ce dernier point est source de problèmes théoriques, abordés notamment par David et al. dans [76] pour les aspects observabilité partielle, et plus récemment par Henry et al. dans [121] pour la contrôlabilité partielle. Dans [76], David et al. proposent de considérer uniquement un sous-ensemble des sorties venant de l'*IUT* comme pouvant être observées, et proposent une nouvelle relation de conformité et une stratégie de jeu pour générer des cas de test. Le problème est en fait transformé en problème (décidable) de contrôle sur les automates temporisés à jeux, modèle utilisé dans *Uppaal-Tiga* qui fournit une stratégie gagnante. Le principe décrit dans [121] est assez ressemblant, mais les auteurs s'intéressent à la contrôlabilité. Ils transforment aussi le problème en jeu à deux joueurs (testeur et *IUT*), et proposent une stratégie cherchant à maximiser le contrôle, et définissent notamment une zone dans laquelle le testeur peut mettre en œuvre une stratégie gagnante. Ces aspects sont aussi liés à d'autres plus pratiques. Les approches de test distribué nécessitent généralement un testeur par composant, et/ou des points d'observation, ce qui n'est pas toujours simple à mettre en œuvre. Il serait donc utile de réfléchir à la possibilité d'intégrer de façon incrémentale ces différents éléments, en fonction des résultats courants du test, un peu à la manière du test d'intégration ou chaque composant est ajouté de façon progressive. Ainsi, selon l'état courant du test et notamment une sorte d'analyse d'observabilité et de contrôlabilité de l'ensemble, il serait possible d'identifier si le test peut s'effectuer dans des conditions satisfaisantes, ou si d'autres entités sont nécessaires dans l'architecture, et si oui à quel endroit les intégrer. Finalement un autre enjeu du *MBT* réside dans son domaine d'application ou encore dans la caractéristique testée. Des domaines émergeant et ayant leurs propres spécificités

sont très consommateurs de nouvelles techniques de test. C'est notamment le cas des systèmes cyber-physiques, et plus spécifiquement dans le domaine médical, mais aussi les systèmes autonomes. Le test de ce genre de systèmes en monde virtuel ouvre notamment des perspectives intéressantes pour réduire le coût des tests. Concernant la caractéristique du test, le test de sécurité devient un enjeu majeur de société. Ces différents domaines seront développés au chapitre 5 consacré à mes perspectives de recherche.

Chapitre 3

Vérification de programme : l'étude de cas du *FlashManager*

Lorsque la spécification d'un système a été vérifiée, il est possible de procéder à son implémentation. Cette phase se déroule généralement en plusieurs étapes, puisqu'il est nécessaire d'écrire le code du programme, puis de le compiler sur une architecture processeur donnée pour obtenir l'implémentation. Les méthodes de test décrites au chapitre 2 considèrent ensuite cette implémentation comme une boîte noire, et utilisent la spécification pour générer des données de test. Comme nous l'avons vu précédemment, il s'agit d'un test dynamique car l'implémentation est exécutée. On parle aussi de test *Hardware in the Loop (HIL)* car le système est compilé sur son architecture finale. Cependant, comme discuté au chapitre 1, il est avéré que plus une faute est détectée tard dans le cycle de développement, plus elle coûte cher à réparer. Ainsi, il est utile de mettre en place en amont du test dynamique des méthodes de test statique permettant de s'assurer que le code du programme satisfait certaines propriétés. Cette vérification se fait directement en analysant le code source du programme ; on parle donc de *vérification de code*. De plus, cette approche est par définition structurelle, dans le sens où elle analyse la structure du programme (ici le code). Elle permet donc potentiellement de détecter d'autres fautes que les méthodes fonctionnelles (comme le MBT vu au chapitre 2). Ces différentes techniques sont donc complémentaires.

Avant d'aborder nos contributions concernant la vérification de code, nous allons rappeler quelques éléments de base. Le *model-checking* [91, 64] est la technique de vérification la plus connue permettant de vérifier des propriétés, initialement exprimées sous forme de logique temporelle, sur un modèle à états finis. Cette technique est efficace mais très combinatoire. Ainsi, la technique du *model-checking symbolique* [41, 69] où les ensembles d'états sont représentés implicitement à l'aide de fonctions booléennes, puis la représentation à l'aide de *Binary Decision Diagrams* (BDD, [69, 159]) ont permis d'améliorer nettement les performances du model-checking. Cependant, dans certaines situations, la taille de mémoire nécessaire pour stocker les BDD est trop importante pour pouvoir être mise en œuvre. Pour cette raison, Biere et al. proposent une nouvelle technique, le *Bounded Model Checking (BMC)* [27]. Le principe du BMC consiste simplement à chercher un contre-exemple à la propriété en bornant la longueur des exécutions possibles à un entier k. Si un bug est trouvé, alors le système est défaillant. Dans le cas contraire, la valeur de k est incrémentée jusqu'à ce que, soit une erreur soit trouvée, soit il ne soit plus possible de calculer le résultat pour cette longueur d'exécution (en général, il s'agit d'un *timeout*). Bien entendu, si aucune erreur n'est trouvée, la correction n'est pas ac-

quise, c'est d'ailleurs la principale faiblesse de cette technique. Cependant, dans la pratique, l'assurance que le programme est correct pour une valeur de k donnée peut suffire pour garantir une certaine confiance. Les techniques de BMC ont été utilisées à l'origine dans le domaine du design de hardware [28] et plus récemment dans le domaine la vérification de logiciels (le lecteur trouvera une étude à ce sujet dans l'article de D'Silva et al. [85]). Par ailleurs, cette technique est proposée dans bon nombre d'outils, qu'il s'agisse de model-checking de modèle, comme dans NuSMV [1] [61], ou d'outils d'analyse de programmes *C* [63, 67, 43, 131, 102], ou plus récemment Java [68]. Pour analyser un programme, le BMC [111] consiste à transformer le programme déplié k fois et la propriété à vérifier en une (potentiellement grosse) formule propositionnelle ϕ de telle sorte que ϕ est satisfiable *si et seulement si* il existe un contre-exemple de profondeur inférieure à k. Ensuite, la satisfiabilité de ϕ est calculée à l'aide d'un solveur de contraintes, qui peut être par exemple un solveur SAT comme par exemple Glucose [11] ou SMT comme par exemple Yices [87], CVC4 [194], ou z3 [78], dont l'efficacité permet de résoudre des problèmes ayant un très grand nombre de paramètres. Précisons que le BMC n'est pas la seule technique de vérification utilisée pour analyser du code. On peut citer par exemple les techniques d'interprétation abstraite [70] ou de vérification déductive [128, 106, 81] qui sont de plus en plus utilisées dans les outils récents, comme Frama-C [2] [72, 139] ou Why3 [105].

Ce chapitre présente nos contributions autour du BMC de code *C* dans un contexte temps-réel. Ce travail a été publié dans [J5]. Tout d'abord, nous décrivons une stratégie de recherche dynamique en *backjump* basée sur le code source permettant d'élaguer au plus tôt les branches inutiles lors de la vérification de propriété. Cette optimisation a été ajoutée à l'outil de BMC nommé *DPVS*, développé par l'I3S. Ensuite, à travers un exemple fourni par la société Geensys (maintenant Dassault Systèmes), nous montrons que l'approche BMC standard permet difficilement de vérifier certaines propriétés temps-réel, et nous décrivons dans quelle mesure l'ajout de la *programmation par contraintes* permet d'améliorer ces résultats. Pour cela, nous avons comparé nos résultats à l'outil *CBMC* [63], un des outils de BMC les plus connus pour analyser du code *C*.

Ce travail a été effectué entre autres dans le cadre du projet ANR TESTEC. Il est le fruit d'une collaboration avec Hélène Collavizza, Le Vinh Nguyen, Olivier Ponsini et Michel Rueher et a généré la publication [J5].

Nous avons choisi de consacrer un chapitre complet sur ce travail essentiellement pour des raisons thématiques. Le travail décrit ici a eu lieu sur une période assez brève par comparaison aux contributions présentées dans les autres chapitres. C'est la raison pour laquelle il est nettement plus court. Son organisation est la suivante :

— La section 3.1 présente la stratégie de recherche dynamique en *backjump* qui a été implémentée à l'outil de BMC *DPVS*.

— La section 3.2 présente notre étude de cas sur une application industrielle réelle, dans un cadre temps-réel : le *FlashManager*. Nous y décrivons notamment notre méthodologie pour décrire les propriétés attendues, et résumons les résultats expérimentaux obtenus, comparés à l'outil de BMC de référence *CBMC*.

1. NuSMV propose trois principales fonctionnalités : model-checking de formules CTL et LTL basé sur les BDD, BMC, et simulation de modèles.

2. Frama-C est une plate-forme open-source d'analyse de code C permettant de l'analyse statique, de la vérification déductive, mais aussi la génération de tests.

3.1 Stratégie de recherche dynamique en *backjump*

La *programmation par contraintes* (ou *Constraint Programming (CP)*) correspond à la "manière de modéliser et résoudre des problèmes d'optimisation combinatoire". Pour y parvenir, on utilise généralement un solveur de contraintes qui peut être industriel (e.g. ILOG/IBM, Comet, Microsoft z3), ou académique (e.g. Gecode, Choco, Minion, Glucose). Généralement, les pistes pour résoudre ce type de problème consistent à (1) considérer chaque contrainte séparément, et supprimer les valeurs rendant le système trivialement inconsistant, (2) mettre en place des stratégies de recherche qui essaient d'exploiter la structure du problème, ou (3) essayer d'extraire des sous-classes des contraintes permettant d'utiliser des algorithmes spécifiques efficaces. L'efficacité de la résolution à l'aide d'un solveur dépend de la stratégie de recherche utilisée. La programmation par contraintes a surtout montré son utilité pour la génération de tests à partir de code source. De nombreux outils utilisant cette technique ont été développés, que ce soit pour du code *C / C++* (e.g. Cute, Crest, Dart, EXE, Inka, Pathcrawler, Taupo), du code *Java / C#* (e.g. Catg, JCute, Java Path Finder, Pex, Pet), ou encore directement sur du code binaire (e.g. Osmose, Sage, Triton).

L'outil *DPVS* sur lequel nous avons travaillé utilise le principe du BMC : la propriété que l'on cherche à satisfaire est vérifiée pour une borne k donnée, ce qui signifie que le programme est déplié k fois avant d'entamer la résolution. Dans une approche BMC classique, comme dans *CBMC*, le programme déplié et la propriété sont traduits en une formule propositionnelle ϕ qui est satisfiable si et seulement si il existe un contre-exemple de profondeur inférieure à k. Puis un solveur de contraintes permet de calculer cette satisfiabilité. Dans *DPVS*, le programme est aussi déplié, puis un *Graphe de Flot de Contrôle (GFC)* (une représentation équivalente du code source sous forme de graphe orienté représentant toutes les exécutions possibles du programme [3]) simplifié est construit, et finalement ce GFC ainsi que la propriété sont traduits en contraintes à la volée. Contrairement aux méthodes BMC classiques, cette approche permet de décider dans quel ordre le GFC va être exploré, et ainsi d'élaguer les chemins infaisables au plus tôt. *DPVS* effectue les pré-traitements suivants : considérons un programme P, une précondition pre et une postcondition $prop$.

1. *P est déplié k fois, noté P_{depl}*
2. *P_{depl} est traduit sous forme* DSA [4] *noté $DSA_{P_{depl}}$.*
3. *$DSA_{P_{depl}}$ est réduit à l'aide de techniques de* slicing *dépendant de $prop$, permettant de ne conserver que les instructions qui risquent de modifier (directement ou indirectement) la valeur des variables présentes dans la propriété à vérifier*
4. *Le GFC équivalent à $DSA_{P_{depl}}$ simplifié, noté G, est construit*
5. *les domaines de valeur des variables de G sont réduits en propageant les valeurs constantes dans G.*

Ensuite, G est exploré afin de générer un système de contraintes (SC) à la volée de la façon suivante :

1. *SC est initialisé à $pre \wedge \neg prop$*

3. Plus précisément, cette représentation n'est pas équivalente, mais un sur-ensemble des exécutions possibles.
4. Dynamic Single Assignment : transformation dans laquelle les variables sont affectées une seule fois par chemin d'exécution.

2. *Les nœuds de G sont ajoutés progressivement à SC. S'il s'agit d'un nœud d'instructions, la contrainte équivalente est ajoutée à SC. Si c'est un nœud conditionnel, cette condition est ajoutée temporairement à SC pour* vérifier *(et non résoudre) la faisabilité du système. S'il est inconsistant, le chemin correspondant dans G est coupé.*
3. *Lorsque la fin d'un chemin d'exécution est atteinte, SC est résolu. Si une solution est trouvée, il s'agit d'un contre-exemple, dans le cas contraire, le programme est correct sur ce chemin (pour la profondeur k).*

Précisons que les préconditions et les postconditions dans *DPVS* sont exprimées sous forme d'assertions. Ainsi, même si le principe s'inspire de la notion de *contrats* introduite par Eiffel [161] elle est moins expressive que des langages d'annotations comme ACSL pour le langage *C* [18], utilisé dans Frama-C ou JML pour le langage *Java* [42] utilisé par exemple dans les outils LOOP ou JACK.

La stratégie de recherche utilisée pour explorer le GFC est un choix important pour améliorer l'efficacité de l'approche. Par exemple, les premiers tests effectués avec une approche *depth first* naïve ne permettaient pas d'appréhender l'application *FlashManager* que nous verrons par le suite. Ainsi, pour être la meilleure possible, une stratégie de recherche doit : (1) couper les chemins infaisables au plus tôt, (2) s'il existe un contre-exemple, explorer le chemin correspondant le plus tôt possible. Nous allons maintenant décrire brièvement la stratégie de recherche qui a été utilisée pour analyser le *FlashManager*. Il s'agit d'une recherche dynamique en *backjump*. Cette approche a été publiée dans [J5] où le lecteur pourra trouver l'algorithme précis et des explications plus détaillées. Elle se base sur le fait que lorsque qu'un programme est sous forme SSA [5] [74] ou similaire (ce qui est le cas de la forme DSA utilisée, comme discuté dans [14]), alors il n'est pas nécessaire d'explorer le CFG de façon séquentielle, et ainsi un chemin peut être construit de façon dynamique. Dans le framework que nous avons proposé dans [J5], l'exploration de G est guidée par les variables. Cette idée s'inspire notamment des travaux de Rapps et al. sur la sélection de tests à partir d'un graphe de flot de données [190]. La stratégie commence par le nœud correspondant à la postcondition. Les nœuds suivants à être explorés sont ceux qui affectent des variables impliquées dans la postcondition, et ainsi de suite : à chaque étape, le nouveau nœud exploré contient (1) soit une affectation de variable directement impliquée dans la postcondition, (2) soit une variable impliquée indirectement dans la postcondition. Cette stratégie permet de détecter les inconsistances avant que tous les nœuds d'un chemin ne soient explorés, et par conséquent, elle permet de détecter les contre-exemples plus tôt. C'est cette approche qui a été implémentée dans l'outil *DPVS*, puis testée à l'aide du solveur de contraintes par défaut de l'environnement Comet [162], et du solveur z3 de Microsoft.

Plutôt que de fournir l'algorithme complet (que le lecteur trouvera dans [J5] s'il le souhaite), nous avons choisi d'expliquer son principe au travers d'un exemple décrivant les différentes étapes de l'algorithme.

Exemple 11. *Considérons le programme* `foo` *de la figure 3.1 et la propriété $p_1 : c >= d + e$. Les figures 3.2a et 3.2b montrent les chemins qui sont explorés par notre approche sur le GFC simplifié (à l'aide de techniques de slicing [6] [145]). A l'initialisation, le système de contraintes SC vaut $c_1 < d_0 + e_0$. Le nœud (12) [7] (par lequel toute exécution passe nécessairement) montre*

5. Static Single Assignment : transformation sémantiquement équivalente du programme dans laquelle chaque variable est affectée une seule fois.

6. Ayant permis de retirer les lignes 4 à 9 et 20 à 25 qui n'ont pas d'impact sur la propriété p_1.

7. La notation (n) est utilisée pour les nœuds correspondant à des instructions, et $\langle n \rangle$ pour les nœuds corres-

```
1 void foo(int a, int b)
2   int c, d, e, f;
3   if(a >= 0) {
4     if(a < 10) {
5       f = b - 1;
6     }
7     else {
8       f = b - a;
9     }
10    c = a;
11    if(b >= 0) {
12      d = a; e = b;
13    }
14    else {
15      d = a; e = -b;
16    }
17  }
18  else {
19    c = b; d = 1; e = -a;
20    if(a > b) {
21      f = b + e + a;
22    }
23    else {
24      f = e * a - b;
25    }
26  }
27  c = c + d + e;
28  assert(c >= d + e); // property p1
29 }
```

Figure 3.1 – Programme foo

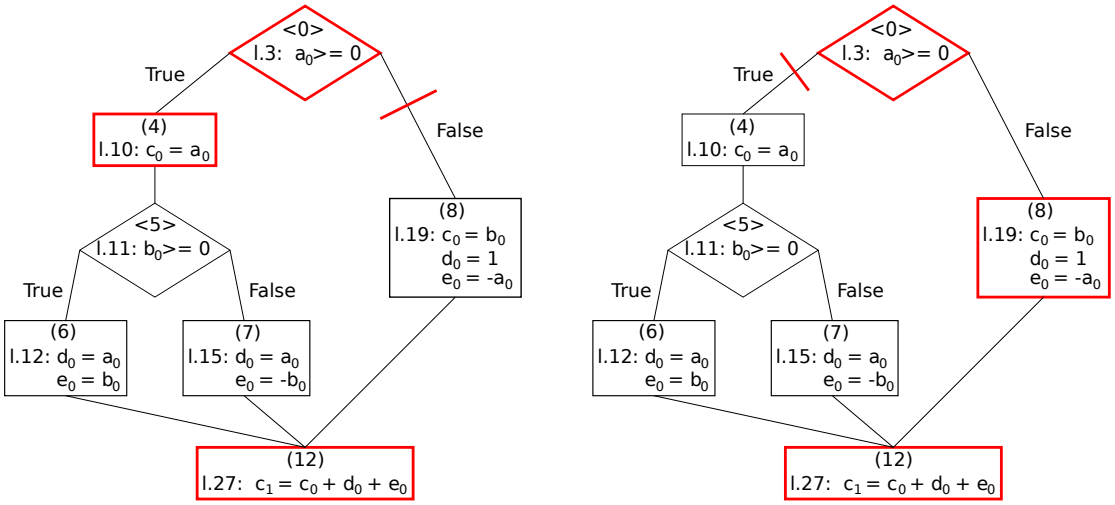
que c_1 dépend notamment de c_0. Par conséquent, l'algorithme de recherche sélectionne ensuite le nœud (4) où la variable c_0 est affectée. Pour atteindre le nœud (4), la condition du nœud $\langle 0 \rangle$ doit être vraie. Elle est donc ajoutée au système de contraintes SC, et l'autre alternative ($a_0 < 0$) est coupée. A ce stade (voir Fig. 3.2a) SC contient les contraintes suivantes : $\{c_1 < d_0 + e_0 \wedge c_1 = c_0 + d_0 + e_0 \wedge c_0 = a_0 \wedge a_0 \geq 0\}$ qui peut être simplifié en $\{a_0 < 0 \wedge a_0 \geq 0\}$. Ce système est inconsistant, et ainsi, on sélectionne maintenant le nœud (8), où la variable c_0 est aussi affectée. Pour atteindre, le nœud (8), la condition du nœud $\langle 0 \rangle$, doit être fausse. Ainsi, la négation de la condition est ajoutée à SC, et l'autre alternative est coupée. A ce stade (voir Fig. 3.2b), SC contient $\{c_1 < d_0 + e_0 \wedge c_1 = c_0 + d_0 + e_0 \wedge c_0 = b_0 \wedge a_0 < 0 \wedge d_0 = 1 \wedge e_0 = -a_0\}$ qui revient à $\{a_0 < 0 \wedge b_0 < 0\}$. Ce système de contraintes est consistant, et le solveur calcule une solution, e.g. $\{a_0 = -1, b_0 = -1\}$. Un contre-exemple de p_1 a donc été trouvé.

3.2 Une étude de cas industrielle dans un contexte temps-réel : le *FlashManager*

Dans le cadre du projet ANR TESTEC, nous avons eu l'opportunité de mettre à l'épreuve nos outils académiques sur des études de cas proposées par nos partenaires industriels. Ainsi, la société Geensoft, devenue depuis Dassault Systèmes, a mis à notre disposition une application temps-réel provenant d'un constructeur automobile : le *FlashManager*[8]. Nous avons donc cherché dans quelle mesure il était possible de vérifier des propriétés directement sur le code

pondant à des conditions du GFC des figures 3.2a et 3.2b.

8. En français : contrôleur de clignotants.



(a) GFC simplifié de p_1, étape 1

(b) GFC simplifié de p_1, étape 2

Figure 3.2 – Chemins explorés par la stratégie de recherche dynamique en *backjump* pour la propriété $p1$

du *FlashManager* à l'aide de l'outil *DPVS*, et comparé ces résultats à *CBMC*, un outil de BMC classique. Nous allons présenter ci-dessous les grandes lignes de cette étude de cas. Le lecteur intéressé par l'étude détaillée pourra la trouver dans [J5].

Le *FlashManager* est une application temps-réel du domaine de l'automobile qui a été modélisée et simulée à l'aide de la plate-forme *Simulink*. La société Geensys nous a aussi soumis quatre propriétés à vérifier sur le programme *C* correspondant. Il s'agit d'un problème complexe : le programme *C* étant généré automatiquement par un module propriétaire de *Simulink*, les fonctions à tester sont particulièrement longues et difficiles à comprendre. De plus, l'aspect temps-réel du code, notamment la gestion des cycles d'horloge, rend ce travail encore plus difficile. Au final, la société Geensys nous a fourni une fonction `f_manage` à tester, le modèle *Simulink* correspondant, et une description textuelle informelle des propriétés à vérifier sur cette fonction. La fonction `f_manage` utilise 81 variables booléennes, dont 6 entrées et 2 sorties, et 28 variables entières. Elle contient 300 lignes de code essentiellement formées de conditions imbriquées. Un appel à la fonction `f_manage` correspond à un cycle du module *Simulink* du *FlashManager* : les variables d'état et de sortie sont modifiées en fonction des entrées.

3.2.1 Description haut niveau du *FlashManager* et des propriétés attendues

Nous allons dans un premier temps décrire les principaux aspects du module *Simulink* du *FlashManager*[9]. Le *FlashManager* est un contrôleur qui commande les lampes clignotantes d'une voiture. En fonction des différentes valeurs entrées, il active une ou plusieurs lampes (gauche et/ou droite), sur une durée précise (le nombre de cycles d'horloge entre deux changements de front). La figure 3.3 montre le modèle *Simulink* simplifié décrivant les différentes entrées-sorties du système, et la figure 3.4 fournit le modèle plus détaillé. Le module contient

9. La description complète est disponible à l'adresse `http://hal.archives-ouvertes.fr/hal-00720921`.

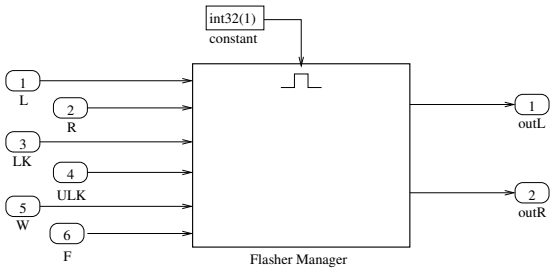


Figure 3.3 – Modèle *Simulink* simplifié du *FlashManager*

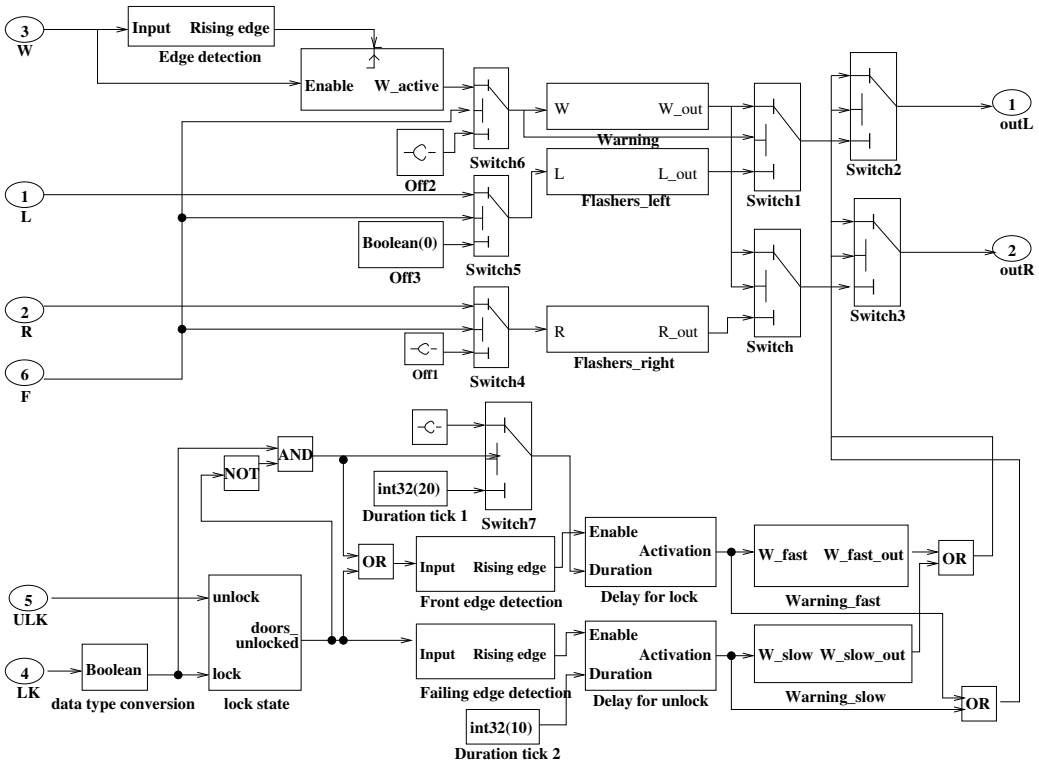


Figure 3.4 – Modèle *Simulink* détaillé du *FlashManager*

six entrées :

— `L` and `R` pour indiquer les changements de direction depuis la manette près du volant (Left ou Right), `LK` and `ULK` pour (dé)-verrouiller la voiture à partir de la télécommande (Lock ou Unlock), `W` pour (dés-)activer les feux de détresse (Warning), et `F` pour (dés-)activer le *FlashManager* globalement.

et deux sorties :

— `outL` et `outR` qui commandent les lumières gauche et droite.

Le *FlashManager* est utilisé dans trois situations usuelles : les changements de direction (fonctions `Flashers_left` et `Flashers_right`), le (dé-)verrouillage de la voiture (fonctions `Lock` et `Unlock`), et les feux de détresse (fonction `Warning`). Par exemple, lorsque le conducteur indique un changement de direction à droite, l'entrée booléenne `R` (resp. `L`) passe de 0 à 1. La lumière correspondante (commandée par `outR`, resp. `outL`) va alors osciller entre les états *on/off* sur une période de 6 unités de temps (correspondant par défaut à 3 secondes).

Ainsi, la séquence $[111000]$ est répétée sur la lampe droite (resp. gauche). Lorsque l'entrée `R` (resp. `L`) revient à 0, la lampe correspondante arrête de clignoter. Notons que la prise en compte d'une commande en entrée est immédiate.

Dans [J5], nous avons proposé une étude de cas sur quatre propriétés fournies par Geensoft, que nous avons cherché à vérifier directement sur le code *C* fourni. Pour éviter d'introduire une ambiguïté due à la traduction, nous avons choisi de les laisser en anglais :

— *Property 1 : "`Warning` function has priority over other flashing functions."*

— *Property 2 : "When the warning button has been pushed and then released, the `Warning` function resumes to the `Flashers_left` (or `Flashers_right`) function, if this function was active when the warning button was pushed."*

— *Property 3 : "When the `F` signal (for flasher active) is off, then the `Flashers_left`, `Flashers_right` and `Warning` functions are disabled. On the contrary, all the functions related to the lock and unlock of the car are maintained."*

— *Property 4 : "Lights should never remain lit infinitely."*

Afin de limiter l'ensemble des combinaisons possibles, nous avons adopté les restrictions suivantes pour vérifier ces propriétés, en accord avec les concepteurs du *FlashManager*. Il s'agit de situations qui sont normalement physiquement impossibles :

— Les entrées `L` et `R` ne peuvent pas être simultanément vraies sur un même cycle ;

— Les entrées `LK` et `ULK` ne peuvent pas être simultanément vraies sur un même cycle.

Dans la suite de ce chapitre, afin de ne pas alourdir son contenu, nous avons décidé de nous concentrer uniquement sur la propriété 4. Le principe est similaire pour les autres propriétés, et l'étude complète incluant ces dernières est disponible dans [J5].

3.2.2 Traduction des propriétés et résultats expérimentaux

Pour vérifier les propriétés à l'aide d'un outil de BMC, nous associons un programme *C* à chacune d'elle. A chaque fois, les boucles doivent être itérées un certain nombre de fois : un nombre trop grand rendrait le BMC impossible à réaliser, et un nombre trop faible le rend sans intérêt. En réalité, comme discuté précédemment, les outils de BMC augmentent généralement progressivement la borne de vérification, jusqu'à obtenir une erreur, ou dépasser les capacités de calcul de la machine.

La propriété 4 concerne un comportement du *FlashManager* sur une période de temps infinie. En pratique, nous ne pouvons vérifier qu'une version bornée de cette propriété : nous considérons donc que la propriété est violée si les lumières restent allumées pendant d périodes de temps consécutives. La programme *C* permettant de vérifier cette version bornée de la propriété 4 est donné figure 3.5. Nous y avons intégré une boucle bornée par d (ligne 8) qui compte le nombre de fois où les sorties du *FlashManager* ont été consécutivement vraies. A la sortie de la boucle, si cette valeur est égale à d (ligne 40), alors la propriété 4 a été violée dans le sens où la propriété est restée vraie pendant toute la période considérée. La valeur de la borne d est fixée la plus grande possible, sachant que sa valeur maximale dépend des capacités de la machine.

Le tableau 3.1 et la figure 3.6 résument les résultats obtenus lors de nos expérimentations. Tous les détails sont fournis dans [J5]. Pour chaque propriété, nous avons comparé les résultats

```
1 void prop4(int d) {
2   init();
3   // number of time where the left light has been consecutively true
4   int countL = 0;
5   // number of time where the right light has been consecutively true
6   int countR = 0;
7   // consider d units of time
8   for(int i=0;i<d;i++) {
9     // non-deterministic values of the inputs
10     L=nondet_in();
11     if (L)
12       R=FALSE;
13     else
14       R=nondet_in();
15     W=nondet_in();
16     LK=nondet_in();
17     if (LK)
18       ULK=FALSE;
19     else
20       ULK=nondet_in();
21     F=nondet_in();
22     // call to f_manage() to simulate one pass through the module
23     f_manage();
24     if (outL)
25     // the left light has been consecutively true
26     // one more time
27       countL++;
28     else
29     // the left light has not been consecutively true
30       countL=0;
31     if (outR)
32      // the right light has been consecutively true
33      // one more time
34       countR++;
35     else
36      // the right light has not been consecutively true
37       countR=0;
38   }
39   // if countL and countR are less than d, then the lights did not remain lit
40   assert (countL<d && countR<d);
41 }
```

Figure 3.5 – Fonction à vérifier pour la propriété 4

avec l'outil de référence *CBMC*. Toutes les propriétés ont été formulées comme des propriétés de *safety* (quelque chose de mauvais ne doit pas arriver). Nous avons fixé initialement une borne de 10 pour le nombre de dépliages, et nous l'avons augmentée progressivement jusqu'à atteindre un timeout. Pour chaque expérience, une limite de 10 minutes a été fixée. En cas de dépassement, l'exécution est stoppée et le résultat est considéré comme un timeout ($T.O._n$ avec $n \in \mathbb{N}$ signifiant que le timeout a eu lieu pour n dépliages). Toutes les mesures ont été faites sur une plate-forme 64-bit Linux à 3.16 GHz avec 16 GB de RAM. La version de *CBMC* utilisée au moment de ces expérimentations était la version 3.6 en 32 bits.

En ce qui concerne la propriété 4, elle est fausse s'il est possible de trouver une combinaison d'entrées qui maintienne la lampe toujours allumée. Comme discuté auparavant, nous sommes dans un contexte borné, donc il est seulement possible de montrer qu'elle reste allumée sur un certain nombre de dépliages, le plus grand possible, mais pas infiniment. Nous avons finalement montré qu'il était possible de maintenir la lampe allumée en permanence sur 1600 dépliages, ce qui traduit en nombre de cycles revient à environ 13 minutes d'utilisation du *FlashManager*. Notons qu'aucun des deux outils de BMC utilisés n'a pu appréhender la propriété 3 à cause d'une explosion combinatoire. Nous avons alors introduit une autre propriété 3b, se concentrant

Tableau 3.1 – Tableau global des résultats expérimentaux

Property	*DPVS*	*CBMC*
1	False , Few seconds	False, Few seconds
2	Intractable	Intractable
3	Intractable	Intractable
3b	True until 400 unfoldings	$T.O._{400}$
4	False until 1600 unfoldings	$T.O._{200}$

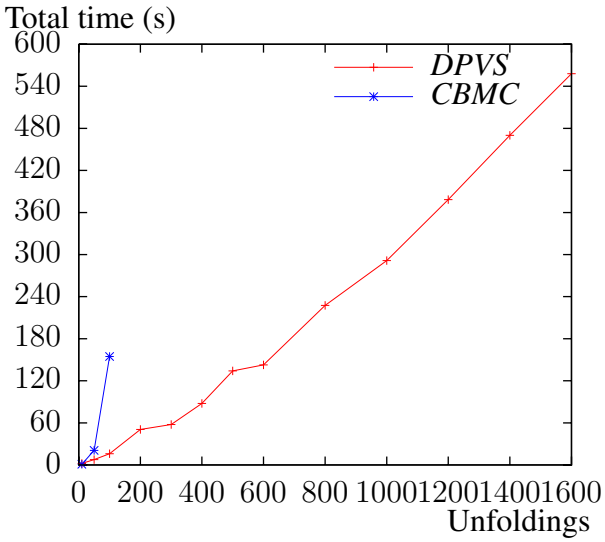


Figure 3.6 – Temps total nécessaire (en secondes) pour vérifier la propriété 4 en fonction du nombre de dépliages

sur les fonctions `Warning`, `Flashers_left` et `Flashers_right` (première partie de la propriété), mais ne prenant pas en compte les fonctionnalités de (dé-)verrouillage (deuxième partie de la propriété).

Le tableau 3.1 montre que sur les propriétés 1, 2 et 3, les performances entre *DPVS* et *CBMC* sont similaires. En revanche, *DPVS* surpasse *CBMC* sur les propriétés 3*b* et 4. On peut donc estimer que l'approche que nous avons proposée est pertinente dans un cadre temps-réel, même si des études plus poussées auraient été nécessaires pour s'en assurer.

3.3 Conclusion et perspectives

Les contributions de ce chapitre ont été réalisées en 2011, et publiées en 2013. Malgré des résultats significatifs, ils ont été vite concurrencés par les approches par interprétation abstraite, notamment celle proposée par Frama-C [139], qui permettent d'exprimer des propriétés en ACSL plus riches, et d'obtenir de très bons résultats pour vérifier du code. Notons toutefois que ces travaux, et notamment notre étude de cas sur le *FlashManager*, ont été repris dans des travaux récents de Cabrera-Castillos et al. [50] sur la vérification de modèles Simulink. L'approche qu'ils proposent utilise directement la spécification du *FlashManager* ainsi que la propriété à vérifier au format Simulink, et à la manière de l'approche concolique [113, 202],

utilisent les contre-exemples obtenus aux phases précédentes pour activer de nouveaux chemins dans la spécification, et ainsi obtenir de nouveaux contre-exemples.

Ces travaux pourraient faire l'objet d'extensions diverses. Par exemple, une des faiblesses de l'outil *DPVS* c'est le manque d'expressivité des assertions qu'il est possible de vérifier. Ainsi, il serait utile de réfléchir à des possibilités d'enrichir son pouvoir d'expression à l'aide d'un langage, à la manière d'ACSL ou JML par exemple. Par ailleurs, l'approche proposée suppose que l'utilisateur a accès à l'ensemble du code source, ce qui peut poser problème e.g. dans le cas d'appel de fonctions système. L'apport de techniques symboliques, voire concoliques [113, 202], permettrait vraisemblablement d'améliorer ce point. Il serait aussi utile de fournir un framework d'aide à la détection mais aussi à la localisation d'erreur. Des premiers travaux dans ce sens ont été proposés par Bekkouche et al. dans [21]. De façon plus pragmatique, cet environnement pourrait s'appuyer sur les travaux de test mutationnel, pour muter les conditions ou les affectations et voir si cette modification permet de satisfaire les propriétés recherchées, en ciblant en priorité les conditions suspectes (par exemple des opérations sur des nombres flottants, ou des instructions suspectes extraites à la manière de l'outil *LocFaults* [21]). En outre, l'approche utilisée dans les travaux décrits dans cette section se base sur une technique de slicing assez ancienne et non optimale. Il serait donc judicieux d'améliorer cette phase, notamment en utilisant la technique proposée récemment par Léchenet et al. dans [145].

Dans une perspective de test logiciel, il nous semble que l'adaptation de l'outil *DPVS* dans l'optique d'en créer un outil de génération de données de test à base de code source nécessiterait un investissement faible. L'idée serait de fixer les entrées du programme comme indéterminées, et de rajouter des assertions trivialement fausses au niveau des instructions de code que l'on souhaiterait couvrir, et ainsi l'outil fournirait (quand c'est possible) un contre-exemple permettant d'obtenir un cas de test pour chaque assertion. Cela permettrait ainsi un prototypage rapide pour obtenir un outil, mais ce dernier n'aurait évidemment pas les performances d'outils aboutis comme Cute ou Pathcrawler par exemple.

Chapitre 4

Enforcement à l'exécution de propriétés (temporisées)

Une fois qu'un système est en production, il peut devenir assez difficile d'y intégrer des corrections, que ce soit parce que la moindre modification nécessite a priori de repasser par l'ensemble des phases de développement du système, ce qui est coûteux, ou tout simplement parce que la spécification complète du système n'est pas (ou plus) disponible si ce dernier a été développé il y a un certain temps. Dans cette situation, il est possible de recourir aux méthodes de vérification à l'exécution, en intégrant un dispositif dans le système pour surveiller le respect de certaines propriétés à tout moment, et remonter des alertes en cas de problème. L'idée sous-jacente est d'utiliser un "moniteur" pour vérifier si une trace vérifie une propriété P sans modifier le comportement du système. L'intérêt de cette approche, c'est qu'il n'est pas nécessaire de posséder la spécification complète du système pour la mettre en œuvre, seule la propriété à vérifier est nécessaire. Cependant, dans beaucoup de situations, le fait de remonter des alertes est insuffisant, et le concepteur du système souhaiterait effectuer des corrections sur le système "à la volée" durant son exécution. Il peut dans ce cas envisager d'utiliser un mécanisme d'*enforcement à l'exécution (runtime enforcement)* (noté *EE* par la suite), dans lequel un mécanisme d'enforcement va agir comme une sorte de filtre (à la manière d'un firewall laissant passer ou non des requêtes) et peut être amené à modifier l'exécution dans un cadre prédéfini afin de garantir la (ou les) propriété(s) en sortie. Il est pourvu d'une mémoire et est capable de stocker des événements. Si c'est possible, il relâche les actions stockées en mémoire, assurant que la propriété souhaitée est vérifiée. Dans ce cas, le mécanisme d'enforcement (noté *EM* par la suite, pour *Enforcement Mechanism*) modifie l'exécution du système. Les séquences de sortie de l'*EM* doivent être "correctes" (les séquences vérifient la propriété) et "transparentes" (les entrées correctes ne sont pas modifiées, ceci afin de garantir une qualité de service). Selon le type de propriétés et les capacités de modification de l'*EM*, il est envisageable de le générer automatiquement. Ce genre de mécanisme est envisageable dans nombre de cas où il est possible d'identifier une situation incohérente, et de contraindre le système à l'éviter. Il peut s'agir par exemple d'interdire certaines positions dangereuses pour un bras robot comme dans les travaux de Machin et al. [155], ou encore éviter la situation où un avion de ligne aurait les aérofreins déployés tout en activant la poussée des réacteurs vers l'avant.

Ce chapitre présente des travaux auxquels j'ai participé autour de l'enforcement de propriétés temporisées. Dans ces derniers, nous avons décrit les fondements théoriques permettant d'étendre les approches classiques d'enforcement dans un cadre temporisé, et nous avons pro-

posé un framework complet pour enforcer des propriétés temporisées régulières quelconques décrites par un TA déterministe, et un autre qui prend en compte des événements incontrôlables, i.e. que le mécanisme d'enforcement ne peut qu'observer, mais pas modifier. Nous avons proposé une modélisation précise du problème de l'enforcement dans un cadre temporisé, en prenant des décisions concernant la définition de certaines contraintes comme la correction, ou la transparence, et nous avons proposé une synthèse de mécanisme d'enforcement qui respecte ces contraintes de façon optimale. Notre mécanisme utilise un buffer pour stocker les événements dangereux, et agit ainsi comme un filtre retardant (parfois à l'infini si besoin) ces événements. Ce mécanisme a été décrit à deux niveaux d'abstractions : fonctionnel et opérationnel (i.e. sous forme de système de transition). A notre connaissance, nous avons été les premiers à proposer un mécanisme d'enforcement pour des propriétés temporisées génériques. Ces travaux ont été adaptés pour gérer des événement incontrôlables, en utilisant notamment une approche à base de théorie des jeux. Ils ont été implémentés dans des prototypes, montrant la viabilité des concepts. Ce chapitre reprend bien entendu nos publications sur le sujet, mais un effort de formalisation, d'harmonisation et de remise à plat de ces différents travaux y a été fait. En effet, ces derniers utilisaient des approches, notations et définitions assez différentes. Le lecteur retrouvera aussi dans ce chapitre le cheminement intellectuel et chronologique qui a été effectué durant ces six dernières années, allant du problème de l'*EE* uniquement de propriétés de safety jusqu'à la prise en compte de propriétés régulières quelconques avec événements incontrôlables. Ce chapitre a été conçu en quelques sortes comme un article scientifique tutoriel synthétisant nos travaux sur l'*EE*. Contrairement aux chapitres précédents, il a notamment été décidé de rentrer dans les détails de formalisation, qui de notre point de vue, font partie de nos contributions. Chaque technique d'*EE* est décrite selon le même principe : un exemple intuitif du comportement souhaité du mécanisme d'enforcement sur une entrée simple est fourni, une représentation de ce mécanisme sous forme fonctionnelle et/ou opérationnelle est discutée (de façon plus ou moins détaillée), un exemple d'évolution de l'état de ce dernier est donné, et enfin les propriétés qu'il vérifie sont brièvement énoncées.

Ces travaux ont été effectués entre autre dans le cadre du projet ANR VACSIM, et d'un projet co-financé par la région Aquitaine, Bordeaux INP, et le GIS Albatros. Ils sont le fruit d'une collaboration avec Yliès Falcone, Thiérry Jéron, Hervé Marchand, Omer Nguena-Timo, Srinivas Pinisetty, et Matthieu Renard et ont généré les publications suivantes : [C25, C24, C23, J6, C21]. Certaines contributions de la thèse de Matthieu Renard ont été discutées avec des membres de Thalès Avionics, notamment Patrice Capircio et Jean-Noël Perbet. Précisons que Thalès Avionics a participé au financement de ces travaux via le GIS Albatros. Par ailleurs, dans le cadre de l'action Européenne COST ARVI (dont je suis membre représentant la France avec Yliès Falcone), l'ensemble des partenaires a publié un livre tutoriel sur la vérification à l'exécution. J'ai participé à la rédaction de ce livre, notamment en étant co-auteur avec Yliès Falcone, Leonardo Mariani et Saikat Saha du chapitre *Runtime Failure Prevention and Reaction* [B1].

L'organisation de ce chapitre est la suivante :

— La section 4.1 rappelle les principes généraux concernant l'enforcement à l'exécution.

— La section 4.2 présente nos contributions concernant l'enforcement à l'exécution de propriétés temporisées, en se focalisant dans un premier temps sur les propriétés de safety, puis sur les propriétés régulières quelconques.

— La section 4.3 s'intéresse à l'enforcement à l'exécution de propriétés en présence d'événements incontrôlables. Elle décrit d'abord comment enforcer une propriété non tempo-

risée quelconque, puis comment il est possible de simplifier cette approche et d'améliorer ses performances grâce à la théorie des jeux, enfin elle montre comment ces travaux peuvent être étendus dans un cadre temporisé. Cette section est essentiellement issue des travaux de thèse de Matthieu Renard [193] dont j'ai assuré la co-direction.

4.1 Quelques généralités sur la vérification et l'enforcement à l'exécution

(a) Schéma général d'un moniteur de vérification, fournissant une séquence de verdicts $V(\sigma)$ en sortie, indiquant si σ satisfait φ.

(b) Schéma général d'un mécanisme d'enforcement, fournissant une séquence d'événements en sortie modifiée $E(\sigma)$ satisfaisant φ.

Figure 4.1 – Schéma montrant la différence conceptuelle entre un moniteur de vérification M et un mécanisme d'enforcement *EM*.

Cette section survole quelques éléments de base concernant la vérification et l'enforcement à l'exécution. Un lecteur souhaitant approfondir ces notions et accéder à un état de l'art détaillé pourra se référer au livre rédigé par l'action COST ARVI [97] évoqué ci-dessus.

La *Vérification à l'Exécution* [148, 99] (*Runtime Verification*, noté *RV* par la suite) est un terme regroupant les théories, techniques et outils pour vérifier la conformité des exécutions du *système sous surveillance* (par la suite on notera *SUS* pour *System Under Scrutiny*), par rapport à une spécification donnée. Le principe décrit figure 4.1a consiste à exécuter un mécanisme appelé généralement *moniteur* qui fournit des verdicts à partir d'une séquence d'événements produite par le *SUS* via une instrumentation, et ce par rapport à une propriété qui formalise la spécification, sous forme d'automate ou de formule de logique temporelle par exemple. Même si on peut considérer que la *RV* puise ses origines dans les techniques de model-checking (et aussi par certains aspects du test), Leucker et Schallhart résument très bien les principales différences conceptuelles entre ces deux techniques [148] :

— en model-checking, toutes les exécutions du système sont examinées. Par conséquent, le fait de vérifier si un système vérifie une propriété correspond généralement à un problème d'inclusion de langage, alors qu'en *RV* il s'agit d'un problème de mot.

— en model-checking, on considère généralement des traces infinies, alors qu'en *RV*, on considère des exécutions finies (car ces dernières le sont nécessairement).

— en model-checking, en raison du fait que le modèle complet est disponible, on considère généralement une position arbitraire sur les traces. En *RV* (notamment *online*, i.e. à la volée), on considère des exécutions de taille croissante : un moniteur va donc analyser les exécutions de manière incrémentale.

Par ailleurs, la *RV* et le test ont de nombreux points communs, du fait notamment qu'ils ne considèrent qu'un sous-ensemble des exécutions possibles du *SUS*. Dans le cas du test (actif)

à base de modèles, on retrouve aussi la notion de verdict, qui est présente en *RV* (la propriété est vérifiée ou non). Cependant, la différence majeure entre la *RV* et les techniques de test à base de modèle comme par exemple dans TGV [132], c'est qu'en *RV* on ne s'intéresse pas à la façon de générer des données de test, mais plutôt à la façon de monitorer le système afin d'obtenir des alertes en cas de dysfonctionnement. Il s'agit d'une approche comparable au test passif [51, 5, 52, 55, 171]. Ainsi, les notions d'exhaustivité ou de couverture n'interviennent généralement pas en *RV*, dans laquelle le testeur ou le moniteur n'intéragit pas avec le *SUS* mais se contente de l'observer. La technique qui met en œuvre ce concept est souvent appelée *monitoring*. Notons qu'à l'origine, le test passif est plutôt une méthode dont l'objectif est de rendre possible le test pour un système dont le test actif est difficilement envisageable (e.g. système à forte mobilité), alors que l'objectif de la *RV* est plutôt de monitorer des systèmes déjà en service. Cependant, les techniques utilisées sont finalement très similaires. On trouve généralement deux familles de problèmes liés à la *RV* : des problèmes d'ordre théorique, comme le type de propriétés qu'il est possible de monitorer, et comment synthétiser le moniteur [98, 30]. Ces travaux s'appliquent généralement sur des abstractions de haut niveau ; et des problème liés à l'instrumentation, i.e. où et comment récupérer les informations permettant d'analyser le système [56, 73]. Il faut noter que ces problématiques d'instrumentation ne sont pas propres à la *RV*, et qu'on les retrouve aussi dans les travaux liés au test formel.

L'*enforcement* (de propriétés) à l'exécution, parfois aussi appelé *imposition* à l'exécution, est une extension de la *RV*. La différence entre les deux techniques, c'est que l'enforcement peut modifier l'exécution du système si besoin. Ainsi, sous réserve de respecter un ensemble de règles, l'*EM* va transformer une séquence d'exécution incorrecte en une autre séquence qui vérifie la propriété requise. Les travaux sur l'*EE* font généralement abstraction des détails d'implémentation, et notamment sur la façon effective de modifier l'exécution. Cela revient finalement à définir une *relation d'entrées-sorties* sur des séquences d'événements (cf figure 4.1b) : un framework d'*EE* va décrire comment transformer une séquence d'événements en entrée (possiblement incorrecte par rapport à la spécification) en une séquence d'événements en sortie à l'aide d'un *EM*. Cette transformation est réalisée selon la spécification qui va être utilisée pour synthétiser l'*EM*. D'un point de vue abstrait, cet *EM* peut être par exemple un transducteur, ou un modèle plus complexe. De façon usuelle, la relation d'entrée-sortie réalisée par l'*EM* doit satisfaire les deux contraintes suivantes [200, 19] :

— *Correction (soundness)* : la séquence en sortie satisfait la propriété
— *Transparence (transparency)* : la séquence de sortie doit être la plus proche possible de la séquence d'entrée (et par conséquent une séquence d'entrée correcte ne doit pas être modifiée en sortie).

Cette notion de transparence est adoptée dans la majorité des travaux sur le sujet, mais elle laisse beaucoup de liberté sur la façon de corriger les séquences erronées. Par ailleurs, on trouve dans cette définition une notion implicite d'optimalité. Par exemple, [200, 19, 98] satisfont la transparence en faisant en sorte que la sortie soit le plus grand préfixe de l'entrée qui respecte la propriété. Lorsque les propriétés deviennent plus compliquées, par exemple avec du temps, il est plus lisible de séparer la notion initiale de transparence en deux contraintes séparées. La première concerne les modifications autorisées sur l'exécution. Par exemple la sortie peut être nécessairement un préfixe de l'entrée. L'autre définit selon quels critères le résultat est le meilleur possible. C'est l'*optimalité*. Par exemple, la sortie peut être le plus long préfixe de l'entrée respectant la propriété. Dans nos travaux, nous avons choisi de séparer explicitement

ces deux aspects. En outre, on trouve généralement une dernière contrainte, souvent implicite. Il s'agit de la *contrainte physique* : un événement déjà sorti par l'*EM* ne peut être modifié. Cette contrainte, a priori évidente, doit parfois être prise en compte dans le cas d'une modélisation formelle.

Dans nos travaux, nous nous intéressons à l'*EE* de propriétés. Pour identifier ces dernières, nous nous sommes basés sur la hiérarchie *Safety-Progress* (SP) proposée par Pnueli et al. dans [53]. Il s'agit d'une hiérarchisation des propriétés sur des séquences infinies qui se décline sous différentes vues : théorie des langages (des ensembles de séquences), logique (formules LTL), topologique (ensembles ouverts ou fermés) ou automates (mots acceptés par un automate de Streett [209]). Pnueli et al. y distinguent quatre classes de propriétés. Les (propriétés de) *safety* : lorsque qu'une séquence satisfait une propriété, tous ses préfixes la satisfont. Les *guarantee* ou *co-safety* : lorsque qu'une séquence satisfait une propriété, il existe (au moins) un préfixe qui la satisfait. Les *response* : lorsque qu'une séquence satisfait une propriété, un nombre infini de ses préfixes la satisfont. Et enfin les *persistence* : lorsque qu'une séquence satisfait une propriété, tous ses préfixes à partir d'un certain rang la satisfont, i.e. un nombre fini de ses préfixes ne la satisfont pas. Cependant, dans nos travaux, nous considérons des exécutions de programme, ce qui implique que les séquences manipulées sont nécessairement finies. La hiérarchie SP étant conçue pour des séquences infinies uniquement, nous nous sommes en réalité basés sur l'extension proposée par Falcone dans [95], qui intègre uniformément les séquences de longueur finie dans cette classification. En s'appuyant sur ces travaux, nous nous sommes concentrés dans un premier temps sur les propriétés de safety (quelque chose de mauvais ne doit jamais se produire) et de co-safety (un état bon doit être atteint en un temps fini, et une fois atteint, le système y reste) modélisées par un TA déterministe. Ces résultats sont publiés dans [C21]. Ensuite, nous nous sommes intéressés à tout type de propriété régulière, i.e. modélisable à partir d'un TA déterministe. Ces résultats sont publiés dans [J6, C23, C24, C25].

4.2 Enforcement à l'exécution de propriétés temporisées

La première approche d'*EE* (non temporisé) a été proposée par Schneider et al. dans [200]. Elle permet d'enforcer [1] des propriétés décrites sous forme d'automates de Büchi. Le formalisme utilisé pour représenter cet *EM* est un *automate de sécurité*, une sorte de machine à états finis qui est exécutée en parallèle avec le *SUS*, permettant ainsi d'arrêter l'exécution dès que la propriété risque d'être violée. Cette approche se limite aux propriétés de safety. Charafeddine et al. proposent dans [54] un *EM* autorisant de dévier d'une propriété jusqu'à k étapes. Ensuite, si le *SUS* ne revient pas dans un état correct, l'*EM* ramène le système dans son dernier état correct, et le contraint à explorer d'autres exécutions. Sur le même principe, Bloem et al. présentent dans [30] un framework permettant de synthétiser un *EM* (le *shield)*, à partir d'un *automate de safety* en résolvant un jeu de safety à deux joueurs. Le shield est modélisé par une machine de Mealy [160]. Grâce à une notion de distance entre l'entrée et la sortie, il permet d'assurer une sortie correcte et qui dévie le moins possible par rapport à l'entrée. Une extension de cette approche a été proposée par Wu et al. dans [218], utilisant le même principe, à ceci prêt que l'*EM* est capable de gérer des erreurs en rafale. Avec une technique utilisant aussi des jeux mais avec une approche plutôt héritée de la théorie du contrôle [189], on peut aussi citer les travaux de Machin et al. [156] : à partir d'un ensemble d'invariants de safety, et d'un ensemble

1. Nous utiliserons souvent cet anglicisme par la suite, ce dernier étant difficile à traduire en français.

d'interventions correctives possibles (des *inhibitions* ou des *actions*), les auteurs montrent comment synthétiser une stratégie à la fois *permissive* et *sûre*, en utilisant dans un premier temps un algorithme adhoc et le model-checker NuSMV, puis en utilisant l'outil *Uppaal-Tiga*. Cette stratégie est ensuite appliquée par un moniteur *online*.

Dans [149, 151], Ligatti et al. proposent de synthétiser des *edit-automata* à partir d'automates à états finis. Par rapport aux travaux précédents, ils ajoutent à l'*EM* la possibilité d'insérer ou de supprimer des événements au cours de l'exécution du système, et d'utiliser une mémoire pour stocker le suffixe des exécutions invalides en attendant qu'elles deviennent à nouveau valides. Le fait d'utiliser ces nouvelles primitives augmente la capacité d'enforcement de l'*EM*, et permet ainsi d'enforcer un sur-ensemble des propriétés de safety, et plus précisément l'ensemble des propriétés dites *renewal*. Notons que dans les travaux de Ligatti et al. ou Schneider, il n'y a pas de séparation claire entre la spécification de la propriété à garantir et le mécanisme d'enforcement, ce qui rend la mise en application de l'enforcement difficile. Cette distinction est en revanche faite dans les travaux de Falcone et al. qui proposent dans [98, 95] un framework permettant d'enforcer aussi les propriétés de *response*. Ils reprennent l'idée d'utiliser une mémoire, et montrent qu'il est possible de définir une moniteur d'enforcement généralisé (GEM) comme une instance de machine de Mealy avec des opérations *store* et *dump*, permettant respectivement de mémoriser et de relâcher un événement. C'est ce principe que nous utiliserons pas la suite, avec cependant un modèle plus complexe qu'une machine de Mealy.

A notre connaissance, il existe peu de travaux autour de l'*EE* de propriétés temporisées qui ont précédé nos contributions sur le sujet. Dans [158], Matteucci et al. utilisent des techniques de model-checking partiel pour synthétiser un *contrôleur d'opérations* avec une algèbre de processus. L'*EM* ainsi obtenu correspond aux automates de sécurité de Schneider [200]. Le système est décrit sous forme de processus temporisés à l'aide de CSS. Cette approche considère le temps de façon discrète. Plus récemment, Basin et al. proposent dans [17, 16] une approche d'enforcement de politiques de sécurité. Ils discutent notamment les problèmes d'enforçabilité, et décrivent une méthode de synthèse d'*EM* pour plusieurs formalismes (à base de logique ou d'automate). De façon similaire à [200] le moniteur observe le système et l'arrête en cas de violation de propriété. Cette approche se base sur du temps discret : elle utilise des événements incontrôlables particuliers, les *tick* d'horloge, pour modéliser l'écoulement du temps.

Alors qu'il existe beaucoup d'outils pour la vérification à l'exécution [15], il existe assez peu d'outils d'*EE*. La plupart sont basés sur la théorie de Schneider [200]. On peut citer Polymer [19], un langage et un framework pour la définition et la composition d'*EM* dans des applications Java ou encore Mobile [119] permettant d'enforcer des propriétés sous .NET. Concernant les outils liés à l'instrumentation de code, on peut citer JavaMOP [56], ou encore Nomad [73] qui utilisent la programmation par Aspect. Le lecteur pourra se référer au livre de l'action COST ARVI [B1] évoqué précédemment pour un état de l'art détaillé.

Dans cette section, nous nous intéressons à l'enforcement de propriétés temporisées modélisées par des TA déterministes [6]. De façon évidente, utiliser des modèles temporisés permet une plus grande richesse d'expression de propriétés, et permet par exemple de décrire des propriétés comme *"il doit y avoir un délai minimum de 2 secondes entre deux requêtes"*. L'idée de base consiste à s'inspirer des travaux de Ligatti et al. [149, 150] et Falcone et al. [98], notamment en utilisant une mémoire au sein de l'*EM*, et d'étudier la possibilité de les étendre dans un cadre temporisé. Toutefois, en raison du fait que nous étions les premiers à aborder cette question sous cet angle, nous avons été amenés à prendre des décisions sur la signification des contraintes de correction, de transparence et d'optimalité dans un contexte temporisé. Nous

les avons donc adaptées, et opté pour un *EM* utilisant une mémoire interne, et agissant comme un retardateur temporel [2]. Nous avons proposé dans [C21] un nouveau framework théorique permettant de synthétiser un *EM* pour des propriétés de safety et de co-safety décrites par des TA avec des états acceptants et non-acceptants (parfois appelés "Good" et "Bad"). Comparé aux approches précédentes d'*EE* comme [150, 98] nous ajoutons à l'*EM* la capacité de retarder les événements en entrée : les délais sont modifiés par l'*EM* si besoin en utilisant un buffer dans lequel les événements sont stockés, puis relâchés après le délai approprié afin que la sortie de l'*EM* vérifie la propriété. Ce choix a été fait en se basant sur des exemples d'applications réelles. Toutefois, une personne souhaitant utiliser d'autres règles d'enforcement pourrait aisément adapter ce framework avec de nouvelles primitives. Par rapport aux travaux de Matteucci et al. d'une part [158] ou Basin et al. d'autre part [17, 16], nous proposons une approche qui considère le temps de façon dense, utilisant le pouvoir expressif des TA et les bibliothèques *Uppaal* [20], et utilisant des primitives d'enforcement plus riches. De plus, dans notre cas, le mécanisme d'enforcement arrête l'exécution uniquement si le fait de retarder des événements ne suffit pas à vérifier la propriété requise. Finalement, dans [J6] nous avons étendu ces travaux à tout type de propriété régulière modélisée par un TA déterministe.

4.2.1 Enforcement dans un contexte temporisé

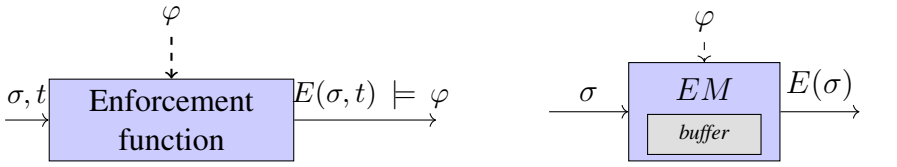
Nous décrivons maintenant nos contributions concernant l'enforcement de propriétés temporisées. Nous présentons de manière synthétique les résultats publiés dans [C21] et [J6]. Le lecteur intéressé pourra trouver dans [C21] une description détaillée de notre framework d'enforcement pour des propriétés de safety et de co-safety temporisées, et dans [J6] une généralisation de ces travaux à toute propriété régulière décrite par un TA déterministe. Dans cette section, afin d'éviter des définitions lourdes, nous avons choisi de ne détailler les descriptions formelles que pour l'enforcement de propriétés de safety, et d'expliquer ensuite de manière plus intuitive les différences pour les autres types de propriétés.

Voici quelques notations utilisées par la suite. Soit $\mathbb{R}_{\geq 0}$ l'ensemble des nombres réels positifs, et Σ un alphabet fini d'*actions*. Une paire $(\delta, a) \in (\mathbb{R}_{\geq 0} \times \Sigma)$ est appelée un *événement*. On note $\mathrm{del}(\delta, a) = \delta$ et $\mathrm{act}(\delta, a) = a$ les projections des événements sur les délais et sur les actions respectivement. Un *mot temporisé* sur Σ, noté $\mathrm{tw}(\Sigma)$, est une séquence finie d'événements de $(\mathbb{R}_{\geq 0} \times \Sigma)$. Pour $\sigma = (\delta_1, a_1) \cdot (\delta_2, a_2) \cdots (\delta_n, a_n)$, δ_i $(2 \leq i \leq n)$ est le délai entre a_{i-1} et a_i, et δ_1 le temps écoulé avant le premier événement. La notion de préfixe s'étend naturellement aux mots temporisés. La *somme des délais* d'un mot temporisé σ est notée $\mathrm{time}(\sigma)$. Etant donnés $t \in \mathbb{R}_{\geq 0}$ et un mot temporisé $\sigma \in \mathrm{tw}(\Sigma)$, on définit l'*observation de σ en temps t* comme le mot temporisé $\mathrm{obs}(\sigma, t) \triangleq \max\{\sigma' \mid \sigma' \preccurlyeq \sigma \wedge \mathrm{time}(\sigma') \leq t\}$, i.e., le plus long préfixe de σ avec une somme de délais inférieure à t. La *projection non temporisée* de σ est $\Pi_\Sigma(\sigma) \triangleq a_1 \cdot a_2 \cdots a_n$ dans Σ^* (i.e. on ignore les délais). Un *langage temporisé* est un sous-ensemble quelconque de $\mathrm{tw}(\Sigma)$. On définit la relation d'ordre suivante sur les mots temporisés : σ' est un *préfixe retardé* σ (notée $\sigma' \preccurlyeq_d \sigma$) si $\Pi_\Sigma(\sigma') \preccurlyeq \Pi_\Sigma(\sigma)$ et $\forall i \leq |\sigma'|, \mathrm{del}(\sigma(i)) \leq \mathrm{del}(\sigma'(i))$.

Dans notre framework, l'*EM* est une sorte de boîte recevant une séquence d'événements en entrée σ, possiblement incorrecte, et envoyant une séquence de sortie o correcte selon une propriété φ, comme dans Falcone et al. [98]. Comme discuté précédemment, il s'agit d'une fonction de transformation entre les entrées et les sorties mais dans un cadre temporisé. D'un

2. Traduction sûrement améliorable de "time-retardant".

point de vue fonctionnel, on parlera d'*entrée* comme paramètre de la fonction d'enforcement, et de *sortie* comme l'image de cette fonction. Les séquences sont décrites par des mots temporisés, et les propriétés par des TA. De plus, comme l'enforcement se fait à la volée, le temps courant est aussi nécessaire. Ainsi, comme illustré figure 4.2a, la fonction d'enforcement E est une fonction de $\mathrm{tw}(\Sigma) \times \mathbb{R}_{\geq 0}$ dans $\mathrm{tw}(\Sigma)$ pour une propriété φ donnée. A cela, il est nécessaire d'ajouter deux *contraintes physiques* : (1) lorsque le temps s'écoule, la fonction d'enforcement ne peut qu'ajouter des événements à sa sortie (pas modifier ceux déjà émis) ; (2) la durée totale de la séquence de sortie ne peut dépasser le temps courant. Ces contraintes sont intégrées directement dans la définition de fonction d'enforcement [3].



(a) Fonction d'enforcement E

(b) Vue conceptuelle de l'*EM* équipé d'un buffer

Figure 4.2 – Vues conceptuelle et fonctionnelle d'un *EM*

Selon le même principe que [98, 150], notre *EM* est équipé d'un *buffer* [4] pour retenir les événements si nécessaire, comme illustré figure 4.2b. Il s'agit d'un schéma global, valable aussi dans le cas de propriétés non temporisées. A tout moment, l'*EM* a la possibilité de stocker un événement dans le buffer (opération *Store*), ou de relâcher le premier événement stocké dans le buffer (opération *Dump*). Contrairement à [98, 150], les dates auxquelles ces opérations ont lieu doivent être calculées par l'*EM*. Pour des besoins de modélisation, une opération *Elapse* d'écoulement du temps a été définie. L'objectif initial de nos travaux était de définir un modèle formel pour effectuer la transformation entre l'entrée et la sortie, et permettant de faire évoluer le buffer, à la manière de [98] qui utilise une machine de Mealy dans ce but. Toutefois, il s'est avéré impossible de se baser sur un modèle aussi simple, et pour cette raison, nous avons défini dans [C21] une version *opérationnelle*, à base de règles sur des configurations globales, puis une version *fonctionnelle* de l'*EM* dans [J6].

A l'instar de Ligatti et al. [150], nous avons choisi de modéliser les propriétés par des automates. Dans notre cas, il s'agit de TA déterministes. Nous utilisons donc des TA équipés de localités acceptantes pour décrire les propriétés à enforcer. Ces TA sont différents de ceux utilisés au chapitre 2, mais finalement plus proches de la définition d'Alur et Dill dans [6]. Nous rappelons quelques notations usuelles concernant les TA. Un lecteur habitué pourra aisément sauter cette partie. Soit $X = \{x_1, \ldots, x_k\}$ un ensemble fini d'*horloges*. Une *valuation d'horloge* pour X est une fonction ν de X vers $\mathbb{R}_{\geq 0}$. $\mathbb{R}_{\geq 0}^X$ représente les valuations d'horloges dans X. Pour $\nu \in \mathbb{R}_{\geq 0}^X$ et $\delta \in \mathbb{R}_{\geq 0}$, $\nu + \delta$ est la valuation qui affecte $\nu(x) + \delta$ à chaque horloge x de X. Etant donné un ensemble d'horloges $X' \subseteq X$, $\nu[X' \leftarrow 0]$ est la valuation d'horloge ν pour laquelle toutes les horloges de X' sont affectées à 0. $\mathcal{G}(X)$ représente l'ensemble des contraintes d'horloges défini comme une combinaison booléenne de contraintes simples de la

3. En réalité, le placement des contraintes physiques a fait l'objet de plusieurs changements. Elles sont intégrées à la transparence dans [C21], mises à part comme une contrainte supplémentaire dans [J6], et intégrées directement dans la définition de fonction d'enforcement dans [J7], ce qui nous semble être la place adéquate.

4. Le terme de buffer est utilisé ici en référence aux travaux précédents. Plus précisément, il est utilisé comme une FIFO.

forme $x \bowtie c$ avec $x \in X$, $c \in \mathbb{N}$ et $\bowtie \in \{<, \leq, =, \geq, >\}$. Etant donné $g \in \mathcal{G}(X)$ et $\nu \in \mathbb{R}^X_{\geq 0}$, on note $\nu \models g$ quand ν vérifie g.

Définition 3 (Automate Temporisé)**.** *Un* automate temporisé (TA) *est un 6-uplet* $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, G)$*, t.q.* L *est un ensemble fini de* localités *avec* $l_0 \in L$ *la* localité initiale*,* X *un ensemble fini d'horloges,* Σ *un ensemble fini d'*événements*,* $\Delta \subseteq L \times \mathcal{G}(X) \times \Sigma \times 2^X \times L$ *la* relation de transition*, et* $G \subseteq L$ *l'ensemble des localités acceptantes.*

La sémantique *d'un TA est un système de transitions temporisé* $[\![\mathcal{A}]\!] = (Q, q_0, \Gamma, \rightarrow, F_G)$ *où* $Q = L \times \mathbb{R}^X_{\geq 0}$ *est l'ensemble (infini) d'*états*,* $q_0 = (l_0, \nu_0)$ *l'état initial où* ν_0 *est la valuation qui affecte chaque horloge à* 0*,* $F_G = G \times \mathbb{R}^X_{\geq 0}$ *l'ensemble des états acceptants,* $\Gamma = \mathbb{R}_{\geq 0} \times \Sigma$ *est l'ensemble des* labels *de transition, i.e., des paires composées d'un délai et d'un événement. La relation de transition* $\rightarrow \subseteq Q \times \Gamma \times Q$ *est un ensemble de transitions de la forme* $(l, \nu) \xrightarrow{(\delta, a)} (l', \nu')$ *avec* $\nu' = (\nu + \delta)[Y \leftarrow 0]$ *s'il existe* $(l, g, a, Y, l') \in \Delta$ *t.q.* $\nu + \delta \models g$ *pour* $\delta \geq 0$*.*

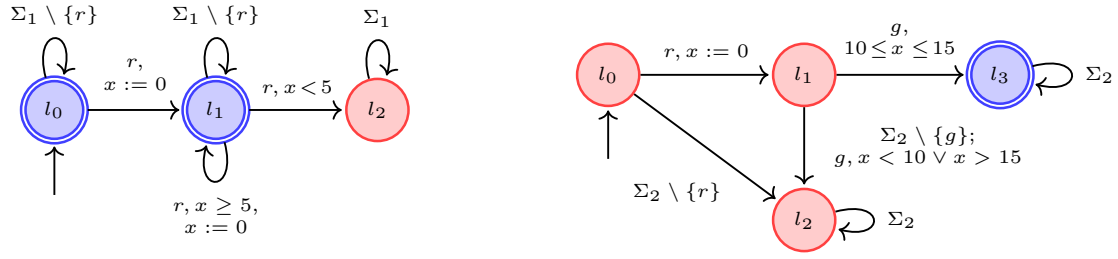
Par la suite, nous n'utiliserons que des TA complets et déterministes au sens usuel (voir [6]). Une propriété est définie par un langage temporisé $\varphi \subseteq \mathrm{tw}(\Sigma)$. Etant donné un mot temporisé $\sigma \in \mathrm{tw}(\Sigma)$, on considère que σ satisfait φ (noté $\sigma \models \varphi$) si $\sigma \in \varphi$. Dans nos travaux, nous nous sommes focalisés sur des propriétés de safety, puis de co-safety (au sens de la hiérarchie SP), et enfin sur des propriétés quelconques, toutes décrites par des TA. On considérera qu'une propriété est *régulière* si elle peut être définie par un langage accepté par un TA.

Définition 4 (TA de Safety et Co-safety)**.** *Un TA déterministe et complet* $(L, l_0, X, \Sigma, \Delta, G)$*, où* $G \subseteq L$ *est l'ensemble des localités acceptantes, est :*

- *un TA de* safety *si* $\nexists (l, g, a, Y, l') \in \Delta, l \in L \setminus G \wedge l' \in G$ *;*
- *un TA de* co-safety *si* $\nexists (l, g, a, Y, l') \in \Delta, l \in G \wedge l' \in L \setminus G$*.*

Cette définition traduit le fait que si on atteint une localité non-acceptante (resp. acceptante) dans un TA de safety (resp. de co-safety), alors on ne peut jamais revenir dans une location acceptante (resp. non acceptante).

Exemple 12. *Les figures 4.3a et 4.3b montrent deux exemples de propriétés temporisées. Les états acceptants sont représentés en bleu et doublement cerclés. La figure 4.3a est une propriété de safety sur* $\Sigma_1 = \{a, r\}$ *qui signifie "*Il doit y avoir un délai d'au moins 5 unités de temps entre deux requêtes*" ; la figure 4.3b est une propriété de co-safety sur* $\Sigma_2 = \{a, g, r\}$ *qui signifie "*L'utilisateur ne peut faire une action a qu'après une authentification réussie. Cette dernière doit survenir entre 10 et 15 unités de temps*".*



(a) Un TA de safety $\mathcal{A}_{\varphi_1}$ (propriété φ_1)

(b) Un TA de co-safety $\mathcal{A}_{\varphi_2}$ (propriété φ_2)

Figure 4.3 – Exemples de propriétés temporisées

4.2.2 Enforcement de propriétés de safety

Dans cette partie, nous considérons $E : \mathrm{tw}(\Sigma) \times \mathbb{R}_{\geq 0} \to \mathrm{tw}(\Sigma)$ une fonction d'enforcement et une propriété φ spécifiée par un automate $\mathcal{A} = (L, l_0, X, \Sigma, \Delta, G)$ et sa sémantique associée $[\![\mathcal{A}]\!] = (Q, q_0, \Gamma, \to, F_G)$. Nous avons été amenés à définir dans [C21, J6] les notions usuelles de correction et transparence. Ces définitions s'inspirent de [98], mais contrairement à ce dernier, l'optimalité est définie séparément de la transparence. L'idée derrière la correction est que si un mot est relâché par E, alors la sortie doit à un moment donné (futur) satisfaire la propriété φ. La transparence permet de restreindre l'ensemble des possibilités de l'*EM*. Dans notre cas, nous considérons que la sortie de E au temps t est un préfixe retardé de l'observation de l'entrée au même moment. Notons qu'il est assez simple d'adapter ces notions pour d'autres primitives, comme proposé dans [185, 100]. Ces définitions, que nous avons utilisées dans [J6], sont assez générales pour être valables avec n'importe quelle propriété régulière φ et ne se limitent pas aux propriétés de safety [5]. Ca n'est pas le cas de la définition d'optimalité [6]. On considérera qu'une fonction d'enforcement E est optimale par rapport à une propriété de safety φ si à tout instant t, la sortie de E est le plus long mot temporisé correct qui retarde la séquence d'entrée (observée en t), ce qui étend naturellement la version non temporisée de [96], et que chaque préfixe de E au temps t possède le plus petit dernier délai possible. Nous obtenons donc les définitions suivantes :

Définition 5 (Correction)**.** *Une fonction d'enforcement E définie sur un alphabet Σ est* correcte *par rapport à φ si :* $\forall \sigma \in \mathrm{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0} : \; E(\sigma, t) \neq \epsilon \implies (\exists t' \geq t : E(\sigma, t') \models \varphi)$.

Définition 6 (Transparence)**.** *Une fonction d'enforcement E définie sur un alphabet Σ est* transparente *si :* $\forall \sigma \in \mathrm{tw}(\Sigma), \forall t \in \mathbb{R}_{\geq 0} : E(\sigma, t) \preccurlyeq_{\mathrm{d}} \mathrm{obs}(\sigma, t)$

Définition 7 (Optimalité (liée à la safety))**.** *Une fonction d'enforcement correcte et transparente E définie sur un alphabet Σ est* optimale *par rapport à une propriété de safety φ si les deux contraintes suivantes sont vérifiées :*

— $\nexists \omega', \omega' \models \varphi \;\wedge\; \omega' \preccurlyeq_d obs(\sigma, t) \;\wedge\; |\omega'| > |E(\sigma, t)|$

— $\forall i \in [1, |E(\sigma,t)|], \neg\nexists \delta'' \in \mathbb{R}_{\geq 0}, \quad \mathrm{del}(\mathrm{obs}(\sigma,t)(i)) \leq \delta'' \leq \mathrm{del}(E(\sigma,t)(i)) \wedge E(\sigma,t)_{[\cdots i-1]} \cdot (\delta'', \mathrm{act}(E(\sigma,t)(i))) \models \varphi$

Avant de décrire formellement l'*EM*, nous allons donner l'intuition de son comportement attendu sur un exemple.

Exemple 13. *Supposons que nous souhaitons enforcer la propriété de safety φ_1 décrite par l'automate $\mathcal{A}_{\varphi_1}$ de la figure 4.3a, et que cet EM s'applique à la séquence d'entrée $\sigma = (1, a) \cdot (3, r) \cdot (1, r)$. A $t = 1$, l'action a est reçue. Sur $\mathcal{A}_{\varphi_1}$, l'application de cet événement ramène le TA en l_0, qui est un état acceptant, donc la séquence ne doit pas être modifiée. A $t = 4$, l'action r est reçue. Sur l'automate, l'application de cet événement amène en l_1, qui est toujours un état acceptant, donc la séquence ne doit toujours pas être modifiée. Enfin, en $t = 5$, la réception de l'action r devrait amener $\mathcal{A}_{\varphi_1}$ en l_2 qui cette fois est un état non-acceptant, donc cette action*

5. Les définitions de correction et transparence dans [C21] sont en revanche explicitement liées aux propriétés de safety et de co-safety.

6. En réalité, la définition d'optimalité donnée dans [J6] est assez générale aussi, mais un peu compliquée, nous avons donc préféré donner celle concernant les propriétés de safety uniquement.

doit être retardée jusqu'à ce que la valeur de l'horloge x de $\mathcal{A}_{\varphi_1}$ vaille 5, soit ici 4 unités de temps. Ainsi, $\mathcal{A}_{\varphi_1}$ revient en l_1 qui est bien un état acceptant. La sortie qu'un EM devrait nous fournir ici est donc $(1, a) \cdot (3, r) \cdot (5, r)$

Nous fournissons la sémantique formelle de notre *EM* qui réalise la fonction d'enforcement pour des propriétés de safety illustrée à l'exemple 13. Nous avons choisi de donner ici uniquement la description sous forme opérationnelle, comme décrit dans [C21], car elle nous semble plus intuitive, et elle permet une implémentation plus aisée. Nous discuterons ci-dessous une description fonctionnelle généralisée à tout type de propriété régulière, présentée dans [J6]. Cette forme opérationnelle est un système de transitions avec des configurations de la forme $(\sigma_{buf}, s, d, b, q)$ où σ_{buf} est l'état courant du buffer (sous forme de mot temporisé), s (resp. d) une valeur d'horloge symbolisant le temps écoulé depuis le dernier ajout (resp. retrait) d'un événement dans le buffer, b un booléen vrai tant que la propriété de safety est toujours satisfaite, et q l'état courant de $[\![\mathcal{A}]\!]$ ($\mathcal{A}$ est un automate de safety ici). Trois règles s'appliquent, par ordre de priorité :

— *Store :* lorsque l'*EM* reçoit un événement (δ, a), il stocke immédiatement dans son buffer l'événement (δ', a) où δ' est le délai minimal d'attente pour la propriété reste satisfaite, possiblement infini (dans ce cas, b devient faux). s est remis à zéro.

— *Dump* : le délai du premier événement dans le buffer correspond au temps qu'il faut attendre depuis le dernier *dump*. Ainsi, la règle *dump* est appliquée quand d est égal à ce temps. Dans ce cas, d est remis à zéro, et le premier événement dans le buffer est relâché.

— *Elapse* : le temps s'écoule et les valeurs de s et d sont incrémentées.

La sémantique formelle est fournie ci-dessous. Elle utilise une fonction $\mathrm{update_s}$ (définie formellement dans [C21]) qui calcule le délai minimal (supérieur au délai du premier événement en attente) permettant au TA de rester dans un état acceptant après avoir effectué une action a. Cela revient à calculer combien de temps a doit être maintenu dans le buffer.

Définition 8 (Mécanisme d'Enforcement pour safety)**.** *Un EM pour une propriété de safety φ est un système de transitions $\mathcal{E} = (C, C_0, \Gamma_{\mathcal{E}}, \hookrightarrow)$ t.q. :*

— $C = \mathrm{tw}(\Sigma) \times \mathbb{R}_{\geq 0} \times \mathbb{R}_{\geq 0} \times \mathbb{B} \times Q$ *est l'ensemble des configurations*

— *la configuration initiale est* $C_0 = (\epsilon, 0, 0, \mathtt{tt}, q_0) \in C$ *;*

— $\Gamma_{\mathcal{E}} = \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big) \times Op \times \big((\mathbb{R}_{\geq 0} \times \Sigma) \cup \{\epsilon\}\big)$ *est l'alphabet d'*entrée-opération-sortie *(noté* entrée/opération/sortie*), où* $Op = \{\mathrm{store}(\cdot), \mathrm{dump}(\cdot), \mathrm{elapse}(\cdot)\}$ *;*

— $\hookrightarrow \subseteq C \times \Gamma_{\mathcal{E}} \times C$ *est la relation de transition définie comme la plus petite relation obtenue en suivant les règles suivantes appliquées dans l'ordre :*

  — *store :* $(\sigma_{buf}, \delta, d, \mathtt{tt}, q) \overset{(\delta,a)/\mathrm{store}(\delta',a)/\epsilon}{\hookrightarrow} (\sigma_{buf} \cdot (\delta', a), 0, d, (\delta' \neq \infty), q')$ *avec :*

    — $\delta' = \mathrm{update_s}(q, a, \delta)$,

    — q' *est défini comme* $q \overset{(\delta',a)}{\to} q'$ *si* $\delta' < \infty$ *et* $q' = q$ *sinon ;*

  — *dump :* $\big((\delta, a) \cdot \sigma_{buf}, s, \delta, b, q\big) \overset{\epsilon/\mathrm{dump}(\delta,a)/(\delta,a)}{\hookrightarrow} (\sigma_{buf}, s, 0, b, q)$ *if* $\delta \neq \infty$ *;*

  — *elapse :* $(\sigma_{buf}, s, d, b, q) \overset{\epsilon/\mathrm{elapse}(\delta)/\epsilon}{\hookrightarrow} (\sigma_{buf}, s + \delta, d + \delta, b, q)$.

Toutes les informations sont disponibles dans ce système de transition pour extraire les

éléments nécessaires à la définition d'une fonction d'enforcement E associée à $\mathcal{E}$. Pour cela, on définit la valeur la fonction E comme étant la sortie de $\mathcal{E}$ à tout moment [7].

Propriétés Une fonction d'enforcement E ainsi définie pour une propriété de safety φ est *correcte*, *transparente*, et *optimale* au sens des définitions 5, 6 et 7. Un lecteur intéressé pourra trouver les preuves dans [C21].

Exemple 14. *La figure 4.4 présente un exemple d'évolution de l'EM de safety enforçant la propriété φ_1 modélisée par le TA $\mathcal{A}_{\varphi_1}$ de la figure 4.3a, avec la séquence d'entrée $\sigma = (1,a)\cdot(3,r)\cdot(1,r)$. La colonne de gauche correspond au temps global. La séquence en rouge (resp. vert, bleu) correspond à la sortie (resp. l'entrée, le buffer) de l'EM. Le buffer (en bleu) est le premier élément du quintuplet correspondant à la configuration actuelle de l'EM (en bleu et noir).*

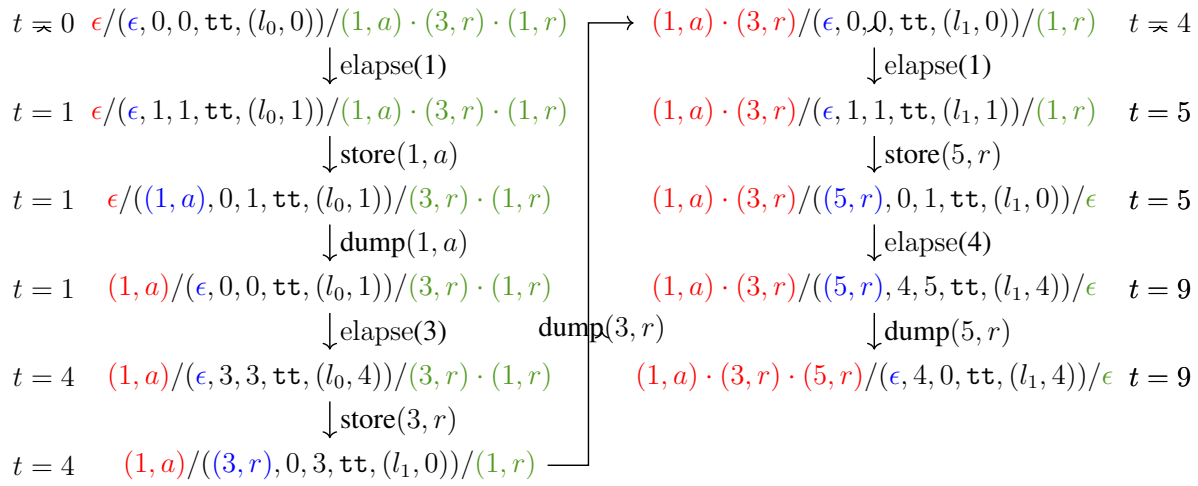


Figure 4.4 – Evolution de l'*EM* de safety avec la séquence d'entrée $(1,a)\cdot(3,r)\cdot(1,r)$ pour φ_1

L'écriture sous forme de système de transitions permet de déduire des algorithmes simples pour implémenter l'*EM*. Ces derniers ont été décrits dans [C21]. L'implémentation est composée d'une file de mots temporisés représentant le buffer décrit précédemment, et de deux processus concurrents, `DumpProcess` et `StoreProcess`. Le premier, détaillé dans l'algorithme 1, gère le premier événement (δ, a) dans la file (s'il existe), et relâche a lorsque le temps depuis le dernier `dump` atteint δ. Le second, décrit dans l'algorithme 2, regarde à la réception d'un nouvel événement (δ, a) si l'état atteint dans le TA par cet événement est acceptant. Dans le cas contraire, une nouvelle valeur $\delta' > \delta$ assurant que le TA va atteindre un état acceptant de façon optimale est calculée (si elle existe), et l'événement (δ', a) est mis dans la file. Ce processus est aussi chargé de mettre à jour l'état symbolique courant du TA.

7. Plus précisément, il s'agit de la concaténation des projections sur le 3è élément de la configuration au moment observé.

Algorithm 1 DumpProcess

$d \leftarrow 0$
while `tt` **do**
  await $(|\sigma_{buf}| \geq 1)$
  $(\delta, a) \leftarrow$ dequeue (σ_{buf})
  wait $(\delta - d)$
  dump (a)
  $d \leftarrow 0$
end while

Algorithm 2 StoreProcess

$(l, X) \leftarrow (l_0, [X \leftarrow 0])$
while `tt` **do**
  $(\delta, a) \leftarrow$ await *event*
  if $(\text{post}(l, X, a, \delta) \notin G)$ **then**
    $\delta' \leftarrow \text{update}(l, X, a, \delta)$
    if $\delta' = \infty$ **then**
      terminate StoreProcess
    end if
  else
    $\delta' \leftarrow \delta$
  end if
  $(l, X) \leftarrow \text{post}(l, a, X, \delta')$
  enqueue (δ', a)
end while

Ces algorithmes ont été implémentés dans un prototype. Les calculs de temps δ sur les TA ont été délégués à la bibliothèque *Uppaal* [20, 144], qui permet toute sorte d'opérations sur les TA de façon efficace. Le lecteur pourra trouver une synthèse des résultats expérimentaux dans [C21]. Ces derniers montrent la viabilité de l'approche. Ce prototype a subi depuis des améliorations et des évolutions effectuées par l'équipe Vertecs d'Inria Rennes Bretagne Atlantique, notamment pour implémenter des propriétés régulières, pour finalement devenir *TiPEX* [186].

4.2.3 Enforcement de propriétés plus complexes

Le framework d'enforcement décrit ci-dessus n'est valable que pour des propriétés de safety. Prenons le cas d'une propriété de co-safety. Par définition, elle consiste à rester dans un état non-acceptant jusqu'à ce que le système passe dans un état acceptant et y reste définitivement. D'un point de vue enforcement, ça implique que l'*EM* ne doit pas relâcher le moindre événement sans être certain que le système va atteindre un état acceptant. En pratique, l'*EM* va adopter une stratégie de type "store and forward" [8], ce qui peut provoquer des délais importants entre la réception d'un événement par l'*EM* et son émission effective par celui-ci, pouvant aller jusqu'à la durée totale de la séquence, alors que la séquence initiale entrée est parfaitement correcte. Cette stratégie implique aussi que le fait de ne rien sortir (la sortie ϵ) doit être considéré comme correct. Les définitions de correction (définition 5) et de transparence (définition 6) restent valables pour une propriété de co-safety. Notons que la définition de correction reste valable car nous avons ajouté le cas particulier de la séquence vide ϵ, et considéré qu'il suffit qu'une propriété soit vérifiée dans le futur pour que l'enforcement soit correct [9]. Par contre, la définition d'optimalité (définition 7) ne convient pas, car il faut prendre en compte la durée totale de la séquence. Nous avons proposé dans [C21] une définition d'optimalité pour la co-safety qui est calculée sur le temps total de la séquence et correspond à la somme minimale des

8. Cette expression est héritée du vocabulaire utilisé pour certains médias réseau, par exemple un switch "store and forward" garde un trame reçue en mémoire, et commence à l'émettre seulement une fois qu'il l'a reçue intégralement.

9. Dans [C21], nous avions séparé les définitions de correction pour les safety et les co-safety.

délais du plus petit préfixe qui satisfait la propriété. Pour le reste de la séquence (i.e. une fois que la propriété est satisfaite), la sortie doit être égale à l'entrée.

Exemple 15. *Prenons l'exemple d'un EM cherchant à enforcer la propriété de co-safety φ_2 décrite par le TA $\mathcal{A}_{\varphi_2}$ de la figure 4.3b avec la séquence d'entrée $(1, r) \cdot (8, g) \cdot (5, a)$. A $t = 1$, l'action r est reçue. Elle ne peut être émise car $\mathcal{A}_{\varphi_2}$ atteindrait l_1 qui est un état non-acceptant. A $t = 9$, l'action g est reçue, et ne peut être émise non plus, car menant en l_2, un état non-acceptant. Toutefois, si on la retarde de 2 unités de temps, alors, le TA passe en l_3 et la propriété devient nécessairement vérifiée. A ce moment, l'EM peut commencer à émettre la séquence, mais compte tenu du décalage qui a été nécessaire pour s'assurer que la propriété était bien vérifiée, l'action r sera sortie au temps $t = 9$, et l'action g au temps $t = 19$. Le reste de la séquence est ensuite inchangé, donc l'action a sortira à $t = 24$. La sortie qu'un EM devrait nous fournir ici est donc $(9, r) \cdot (10, g) \cdot (5, a)$.*

Nous avons proposé dans [C21] un mécanisme d'enforcement sous forme *opérationnelle* basé sur cinq règles (la règle *Store* vue précédemment est remplacée par trois règles). Lorsque un événement (δ, a) survient, et que φ reste insatisfaite en considérant cet événement sur le TA, y compris en le retardant, alors (δ, a) est placé dans le buffer et les variables mises à jour (règle $\text{store-}\overline{\varphi}$). En revanche, si φ est satisfaite (possiblement en retardant l'action), alors la règle $\text{store-}\varphi_{\text{init}}$ s'applique. Les délais du buffer sont remplacés par les meilleures valeurs possibles. Cette règle est appliquée une seule fois. A partir de ce moment, φ sera nécessairement satisfaite, et il suffit simplement de placer tout événement (δ, a) à la suite du buffer (règle $\text{store-}\varphi$). Les autres règles décrites précédemment sont inchangées. Notons que l'état courant de l'automate n'est pas nécessaire dans les configurations ici, car on reste en réalité à l'état initial tant que la propriété φ n'est pas vérifiée ; ensuite, une fois qu'elle le devient, aucun calcul n'est nécessaire. Rappelons que cet *EM* n'est défini que pour des propriétés de co-safety. Le lecteur intéressé trouvera les algorithmes implémentant une variante de ce système de transitions dans [C21] ainsi que les résultats expérimentaux.

Remarque 1. *Un lecteur avisé notera que l'exemple d'évolution de l'EM de cosafety de [C21] ne donne pas exactement les mêmes résultats que l'exemple 16 de ce manuscrit. En effet, dans [C21], le délai initial (correspondant à l'attente que la propriété soit vérifiée) n'était pas intégré explicitement à la séquence placée dans le buffer (mais bien pris en compte), et nous ajoutions un délai supplémentaire correspondant au temps écoulé entre la validation de la propriété, et le temps d'occurrence du premier événement. Il nous est apparu par la suite que ce délai supplémentaire était superflu. Les choix effectués dans cette partie s'inspirent en réalité de notre article [J6] pour assurer une cohérence avec le reste du chapitre.*

Exemple 16. *Un exemple simplifié de l'état de l'EM de co-safety (entrées, sorties, état du buffer et règles appliquées) pour la propriété φ_2 décrite par le TA $\mathcal{A}_{\varphi_2}$ de la figure 4.3b avec la séquence d'entrée $(1, r) \cdot (8, g) \cdot (5, a)$ est donné figure 4.5. La séquence rouge (resp. bleue, verte) représente la sortie (resp. le buffer, l'entrée) de l'EM.*

Dans [J6] nous avons généralisé les résultats précédents à toute propriété temporisée régulière, i.e. décrite par un TA. Un exemple illustrant le comportement attendu de l'*EM* est décrit à l'exemple 17. Notre approche reprend l'idée de Ligatti et al. [149, 150] d'utiliser le buffer pour stocker le suffixe des exécutions invalides en attendant qu'elles deviennent à nouveau valides, mais adaptée dans un cadre temporisé. Nous avons proposé une définition généralisée de

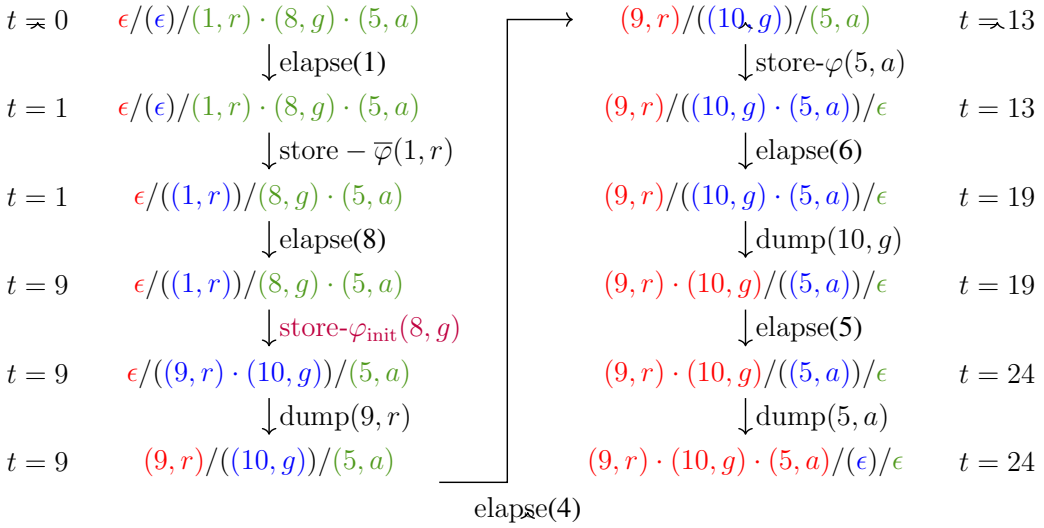


Figure 4.5 – Evolution de l'*EM* de co-safety avec la séquence d'entrée $(1,r)\cdot(8,g)\cdot(5,a)$ pour φ_2

la correction, de la transparence, et de l'optimalité. En réalité, par rapport à ce manuscrit, seule la définition d'optimalité change, car les définitions 5 et 6 sont valables pour toute propriété régulière. Pour adapter la définition d'optimalité, il a été nécessaire de considérer la taille du mot mais aussi le temps global. Le lecteur trouvera la définition détaillée dans [J6].

Exemple 17. *Considérons un EM souhaitant enforcer la propriété φ_3 décrite par le TA $\mathcal{A}_{\varphi_3}$ figure 4.6, avec l'entrée $\sigma = (3,g)\cdot(10,r)\cdot(3,g)\cdot(5,g)$. A $t = 3$, l'action g est reçue. Sur $\mathcal{A}_{\varphi_3}$, elle mènerait en l_1 qui est un état non-acceptant, donc elle ne peut être émise. En $t = 13$, l'EM reçoit l'action r. Cette dernière doit être retardée de 5 unités de temps pour éviter d'aller en l_2, ainsi, on atteint bien l_0 qui est un état acceptant. Il est donc possible de commencer à émettre la séquence en sortie, i.e. l'action g. A $t = 16$, l'action g est reçue. Comme précédemment, elle ne peut être émise. L'action g qui survient à $t = 21$ ne peut pas non plus être émise, mais en plus, comme elle mènerait à l_2 dans tous les cas, y compris en étant retardée, l'EM va bloquer toutes les actions futures. Finalement, en $t = 28$, l'action r qui était en attente d'émission va être émise. La sortie qu'un EM devrait nous fournir ici est donc $(13,g)\cdot(15,r)$.*

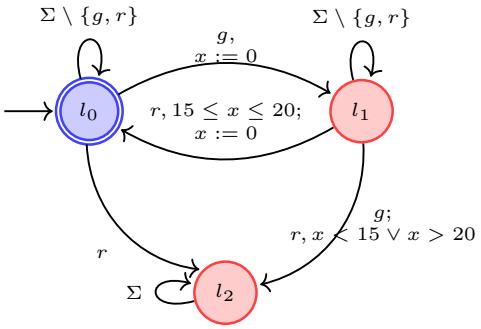


Figure 4.6 – Un TA quelconque $\mathcal{A}_{\varphi_3}$ (propriété φ_3 régulière).

Nous avons aussi redéfini une description de l'*EM* à deux niveaux d'abstraction permettant d'enforcer toute propriété temporisée régulière : une version *fonctionnelle* basée sur la fonction

store, et une autre *opérationnelle* (sous forme de système de transitions), plus proche de l'implémentation, toutes deux équivalentes. Le principe est le suivant : à la réception d'un événement, l'*EM* va calculer à l'aide de sa séquence en mémoire complétée par ce nouvel événement s'il existe une séquence retardante (i.e. même taille, délais augmentés) permettant d'atteindre un état acceptant. Si oui (cas 1), cette dernière est prête pour la sortie, i.e. chaque élément sera relâché par le buffer à la date correspondante. Dans le cas contraire (cas 2), l'événement est stocké dans le buffer, et donc bloqué en attendant d'autre événements. Ce principe peut se décrire sous forme fonctionnelle. Nous avons défini dans [J6] une fonction inductive store qui pour une séquence d'entrée donnée, va retourner deux éléments : σ_s correspondant aux événements prêts à être sortis (quand la date sera atteinte), et σ_{buf} correspondant aux événements stockés ne permettant pas (encore) d'atteindre un état acceptant. Lorsqu'un nouvel événement survient, σ_s et σ_{buf} sont mis à jour selon le principe décrit ci-dessus (cas 1 et 2). La sortie de l'*EM* se trouve finalement dans σ_s à la date correspondante. La version opérationnelle sous forme de système de transitions reprend aussi le même principe, appliqué aux mêmes types de configurations que l'*EM* pour la safety, à ceci près qu'elle comporte maintenant explicitement les séquences σ_s et σ_{buf} dans les configurations. En cas de nouvel événement (δ, a), dans le cas 1, la règle ***store-***φ s'applique et la nouvelle séquence retardante est placée à la suite de σ_s (i.e. elle est prête à être sortie). Dans le cas 2, la règle ***store-***$\overline{\varphi}$ s'applique, ajoutant (δ, a) à la fin de σ_{buf}, i.e. retenu jusqu'à nouvel ordre. Les anciennes règles *dump* et *elapse* restent inchangées. Le lecteur souhaitant obtenir plus de détails pourra trouver la description complète des deux représentations de l'*EM* dans [J6].

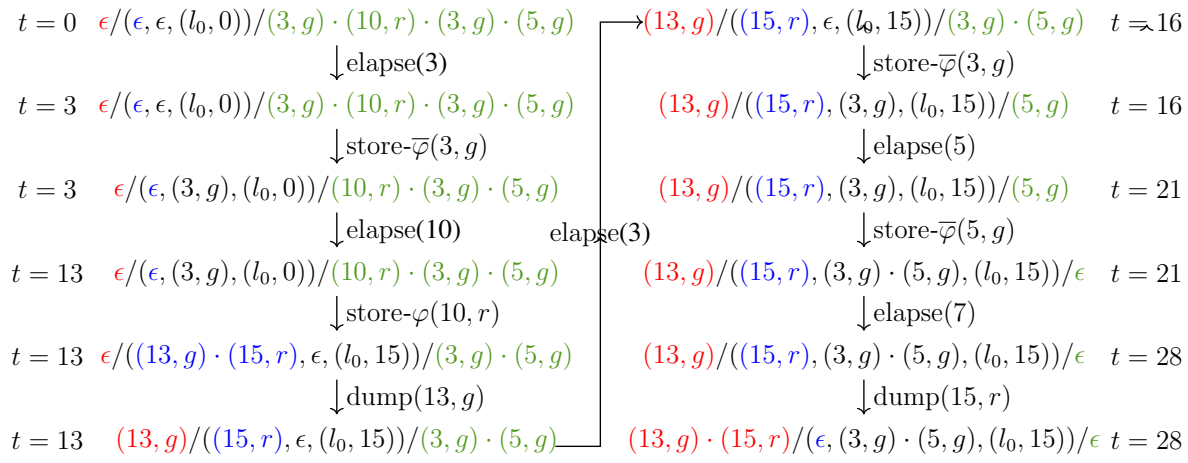


Figure 4.7 – Evolution de l'*EM* avec la séquence d'entrée $\sigma = (3, g) \cdot (10, r) \cdot (3, g) \cdot (5, g)$ pour φ_3

Exemple 18. *La figure 4.7 présente un exemple d'évolution de l'état de l'EM et les règles appliquées pour enforcer la propriété φ_3 modélisée par le TA $\mathcal{A}_{\varphi_3}$ de la figure 4.6, avec la séquence d'entrée $\sigma = (3, g) \cdot (10, r) \cdot (3, g) \cdot (5, g)$. Le mot temporisé en rouge (resp. bleu, noir, vert) représente la sortie (resp. σ_s, σ_{buf}, l'entrée) de l'EM. L'état courant du TA est aussi indiqué en noir.*

Propriétés Une fonction d'enforcement E ainsi définie pour une propriété régulière quelconque φ est *correcte*, *transparente*, et *optimale*. Un lecteur intéressé pourra consulter les

preuves dans [J6]. Notons qu'il est parfois difficile de modéliser une propriété complexe sous forme de TA. Utiliser un système d'automates peut donc faciliter cette modélisation. Nous avons discuté ce point dans [J6] et montré qu'en utilisant des définitions d'intersection et d'union de TA basées sur la notion de produit, les propriétés modélisées par un TA de safety et de co-safety sont closes par union (finie) et par intersection, que la négation [10] d'une propriété modélisée par un TA de safety est une co-safety (et vice-versa), et que les propriétés régulières sont closes par opération booléenne.

4.3 Enforcement à l'exécution de propriétés (temporisées) en présence d'événements incontrôlables

Dans certains cas, il peut être souhaitable que des événements particuliers soient observés par l'*EM*, afin de réagir en conséquence, mais ne puissent être modifiés par celui-ci. Il peut s'agir par exemple d'un message d'alerte en diffusion large, ou encore d'une sorte d'interruption (au sens informatique du terme, par exemple dans les processeurs) dans un système. On parle alors d'*événement incontrôlable*. Cela peut s'avérer utile pour modéliser des événements liés à l'environnement physique d'un système par exemple, et permettre à l'*EM* de réagir en conséquence. Par exemple, dans un système de contrôle d'une voiture, il serait raisonnable de considérer le message "franchissement de ligne continue" comme un événement incontrôlable, i.e. observable mais non modifiable par l'*EM*, mais permettant à ce dernier d'opérer la correction (de trajectoire) nécessaire. D'un point de vue formel, cela revient à considérer deux ensembles d'actions, les unes étant *contrôlables*, les autres *incontrôlables*.

A notre connaissance, il existe assez peu de travaux concernant l'*EE* en présence d'événements incontrôlables, si l'on exclut les travaux concernant la théorie de la supervision et du contrôle introduite par Ramadge et Wonham [189]. Dans cette dernière, les événements sont aussi partitionnés en événement contrôlables et incontrôlables, mais le principe consiste généralement à utiliser un modèle global, une sorte de produit synchronisé entre la spécification complète du système et les états à éviter, pour empêcher le système initial d'atteindre ces états. Même si l'objectif peut paraître assez proche de celui de l'*EE*, ce dernier va se baser sur un propriété spécifique (généralement abstraite) et non sur la spécification globale du *SUS*, et le panel de primitives de l'*EE* est plus riche. Concernant l'*EE*, on peut citer à nouveau les travaux de Basin et al. [17, 16] qui proposent un framework d'enforcement en présence d'événements incontrôlables. Sur le même principe que Schneider [200], l'*EM* va simplement stopper l'exécution du système en cas de violation de la propriété. Cette approche ne permet d'enforcer que des propriétés de safety. Basin et al. proposent d'adapter ce framework dans le cas de systèmes temporisés discrets, en considérant les *ticks* d'horloge comme des événements incontrôlables.

Dans cette section, nous nous intéressons à l'*EE* en présence d'événements incontrôlables. Comparé à Basin et al. [17, 16], nous proposons un *EM* pour tout type de propriété régulière modélisée à l'aide d'un TA déterministe. Ainsi, nous utilisons le pouvoir expressif des TA pour modéliser des propriétés plus riches, avec temps dense. Nous avons cherché à adapter nos travaux précédents sur l'*EE* [C21, J6] dans le cas d'événements incontrôlables. Nous reprenons donc le principe décrit figure 4.2b où l'*EM* est équipé d'un buffer. Nous considérerons un événement incontrôlable comme un événement devant être relâché instantanément par l'*EM*. Ce

10. Par négation, on veut dire ici que les états acceptants deviennent non-acceptants et réciproquement.

fait implique en réalité de nombreux changements non triviaux. Tout d'abord, si des événements (contrôlables) sont en attente dans le buffer lorsqu'un événement incontrôlable survient, alors ils seront finalement relâchés après ce dernier, ce qui signifie que la transparence définie précédemment (définition 6) selon l'idée que la sortie doit être un préfixe retardé de l'entrée, doit être adaptée. Nous avons donc défini une contrainte plus faible, la *transparence partielle*, que l'on nommera par le suite *p-transparence* [11]. De plus, les dates de ces événements en attente dans le buffer peuvent dépendre de l'événement nouvellement reçu, et doivent être recalculées à chaque occurrence d'événement incontrôlable. Il est aussi nécessaire d'anticiper l'occurrence d'un événement incontrôlable à tout moment, i.e. de calculer tous les (mauvais) états accessibles par toute séquence d'événements incontrôlables, ce qui peut s'avérer coûteux. Enfin, avec les définitions précédentes liées à l'enforcement, on constate qu'il existe des situations où il n'est pas possible de garantir la correction d'une propriété. C'est par exemple le cas s'il existe une séquence d'événements incontrôlables partant de l'état initial et menant directement à un état non-acceptant. Ainsi, il est nécessaire d'adapter le framework.

Nous avons proposé une première approche d'*EE* non optimal en présence d'événements incontrôlables dans [C23]. Nous y avons adapté les définitions précédentes, et décrit formellement un *EM* correct et p-transparent à deux niveaux d'abstraction, sur le même principe que [J6] : une version fonctionnelle (fonction store), et une version opérationnelle sous forme de système de transition. Nous avons ensuite adapté ce travail pour rendre l'*EM* optimal dans [J7]. Nous avons ensuite revisité complètement ce framework en utilisant la théorie des jeux. En effet, l'*EM* vérifie en permanence s'il peut atteindre un état acceptant de l'automate de la propriété, ce qui se modélise bien par un jeu de Büchi. Cela a permis de simplifier l'ensemble des notations, et d'améliorer le temps de calcul de l'*EM* en pré-calculant certaines décisions de celui-ci, permettant ainsi d'éviter l'exploration complète de l'arbre d'exécution à la volée. Ce travail a été publié dans [C24]. Enfin, nous avons développé l'outil *GREP* [192] qui implémente ces contributions. Une description de *GREP* et une comparaison avec *TiPEX* [186] a été proposée dans [C25].

Au final, nous avons formalisé un *EM* pour des propriétés non temporisées, puis temporisées, non optimal puis optimal, à chaque fois avec deux niveaux de descriptions équivalents : fonctionnel et opérationnel, en utilisant une approche similaire à celle de [J6], et une version équivalente à base de jeu de Büchi. Il serait bien trop long de détailler ici toutes ces approches, d'autant plus que certaines utilisent des notations assez indigestes. Aussi, nous avons décidé de ne détailler que l'enforcement de propriétés non temporisées, et ensuite de résumer les idées permettant d'étendre cette approche dans un cadre temporisé.

4.3.1 Enforcement à l'exécution de propriétés en présence d'actions incontrôlables

Nous décrivons dans cette partie nos contributions concernant l'*EE* de propriétés non temporisées en présence d'actions incontrôlables. Voici quelques notations nécessaires par la suite. On considère toujours Σ un alphabet fini d'actions, Σ^* l'ensemble des mots finis sur Σ, et un *langage* sur Σ comme un sous-ensemble quelconque de Σ^*. On rappelle qu'un mot w' est un *préfixe* d'un mot w (noté $w' \preccurlyeq w$), s'il existe un mot w'' tel que $w = w'.w''$. Le mot w'' est appelé le *résidu* de w après lecture du préfixe w', et noté $w'' = {w'}^{-1} \,.\, w$. Etant donné un mot $w \in \Sigma^*$

11. Dans nos papiers en anglais, nous l'avons appelée *compliance*.

et $\Sigma' \subseteq \Sigma$, on définit la *restriction* de w à Σ', noté $w_{|\Sigma'}$, le mot $w' \in \Sigma'^*$ dont les actions sont celles de w appartenant à Σ' dans le même ordre. On note $=_{\Sigma'}$ l'égalité des restrictions de deux mots à Σ' : pour σ et σ' dans Σ^*, $\sigma =_{\Sigma'} \sigma'$ si $\sigma_{|\Sigma'} = \sigma'_{|\Sigma'}$. De même, on définit $\preccurlyeq_{\Sigma'}$: $\sigma \preccurlyeq_{\Sigma'} \sigma'$ si $\sigma_{|\Sigma'} \preccurlyeq \sigma'_{|\Sigma'}$. De façon usuelle, un *automate* est un 5-uplet $(Q, q_0, \Sigma, \rightarrow, F)$, où Q est un ensemble fini d'*états*, $q_0 \in Q$ est l'état initial, Σ est l'alphabet, $\rightarrow \subseteq Q \times \Sigma \times Q$ est la relation de transition et $F \subseteq Q$ est l'ensemble des états acceptants. Par la suite, nous ne considérerons que des automates déterministes au sens habituel. Etant donné un automate $\mathcal{A} = (Q, q_0, \Sigma, \rightarrow, F)$ et un mot $\sigma \in \Sigma^*$, pour $q \in Q$, on note q after $\sigma = q'$, où q' est tel que $q \xrightarrow{\sigma} q'$, i.e. q' est l'état [12] atteint depuis q après lecture du mot σ. On définit aussi $\mathrm{Reach}(\sigma) = q_0$ after σ. Dans cette partie, nous considérons φ comme une propriété régulière définie par un automate $\mathcal{A}_\varphi = (Q, q_0, \Sigma, \rightarrow, F)$. Cette fois, l'alphabet Σ est partitionné en deux sous-ensembles $\Sigma_\mathrm{u} \subseteq \Sigma$ l'ensemble des actions *incontrôlables*, i.e. qui ne peuvent être modifiées par l'*EM*, et $\Sigma_\mathrm{c} = \Sigma \setminus \Sigma_\mathrm{u}$ les autres actions, dites *contrôlables*. Comme précédemment, l'*EM* est équipé d'un buffer, comme décrit figure 4.2b, pouvant retenir des actions (uniquement contrôlables), et les relâcher plus tard. Il est représenté par un mot (non temporisé) de Σ_c^*. Nous allons dans un premier temps décrire le

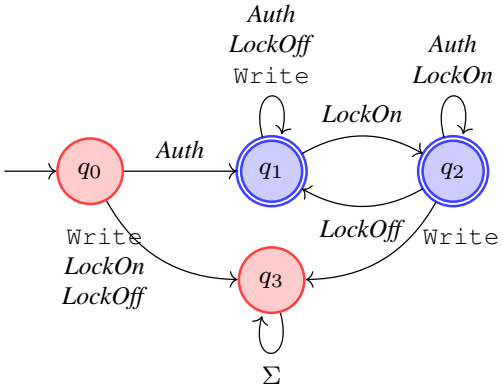


Figure 4.8 – Automate $\mathcal{A}_{\varphi_4}$ modélisant la propriété φ_4

comportement attendu de l'*EM* sur un exemple.

Exemple 19. *Considérons la propriété φ_4 modélisée par l'automate $\mathcal{A}_{\varphi_4}$ de la figure 4.8. Comme précédemment, les états acceptants sont représentés en bleu et doublement cerclés. Il s'agit d'une sorte de mémoire partagée simplifiée, où l'utilisateur, après s'être authentifié (Auth), peut écrire (`Write`) une valeur seulement si la mémoire est déverrouillée. Le fait de (dé)verrouiller la mémoire (LockOn, LockOff) est un choix de l'environnement, et est donc incontrôlable, de même pour l'authentification. Ici, nous avons $\Sigma_\mathrm{c} = \{$`Write`$\}$ et $\Sigma_\mathrm{u} = \{$LockOn, LockOff, Auth$\}$. Considérons le mot Auth.LockOn.`Write`.LockOff en entrée de l'EM. L'occurrence de l'action Auth mène $\mathcal{A}_{\varphi_4}$ en q_1 qui est un état acceptant, puis LockOn le mène en q_2, qui est aussi un état acceptant. Ces actions étant incontrôlables, il n'y a pas d'autre alternative que de les émettre immédiatement. L'occurrence de l'action contrôlable* `Write` *devrait mener $\mathcal{A}_{\varphi_4}$ en q_3 qui est un état non-acceptant, donc on ne doit pas l'émettre, i.e. il est placé dans le buffer. Lorsqu'elle survient, l'action incontrôlable LockOff doit être émise instantanément, faisant passer $\mathcal{A}_{\varphi_4}$ en q_1 et ainsi il est possible de sortir l'action* `Write` *qui ramène à nouveau en q_1. Finalement, la sortie attendue d'un EM devrait être Auth . LockOn . LockOff .* `Write` *qui respecte bien la propriété φ_4.*

12. Unique, car l'automate est déterministe.

Il a été nécessaire d'adapter les définitions de correction, transparence et d'optimalité, non seulement dans un cadre non temporisé, mais avec des actions incontrôlables. L'aspect non temporisé simplifie nettement les définitions qui se rapprochent de [150] et [98], mais l'ajout d'actions incontrôlables nécessite de traiter séparément les deux types d'actions à l'intérieur des définitions. Comme précédemment, une fonction d'enforcement décrit les comportements entrées-sorties de l'*EM*, et doit respecter des contraintes physiques. On retrouve la même idée que précédemment, i.e. elle ne peut "effacer" des actions déjà émises. Dans un cadre non temporisé, cela signifie simplement qu'elle est croissante sur Σ^* par rapport à la relation préfixe. Concernant la correction, il existe des situations où l'*EM* ne peut éviter d'atteindre un état non-acceptant depuis l'état initial. C'est le cas par exemple figure 4.8 où l'occurrence de l'action *LockOn* mène inévitablement à l'état non-acceptant q_3. Ainsi, nous avons considéré qu'un *EM* était correct si une fois qu'un état acceptant est atteint, alors la propriété devient vérifiée, ce qui revient à définir la correction sur un ensemble de mots clos par extension. La transparence doit aussi être adaptée, car comme nous l'avons vu dans l'exemple précédent, une action incontrôlable devant être relâchée instantanément, l'ordre des mots en sortie peut différer de l'entrée, ce qui est contraire à la notion habituelle de transparence [200, 19, C21, J6]. Nous l'avons donc affaiblie, en considérant que l'ordre des actions contrôlables ne doit pas être changé et que les actions incontrôlables doivent être émises instantanément (possiblement suivies par des actions contrôlables stockées dans le buffer). Nous l'avons appelée *p-transparence* (pour *transparence partielle*). Enfin, l'idée derrière l'optimalité est que s'il existe une fonction d'enforcement p-transparente, qui émet un mot plus long qu'un *EM* optimal, alors il existe nécessairement un séquence d'actions incontrôlables qui pourrait mener à un état non-acceptant, i.e. l'*EM* n'est pas correct. De plus, comme il n'est pas toujours possible de satisfaire une propriété depuis l'état initial, cette définition, comme la correction, est restreinte à un ensemble de mots clos par extension. Nous obtenons ainsi les définitions suivantes :

Définition 9 (Correction)**.** *Une fonction d'enforcement* $E : \Sigma^* \to \Sigma^*$ *est* correcte *par rapport à* φ *dans un ensemble extension-clos* $S \subseteq \Sigma^*$ *si* $\forall \sigma \in S, E(\sigma) \models \varphi$.

Définition 10 (p-transparence)**.** *Une fonction d'enforcement* $E : \Sigma^* \to \Sigma^*$ *est* p-transparente *par rapport à* Σ_u *et* Σ_c*, noté* $\text{p-transparent}(E, \Sigma_\mathrm{u}, \Sigma_\mathrm{c})$*, si* $\forall \sigma \in \Sigma^*, E(\sigma) \preccurlyeq_{\Sigma_\mathrm{c}} \sigma \wedge E(\sigma) =_{\Sigma_\mathrm{u}} \sigma \wedge \forall u \in \Sigma_\mathrm{u}, E(\sigma).u \preccurlyeq E(\sigma.u)$.

Définition 11 (Optimalité)**.** *Une fonction d'enforcement* $E : \Sigma^* \to \Sigma^*$ *p-transparente par rapport à* Σ_u *et* Σ_c*, et correcte dans un ensemble extension-clos* $S \subseteq \Sigma^*$ *est* optimale *dans* S *si :* $\forall E' : \Sigma^* \to \Sigma^*, \forall \sigma \in S, \forall a \in \Sigma, (\text{p-transparent}(E') \wedge E'(\sigma) = E(\sigma) \wedge |E'(\sigma.a)| > |E(\sigma.a)|) \implies (\exists \sigma_u \in \Sigma_\mathrm{u}^*, E'(\sigma.a.\sigma_u) \not\models \varphi)$.

Une problématique non triviale liée à ces travaux, c'est d'être capable de déterminer à tout moment à partir d'une séquence entrée dans l'*EM*, la "meilleure" séquence qui peut être à nouveau émise lorsqu'une nouvelle action survient. Pour cela, il est nécessaire de définir un ensemble Safe contenant l'ensemble des mots qu'il est possible de sortir à un moment donné tout en restant correct, puis d'identifier le "meilleur". Comme Safe dépend de ce qui a déjà été sorti, il prend comme paramètre l'état q atteint par cette séquence déjà sortie, et il prend aussi en paramètre l'état actuel du buffer de l'*EM*, i.e. un mot $w \in \Sigma_\mathrm{c}^*$. Ainsi, Safe va contenir tous les préfixes du buffer qui vérifient la correction, i.e. qui mènent à des états sûrs. Cependant, la notion d'état sûr n'est pas simple. Intuitivement, on considère comme état sûr un état acceptant, depuis lequel toutes les séquences d'événements incontrôlables mènent uniquement dans des

états acceptants. Ainsi, dans la figure 4.8, les états q_1 et q_2 sont sûrs, car la seule façon de sortir d'un de ces deux états acceptants, c'est d'émettre `Write`, qui est contrôlable. Ainsi, en supposant par exemple que le buffer contienne `Write.Write` et que l'*EM* ait atteint l'état $q1$, nous aurions $\mathrm{Safe}(q_1, \texttt{Write.Write}) = \{\epsilon, \texttt{Write}, \texttt{Write.Write}\}$. En réalité, cette notion d'état sûr est un peu plus subtile car elle dépend du contenu du buffer. En effet, lorsque l'*EM* atteint un état non-acceptant, il peut décider de libérer un événement du buffer pour attendre finalement un état acceptant. Prenons l'exemple suivant.

Exemple 20. *Considérons la propriété décrite par l'automate de la figure 4.9 avec* $\Sigma_{\mathrm{u}} = \{u\}$ *et* $\Sigma_{\mathrm{c}} = \{c\}$. *L'état* q_2 *est sûr quel que soit le contenu du buffer, notamment si ce dernier est vide (noté* (q_2, ϵ)*).* q_1 *est non-acceptant, donc* (q_1, ϵ) *est non sûr. En revanche,* (q_1, c) *est sûr, car l'EM peut décider de libérer l'action* c *si besoin et atteindre l'état acceptant* q_2.

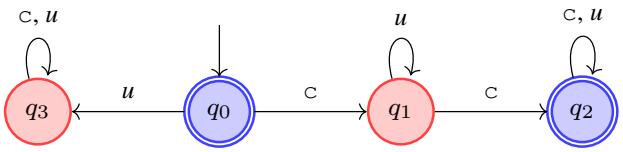


Figure 4.9 – Un exemple d'états sûrs / non-sûrs

Nous n'allons pas détailler plus ici l'écriture de l'ensemble Safe car elle est assez indigeste. Un lecteur intéressé pourra se reporter à [J7] pour plus de détails. Nous reviendrons plus tard sur cet ensemble dont le calcul sera simplifié par l'utilisation de la théorie des jeux. Nous considérons pour la suite que cette fonction Safe est définie sur $Q \times \Sigma_{\mathrm{c}}^*$.

Nous décrivons maintenant la fonction d'enforcement que nous avons définie dans [J7]. Elle correspond à l'*EM* ayant le comportement illustré dans l'exemple 19. Nous allons le décrire sous forme fonctionnelle, sur le même principe que [J6] et que celui décrit brièvement dans la section 4.2, mais sans le temps, en se basant sur une fonction inductive store, permettant de calculer à tout moment la sortie de l'*EM*, en fonction d'une séquence d'entrée donnée, et de l'état courant du buffer. Plus précisément, pour une séquence d'entrée donnée, cette fonction renvoie deux éléments : σ_s qui correspond à la sortie de l'*EM*, et σ_{buf} décrivant le contenu du buffer. Lorsqu'une action survient en entrée de l'*EM*, il faut séparer deux cas. Si cette action est incontrôlable, il faut la relâcher immédiatement, et ajouter derrière elle le plus d'actions possibles stockées dans le buffer (en sortant le plus grand mot de Safe). S'il s'agit d'une action contrôlable, on l'ajoute à la fin du buffer, et l'*EM* sort ensuite le plus long préfixe du buffer qui soit correcte par rapport à φ. Nous obtenons ainsi la définition suivante (que nous fournissons ici pour être complet, mais dont la compréhension n'est pas indispensable pour lire la suite du document) :

Définition 12 (store, E)**.** *La fonction* $\mathrm{store} : \Sigma^* \to \Sigma^* \times \Sigma_{\mathrm{c}}^*$ *est définie inductivement telle que :*

$$\mathrm{store}(\epsilon) = (\epsilon, \epsilon),$$

et pour $\sigma \in \Sigma^*$ *et* $a \in \Sigma$*, si* $(\sigma_s, \sigma_{buf}) = \mathrm{store}(\sigma)$*,*

$$\mathrm{store}(\sigma \,.\, a) = \begin{cases} (\sigma_s \,.\, a \,.\, \sigma'_s, \sigma'_{buf}) \text{ si } a \in \Sigma_{\mathrm{u}} \\ (\sigma_s \,.\, \sigma''_s, \sigma''_{buf}) \text{ si } a \in \Sigma_{\mathrm{c}}, \end{cases}$$

avec :

$$\sigma'_s = \max_{\preccurlyeq}(\mathrm{Safe}(\mathrm{Reach}(\sigma_s \,.\, a), \sigma_{buf}) \cup \{\epsilon\})$$
$$\sigma'_{buf} = \sigma'^{-1}_s \,.\, \sigma_{buf}$$
$$\sigma''_s = \max_{\preccurlyeq}(\mathrm{Safe}(\mathrm{Reach}(\sigma_s), \sigma_{buf} \,.\, a) \cup \{\epsilon\})$$
$$\sigma''_{buf} = \sigma''^{-1}_s \,.\, (\sigma_{buf} \,.\, a)$$

La fonction d'enforcement E est définie telle que : pour $\sigma \in \Sigma^*$, $\mathrm{E}(\sigma) = \Pi_1(\mathrm{store}(\sigma))$ où Π_1 représente la projection sur la première coordonnée.

Dans cette définition, σ'_s identifie les actions dans le buffer qu'il est possible d'émettre après réception de l'action incontrôlable a. C'est donc le plus grand mot de Safe calculé à partir de l'état courant atteint par a dans l'automate. σ'_{buf} correspond à l'ancienne valeur du buffer dont on retire les éléments ayant été émis. σ''_s identifie les actions dans le buffer qu'il est possible d'émettre après réception de l'action contrôlable a. Il correspond au plus long mot de Safe à partir de l'état courant de l'automate (symbolisé par $\mathrm{Reach}(\sigma_s)$), en prenant en compte le nouveau contenu du buffer : $\sigma_{buf} \,.\, a$. Comme σ'_{buf}, σ''_{buf} correspond simplement à la mise à jour de l'ancienne valeur du buffer dont les actions émises ont été retirées.

Exemple 21. *Considérons à nouveau la propriété φ_4 décrite par l'automate de la figure 4.8. Le tableau 4.1 décrit l'évolution des valeurs de σ_{buf} et σ_s de la fonction* store *pour l'entrée $\sigma =$ Auth . LockOn . `Write` . LockOff.*

Tableau 4.1 – Evolution de $(\sigma_s, \sigma_{buf}) = \mathrm{store}(\sigma)$, avec l'entrée *Auth* . *LockOn* . `Write` . *LockOff*

entrée σ	sortie σ_s	buffer σ_{buf}
ϵ	ϵ	ϵ
Auth	*Auth*	ϵ
Auth . *LockOn*	*Auth* . *LockOn*	ϵ
Auth . *LockOn* . `Write`	*Auth* . *LockOn*	`Write`
Auth . *LockOn* . `Write` . *LockOff*	*Auth* . *LockOn* . *LockOff* . `Write`	ϵ

Nous avons aussi proposé dans [J7] une version opérationnelle équivalente sous forme de système de transitions. Elle se rapproche plus de Falcone et al. [101] à ceci près que dans le cas présent, la règle à appliquer ne dépend pas seulement de l'état courant, mais aussi du contenu du buffer. La description opérationnelle que nous proposons est certes équivalente à la version fonctionnelle précédente, mais se prête mieux à une implémentation de l'*EM*. Elle reprend le même principe que la définition 8, mais en plus simple, car ici le temps n'intervient pas. Ainsi, une configuration sera simplement un couple (q, σ_{buf}) représentant respectivement l'état actuel de l'automate, et le contenu du buffer. On retrouve assez nettement les cas décrits dans la version opérationnelle de la définition 12. Trois règles s'appliquent, par ordre de priorité :

— *Dump* : lorsqu'il est possible d'émettre une ou plusieurs actions actuellement stockées dans le buffer de façon correcte, l'élément en tête du buffer est relâché. Pour cela, il est nécessaire d'utiliser la fonction Safe. L'état courant de l'automate est mis à jour.

— *Pass-uncont* : à la réception d'une action incontrôlable, cette dernière doit être relâchée immédiatement. Il faut simplement mettre à jour l'état courant de l'automate.

— *Store-cont* : lorsqu'une action contrôlable est reçue, elle est simplement stockée à la suite du buffer.

Nous obtenons ainsi la sémantique opérationnelle suivante :

Définition 13 (Mécanisme d'enforcement)**.** *Un* mécanisme d'enforcement $\mathcal{E}$ *pour* φ *est un système de transitions* $(C^{\mathcal{E}}, c_0^{\mathcal{E}}, \Gamma^{\mathcal{E}}, \hookrightarrow_{\mathcal{E}})$ *tel que :*

- $C^{\mathcal{E}} = Q \times \Sigma_c^*$ *est l'ensemble des configurations.*
- $c_0^{\mathcal{E}} = (q_0, \epsilon)$ *est la configuration initiale.*
- $\Gamma^{\mathcal{E}} = \Sigma^* \times \{\text{dump}(.), \text{pass-uncont}(.), \text{store-cont}(.)\} \times \Sigma^*$ *est l'alphabet, où les premier, deuxième et troisième éléments sont respectivement la séquence d'entrée, l'opération, et la séquence de sortie.*
- $\hookrightarrow_{\mathcal{E}} \subseteq C^{\mathcal{E}} \times \Gamma^{\mathcal{E}} \times C^{\mathcal{E}}$ *est la relation de transition, définie comme la plus petite relation obtenue en appliquant les règles suivantes par ordre de priorité (où* $w/ \bowtie /w'$ *signifie* $(w, \bowtie, w') \in \Gamma^{\mathcal{E}}$*) :*
  - *Dump :* $(q, a.\sigma_{buf}) \xhookrightarrow{\epsilon/\,\text{dump}(a)/a}_{\mathcal{E}} (q', \sigma_{buf})$*, si* $a \in \Sigma_c$*,* $\text{Safe}(q, a.\sigma_{buf}) \neq \emptyset$ *et* $\text{Safe}(q, a.\sigma_{buf}) \neq \{\epsilon\}$*, avec* $q' = q$ after a*,*
  - *Pass-uncont :* $(q, \sigma_{buf}) \xhookrightarrow{a/\,\text{pass-uncont}(a)/a}_{\mathcal{E}} (q', \sigma_{buf})$*, avec* $a \in \Sigma_u$ *et* $q' = q$ after a*,*
  - *Store-cont :* $(q, \sigma_{buf}) \xhookrightarrow{a/\,\text{store-cont}(a)/\epsilon}_{\mathcal{E}} (q, \sigma_{buf}.a)$*, avec* $a \in \Sigma_c$*.*

Exemple 22. *La figure 4.10 présente un exemple d'évolution de l'état de l'EM et les règles appliquées pour enforcer la propriété* φ_4 *modélisée par l'automate* $\mathcal{A}_{\varphi_4}$ *de la figure 4.8, avec la séquence d'entrées* $\sigma =$ *Auth . LockOn .* `Write` *. LockOff. Le mot en rouge (resp. vert, bleu) correspond à la sortie* σ_s *(resp. l'entrée, le buffer* σ_{buf}*) de l'EM. L'état courant de* $\mathcal{A}_{\varphi_4}$ *est indiqué en noir.*

$$
\begin{aligned}
&\epsilon/(q_0, \epsilon)/\textit{Auth . LockOn .}\,\texttt{Write}\,\textit{. LockOff}\\
&\qquad\qquad\downarrow \text{pass-uncont}(\textit{Auth})\\
&\textit{Auth}/(q_1, \epsilon)/\textit{LockOn .}\,\texttt{Write}\,\textit{. LockOff}\\
&\qquad\qquad\downarrow \text{pass-uncont}(\textit{LockOn})\\
&\textit{Auth . LockOn}/(q_2, \epsilon)/\texttt{Write}\,\textit{. LockOff}\\
&\qquad\qquad\downarrow \text{store-cont}(\texttt{Write})\\
&\textit{Auth . LockOn}/(q_2, \texttt{Write})/\textit{LockOff}\\
&\qquad\qquad\downarrow \text{pass-uncont}(\textit{LockOff})\\
&\textit{Auth . LockOn . LockOff}/(q_1, \texttt{Write})/\epsilon\\
&\qquad\qquad\downarrow \text{dump}(\texttt{Write})\\
&\textit{Auth . LockOn . LockOff .}\,\texttt{Write}/(q_1, \epsilon)/\epsilon
\end{aligned}
$$

Figure 4.10 – Evolution de l'*EM* avec la séquence d'entrée *Auth . LockOn .* `Write` *. LockOff* pour φ_4

Propriétés Comme nous l'avons discuté auparavant, la correction au sens de Ligatti et al. ou Schneider [200, 19] ne peut être assurée, notamment s'il existe une séquence d'actions incontrôlables menant à un état non-acceptant depuis l'état initial. Nous avons donc défini l'ensemble $\text{Pre}(\varphi)$ des mots pour lesquels la correction peut être garantie.

Définition 14 ($\mathrm{Pre}(\varphi)$)**.** $\mathrm{Pre}(\varphi) = \{\sigma \in \Sigma^* \mid \mathrm{Safe}(\mathrm{Reach}(\sigma_{|\Sigma_\mathrm{u}}), \sigma_{|\Sigma_\mathrm{c}}) \neq \emptyset\} \,.\, \Sigma^*$

Intuitivement, pour une entrée σ donnée, $\mathrm{Reach}(\sigma_{|\Sigma_\mathrm{u}})$ identifie un état qu'on ne peut éviter d'atteindre (car ce sont des actions incontrôlables), et ensuite, on regarde grâce à la fonction Safe si le contenu du buffer, ici $\sigma_{|\Sigma_\mathrm{c}}$, permet d'atteindre un état sûr. L'ensemble de ces mots forme $\mathrm{Pre}(\varphi)$, qui est bien extension-clos. Finalement, pour une propriété régulière (i.e. modélisée par un automate) quelconque φ, une fonction d'enforcement E suivant la définition 12 est correcte par rapport à φ dans $\mathrm{Pre}(\varphi)$, p-transparente par rapport à Σ_u et Σ_c, et optimale dans $\mathrm{Pre}(\varphi)$. De plus, la sortie d'un *EM* $\mathcal{E}$ suivant la définition 13 est équivalente à la sortie de la fonction E de la définition 12. Le lecteur pourra consulter les preuves dans [J7].

4.3.2 Apport de la théorie des jeux

Comme nous l'avons vu précédemment, une des difficultés principales de cette approche réside dans le calcul de Safe (l'ensemble des mots qu'il est possible de sortir à un moment donné tout en restant correct), que ça soit d'un point de vue modélisation, Safe étant assez difficile à formaliser, mais surtout performance, car ce calcul nécessite d'explorer l'arbre d'exécution complet pour toutes les sorties possibles, et ce à la volée. Avec un peu de recul, on se rend compte que l'*EE* peut en réalité être vu comme un jeu à deux joueurs, entre l'*EM* lui-même d'un côté, et l'environnement de l'autre côté. La stratégie de l'*EM* est d'essayer d'atteindre en permanence un état acceptant de l'automate décrivant la propriété, ce qui correspond assez bien à un jeu de Büchi, dans lequel le joueur essaie en permanence d'atteindre des états, appelés nœuds de Büchi. L'intérêt de cette approche est double : elle permet une écriture simplifiée de Safe, donc globalement de l'*EM*, et d'obtenir des performances meilleures, en pré-calculant certaines décisions de l'*EM* en amont. Rappelons qu'un jeu de Büchi est défini sur un graphe $\mathcal{G}$ dont les nœuds sont partitionnés en deux sous-ensembles, V_0 et V_1, correspondant respectivement au tour de chaque joueur P_0 ou P_1. Un *jeu* est un chemin de $\mathcal{G}$. L'objectif d'un jeu de Büchi est de trouver une *stratégie gagnante* assurant que tous les jeux permettent de visiter infiniment souvent un ensemble d'états objectifs, appelés nœuds de Büchi, et ce quelle que soit la stratégie de l'adversaire. Une propriété intéressante de ce type de jeu, c'est qu'il est possible de calculer l'ensemble W_0 des *nœuds gagnants* pour le joueur P_0, i.e. l'ensemble des nœuds à partir desquels il existe une stratégie gagnante pour P_0. Nous avons donc revisité l'*EE* avec une approche à base de jeu [C24] pour calculer Safe. Nous décrivons ci-dessous comment construire un graphe de jeu sur lequel il suffira ensuite de résoudre un jeu de Büchi. Nous devons donc considérer deux joueurs, l'*EM* (joueur P_0), et l'Environnement (joueur P_1). Les arêtes du graphe traduisent les actions possibles pour chacun des joueurs. Concernant l'*EM*, il a la possibilité d'émettre la première action stockée dans son buffer, ou de ne rien faire (i.e. laisser l'Environnement jouer). L'Environnement a la possibilité d'émettre (par la suite on parlera de réception du point de vue de l'*EM*) une action contrôlable ou incontrôlable, ou de ne rien faire (i.e. laisser l'*EM* jouer). Ce qui nous donne un total de cinq types d'arêtes possibles dans le graphe de jeu. Par ailleurs, nous avons vu précédemment que pour prendre une décision sur le fait d'émettre ou pas une action, il fallait prendre en compte non seulement l'état actuel atteint dans l'automate de la propriété, mais aussi le contenu du buffer. Ainsi, de façon logique et similaire aux configurations décrites dans la version opérationnelle de l'*EM*, un nœud du graphe est composé d'un état de l'automate de la propriété, et d'un mot symbolisant le contenu du buffer. Il est aussi associé à un joueur (0 ou 1) pour symboliser le tour. Précisons que le nombre de

nœuds du graphe est a priori infini. Cela vient du fait que la taille du mot composant un nœud n'est pas bornée. En réalité, il est possible de borner cet ensemble en application du Lemme de l'Etoile [12]. On notera cet ensemble de mots fini Σ_c^n par la suite. Nous obtenons donc le *graphe de jeu* suivant :

Définition 15 (Graphe de jeu)**.** *Le graphe de jeu $\mathcal{G}$ est défini par $\mathcal{G} = (V, \mathsf{E})$, tel que*

- $V = Q \times \Sigma_c^n \times \{0, 1\}$,
- $\mathsf{E} = \bigcup_{i=1}^{5} E_i$, *avec :*
  - $E_1 = \{((q, w, 0), (q, w, 1)) \in V \times V\}$,
  - $E_2 = \{((q, c.w, 0), (q \text{ after } c, w, 0)) \in V \times V \mid c \in \Sigma_c\}$,
  - $E_3 = \{((q, w, 1), (q \text{ after } u, w, 0)) \in V \times V \mid u \in \Sigma_u\}$,
  - $E_4 = \{((q, w, 1), (q, w.c, 0)) \in V \times V \mid c \in \Sigma_c\}$,
  - $E_5 = \{((q, w, 1), (q, w, 0)) \in V \times V\}$,

Une fois le graphe construit, pour obtenir l'ensemble Safe des mots menant à un état sûr, il suffit de résoudre le jeu de Büchi sur $\mathcal{G}$ en considérant comme nœuds de Büchi les nœuds correspondant à des états acceptants dans l'automate de la propriété. Finalement, après avoir calculé l'ensemble des nœuds gagnants de P_0 dans $\mathcal{G}$, l'ensemble Safe est obtenu à partir des arêtes de $\mathcal{G}$ qui permettent de rester dans un état gagnant pour P_0. Le lecteur intéressé pourra trouver la définition précise dans [C24].

Exemple 23. *Considérons à nouveau la propriété φ_4 modélisée par l'automate $\mathcal{A}_{\varphi_4}$, figure 4.8 (p. 73). Le graphe de la figure 4.11 est le graphe de jeu correspondant, avec le calcul des états gagnants intégré. L'action* `Write` *est abrégée en* `w`*. Les nœuds de Büchi sont doublement cerclés, et les états gagnants du joueur P_0 (l'EM), i.e. W_0, sont en bleu. Les arêtes utilisent la convention de couleurs suivante :*

- *les arêtes bleues (⟶) appartiennent à E_1 (l'EM (P_0) passe son tour),*
- *les arêtes vertes (⟶) appartiennent à E_2 (émission de l'élément en tête du buffer),*
- *les arêtes oranges (⟶) appartiennent à E_4 (réception d'une action contrôlable),*
- *les arêtes rouges (⟶) appartiennent à $E_3 \cup E_5$ (réception d'une action incontrôlable, ou l'Environnement (P_1) passe son tour).*

Pour des raisons de lisibilité, les étiquettes des arêtes n'ont pas été intégrées à la figure. Afin d'illustrer plus précisément ce modèle, la figure 4.12 "zoome" sur une partie du graphe de jeu. Le nœud initial du graphe étant noir, cela signifie qu'il n'est pas possible de garantir la propriété depuis le début. Toutefois, l'occurrence de l'action Auth en $(q_0, -, 1)$ mène en $(q_1, -, 0)$ qui est gagnant, i.e. à partir de cet instant la propriété peut être garantie. Dans le nœud gagnant $(q_2, w, 0)$, le fait pour P_0 de passer son tour (stop) permet de rester dans un nœud gagnant, alors que si P_0 décide d'émettre la seule action w de son buffer, il arrive au nœud $(q_3, \epsilon, 0)$ qui est perdant.

Une fois que l'on a redéfini Safe de la sorte, le reste est inchangé, car cet ensemble est utilisé à la fois dans la version fonctionnelle et dans la version opérationnelle de l'*EM*. Nous fournissons cependant ci-dessous une autre écriture sous forme opérationnelle de l'*EM*. Cette version est très proche de celle que nous avons vue précédemment, et équivalente, mais elle

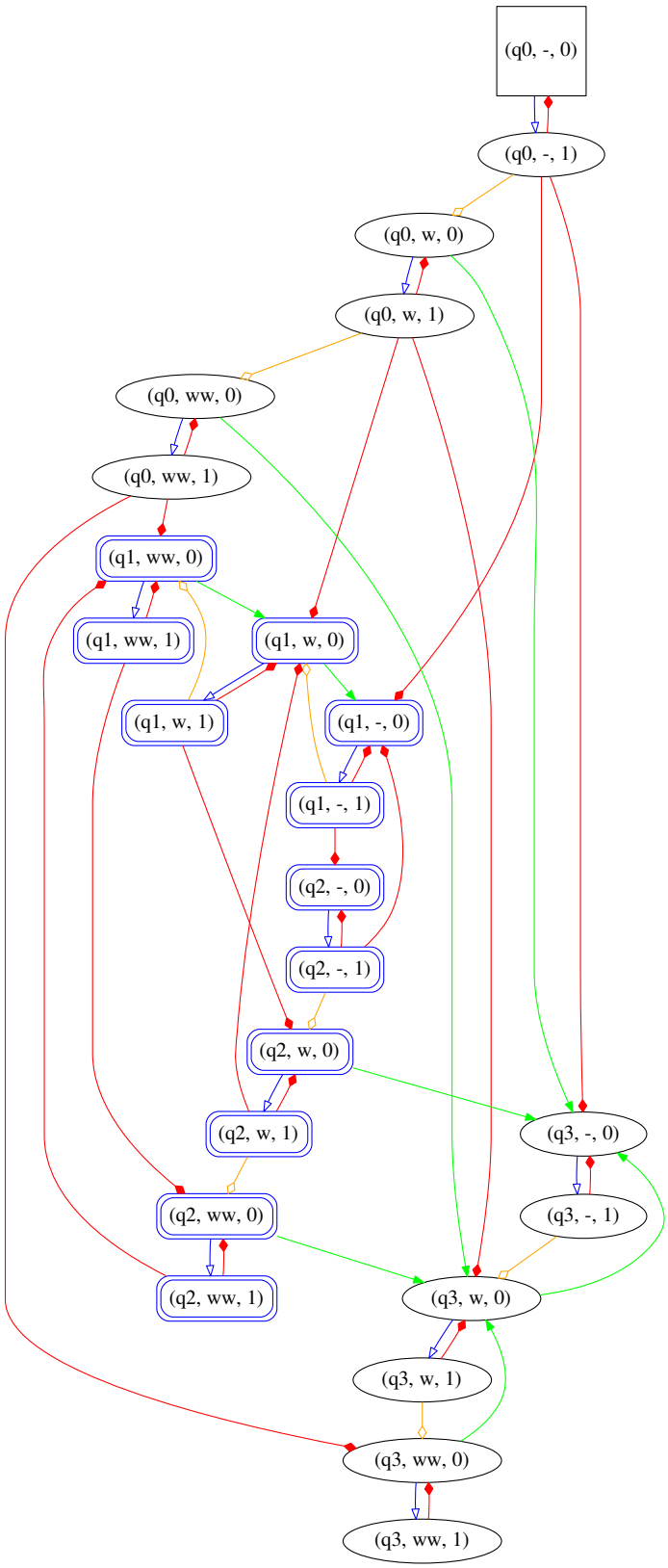


Figure 4.11 – Graphe de jeu associé à la propriété φ_4

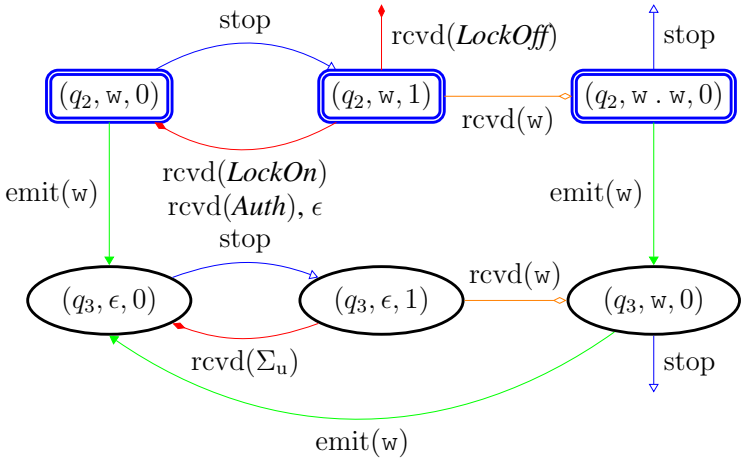


Figure 4.12 – Zoom sur certains nœuds et certaines arrêtes du graphe de jeu associé à φ_4

permet de mieux cerner l'intuition de l'utilisation du graphe de jeu. Nous reprenons donc la définition 13, sous forme simplifiée, en considérant des actions quelconques $a, c \in \Sigma_\mathrm{c}$, et $u \in \Sigma_\mathrm{u}$, et nous y définissons les règles suivantes, toujours par ordre de priorité :

— Dump : $(q, a.\sigma_{buf}) \xhookrightarrow{\epsilon/\,\mathrm{dump}(a)/a}_{\mathcal{E}} (q', \sigma_{buf})$, avec $q' = q$ after a, si $(q', \sigma_{buf}, 0) \in W_0$ i.e. est un nœud gagnant pour P_0.

— Pass-uncont : $(q, \sigma_{buf}) \xhookrightarrow{u/\,\text{pass-uncont}(u)/u}_{\mathcal{E}} (q \text{ after } u, \sigma_{buf})$

— Store-cont : $(q, \sigma_{buf}) \xhookrightarrow{c/\,\text{store-cont}(c)/\epsilon}_{\mathcal{E}} (q, \sigma_{buf}.c)$, avec $c \in \Sigma_\mathrm{c}$.

Lorsque l'*EM* est décrit sous cette forme, l'utilisation du graphe de jeu est plus explicite. Au final, cette approche à base de jeu simplifie assez nettement l'écriture de l'*EM*. Par ailleurs, cette approche transforme une complexité "temporelle" en une complexité "spatiale". En effet, il est nécessaire de pré-calculer et de stocker l'intégralité du graphe de jeu, mais une fois que cela a été fait, les calculs à la volée sont peu coûteux. En réalité, cette simplification d'écriture et cette amélioration des performances seront plus significatives dans le cadre temporisé que nous verrons par la suite.

4.3.3 Enforcement à l'exécution de propriétés temporisées en présence d'événements incontrôlables

Dans cette partie, nous exposons nos contributions concernant l'*EE* de propriété décrite par un TA quelconque, en présence d'événements incontrôlables. Il s'agit d'une extension de la technique vue précédemment. Toutefois, même si le principe général est très similaire à l'approche non temporisée, le fait d'ajouter le temps dans la formalisation alourdit fortement les notations, rendant certaines définitions difficilement lisibles. Pour cette raison, nous avons choisi pour cette partie de ne pas détailler les définitions, mais plutôt de focaliser sur les idées menant à cette approche. Le lecteur intéressé par l'ensemble des formalisations pourra les trouver dans [J7]. Nous considérons dans cette partie φ une propriété temporisée modélisée par un TA $\mathcal{A}_\varphi = (L, l_0, X, \Sigma, \Delta, G)$ et sa sémantique $[\![\mathcal{A}_\varphi]\!] = (Q, q_0, \Gamma, \to, F_G)$. L'alphabet Σ est à nouveau partitionné en deux sous-ensembles $\Sigma_\mathrm{u} \subseteq \Sigma$ l'ensemble des actions *incontrôlables*,

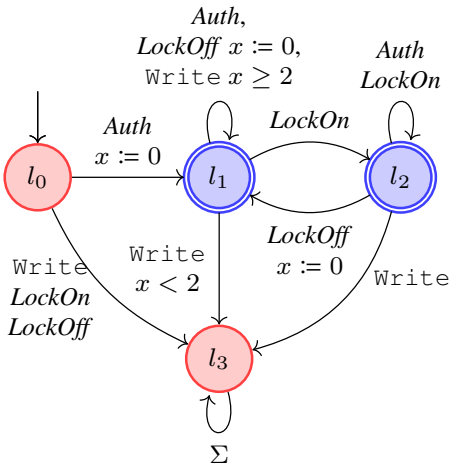


Figure 4.13 – Propriété φ_5 modélisée par le TA $\mathcal{A}_{\varphi_5}$

i.e. qui ne peuvent être modifiées par l'*EM*, et $\Sigma_c = \Sigma \setminus \Sigma_u$ les autres actions, dites *contrôlables*. Comme dans la section 4.2, l'*EM* est équipé d'un buffer pouvant retenir des actions (uniquement contrôlables), et les relâcher plus tard. L'*EM* est une fonction d'enforcement de $\mathrm{tw}(\Sigma) \times \mathbb{R}_{\geq 0}$ dans $\mathrm{tw}(\Sigma)$. A l'instar de l'approche non temporisée, les événements incontrôlables rendent le problème nettement plus difficile. De façon évidente, nous retrouvons le fait qu'il n'est pas toujours possible d'enforcer une propriété depuis l'état initial s'il existe une séquence d'événements incontrôlables menant à un état non-acceptant. De plus, dans la section 4.2, lorsqu'une séquence d'événements était mise en attente dans le buffer, l'ordre de ces événements ainsi que les délais entre eux restaient inchangés. Dans le cas présent, si un événement incontrôlable survient, il doit être émis instantanément, et cette émission doit être prise en compte dans le calcul des délais des événements dans le buffer. Comme dans la section 4.2 et [J6], les calculs se font sur le graphe de zones correspondant au TA de la propriété. Par ailleurs, nous avons fait le choix dans cette partie d'autoriser l'*EM* à "rattraper le temps", ce qui signifie que le délai entre deux événements contrôlables peut être réduit par l'*EM* tant que l'ordre de ces événements est maintenu et que la propriété est vérifiée. Cette hypothèse semble raisonnable dans bon nombre d'applications. Remarquons que le fait d'imposer à l'*EM* de maintenir ces délais comme dans la section 4.2 est possible, et ne nécessiterait pas beaucoup de modifications. Afin de donner les intuitions liées à la synthèse de l'*EM*, nous allons illustrer son comportement, tel que nous l'avons formalisé dans [J7], au travers d'un exemple.

Exemple 24. *Considérons la propriété φ_5 modélisée par le TA $\mathcal{A}_{\varphi_5} = (L, l_0, X, \Sigma, \Delta, G)$ de la figure 4.13. Les états acceptants sont toujours représentés en bleu et doublement cerclés. Il s'agit d'une version temporisée de la mémoire partagée φ_4 vue figure 4.8. L'ensemble des événements est toujours $\Sigma_c = \{\texttt{Write}\}$ et $\Sigma_u = \{$Auth, LockOff, LockOn$\}$. Comme dans φ_4, il faut attendre que l'environnement ait déverrouillé la mémoire partagée (LockOff) pour pouvoir écrire (`Write`), mais cette fois il est nécessaire d'attendre 2 unités de temps après déverrouillage (ou authentification Auth) pour pouvoir écrire. En effet, toutes les transitions arrivant à la localité l_1 remettent l'horloge x à zéro. Depuis l_1, émettre l'événement `Write` après 2 unités de temps fait reboucler dans la même localité, alors que l'émettre avant 2 unités de temps mène à la localité l_3, qui est un état non-acceptant. Considérons un EM cherchant à enforcer cette propriété avec l'entrée $\sigma = ((1, $Auth$).(1, $LockOn$).(2, \texttt{Write}).(1, $LockOff$).(1, $LockOn$).(1, \texttt{Write}).(1, $LockOff$))$. Au temps $t = 1$, l'événement incontrôlable Auth survient. Il doit être*

émis instantané, et $\mathcal{A}_{\varphi_5}$ passe en l_1. En $t = 2$, l'événement incontrôlable LockOn est reçu. Il est aussi émis instantanément, menant $\mathcal{A}_{\varphi_5}$ en l_2. A $t = 4$ l'événement contrôlable `Write` arrive, il ne doit pas être émis sous peine d'aller en l_3 qui est un état non-acceptant. Il est donc placé dans le buffer. Il n'est pas nécessaire de stocker de délai associé à cette action, car tant qu'un événement incontrôlable n'arrive pas, il n'est pas possible de calculer un délai de sortie de `Write`. Au temps $t = 5$ l'événement incontrôlable LockOff se produit, il doit être émis immédiatement, et $\mathcal{A}_{\varphi_5}$ se déplace en l_1. Cette fois, il est possible d'émettre l'événement `Write` actuellement en mémoire, mais il faut pour cela attendre 2 unités de temps. Cette valeur est associée à `Write` qui est en attente d'émission, à la manière décrite dans la partie 4.2.3 et dans [J6], et il sera émis si aucun événement incontrôlable ne se produit entre-temps. A $t = 6$ l'événement incontrôlable LockOn se produit, alors $\mathcal{A}_{\varphi_5}$ passe en l_2, et il n'est à nouveau plus possible d'émettre `Write`, qui n'est plus associé à un délai de sortie. Au temps $t = 7$, le deuxième événement `Write` qui arrive sera aussi placé dans le buffer sans possibilité de déterminer sa date de sortie. L'occurrence en $t = 8$ de l'action incontrôlable LockOff débloque la situation, menant $\mathcal{A}_{\varphi_5}$ en l_1, et les deux actions `Write` dans le buffer sont à nouveau associées à des délais de sortie, respectivement 2 et 0, qui signifient qu'ils seront tous les deux émis en $t = 10$ si aucun événement incontrôlable ne survient entre-temps. Ainsi, pour l'entrée donnée $\sigma = ((1, \textit{Auth}) \,.\, (1, \textit{LockOn}) . (2, \texttt{Write}) . (1, \textit{LockOff}) . (1, \textit{LockOn}) . (1, \texttt{Write}) . (1, \textit{LockOff}))$ la sortie de l'EM est $(1, \textit{Auth}) \,.\, (1, \textit{LockOn}) \,.\, (3, \textit{LockOff}) \,.\, (1, \textit{LockOn}) \,.\, (2, \textit{LockOff}) \,.\, (2, \texttt{Write}) \,.\, (0, \texttt{Write})$

Les notions de correction, p-transparence et optimalité (définitions 9, 10 et 11) ont été étendues dans un cadre temporisé de façon assez naturelle. Pour la correction, comme à la section 4.2, nous considérons qu'un *EM* est correct si sa sortie finit par satisfaire la propriété φ (dans le futur). Comme dans le cas non temporisé, il est nécessaire de définir la correction sur un ensemble extension-clos, afin de prendre en compte les propriétés non enforçables depuis l'état initial. Cette nouvelle définition est donc très proche de la définition 5 à ceci près qu'il n'est plus nécessaire de considérer la sortie différente d'ϵ, ce problème étant résolu par la réduction à un ensemble extension-clos. La définition de p-transparence reprend naturellement les trois cas de la définition 10 sur des mots temporisés, donnant en plus la possibilité de retarder des actions contrôlables, comme dans la section 4.2, ce qui donne :

— l'ordre de événements contrôlables ne doit pas être modifié, i.e. si on considère la projection sur les événements contrôlables, la sortie doit être un préfixe retardé ($\preccurlyeq_d$) de l'entrée.

— tout événement incontrôlable doit être émis immédiatement, i.e. les projections sur les événements incontrôlables de l'entrée et de la sortie à un temps donné doivent être égales,

— la causalité doit être préservée,

Nous avons redéfini la notion de préfixe retardé ($\preccurlyeq_d$) de la section 4.2 utilisée dans [J6], en considérant les dates et non plus les délais, ce qui revient à $\sigma' \preccurlyeq_d \sigma$ si $\Pi_\Sigma(\sigma') \preccurlyeq \Pi_\Sigma(\sigma)$ et $\forall i \leq |\sigma'|, \text{date}(\sigma(i)) \leq \text{date}(\sigma'(i))$, où *date* correspond à la date d'occurrence de l'événement. Cette définition autorise en quelque sorte l'*EM* à "rattraper" le temps entre événements contrôlables, tant que la date en sortie reste supérieure à la date d'entrée. Enfin, l'optimalité reprend les idées de la définition 11. Elle est définie sur un ensemble extension-clos, considérant que le fait d'émettre un mot plus long (au sens de $\preccurlyeq_d$) qu'un *EM* optimal signifie que la correction ou la p-transparence ne peuvent plus être garanties. Le lecteur intéressé par ces définitions précises pourra les trouver dans [J7].

Comme dans le cas non temporisé, la synthèse de l'*EM*, que ce soit sous forme fonctionnelle ou opérationnelle, dépend de l'ensemble Safe des mots qu'il est possible d'émettre à un moment

donné tout en restant correct, i.e. menant à un état sûr. Le fait d'ajouter le temps complique encore les choses, car non seulement l'occurrence d'un événement non contrôlable peut mener à un état non sûr, mais parfois le simple écoulement du temps peut mener d'un état sûr à un état non sûr. Comme dans le cas non temporisé, la décision d'émettre ou non un événement contrôlable dépend de deux éléments : l'état symbolique atteint par la sémantique du TA, et le contenu du buffer. Comme nous l'avons évoqué dans l'exemple précédent, Safe n'a pas besoin des délais entre les événements pour calculer la sortie de l'*EM*, ainsi, la fonction Safe est définie sur $Q \times \Sigma_c^*$. Il est à nouveau possible d'utiliser un jeu de Büchi pour synthétiser l'*EM*, en utilisant le même principe que dans le cas non temporisé. Pour cela, on définit à nouveau à graphe de jeu à deux joueurs, P_0 étant l'*EM*, P_1 l'environnement. Les nœuds du graphe symbolisent les configurations possibles de l'*EM*, composés d'un état symbolique, un mot de Σ_c^* représentant le buffer, et un joueur (0 ou 1). On résout ce jeu en considérant comme nœuds de Büchi les nœuds correspondants à des états acceptants de la sémantique du TA. Les actions possibles des joueurs sont identiques à la définition 15, à ceci près que lorsque l'environnement passe son tour, cela signifie que le temps s'écoule. De plus, les jeux de Büchi étant infinis, il doit exister une transition sortante sur chaque nœud. Pour cette raison, une transition supplémentaire de type boucle est ajoutée si besoin pour permettre d'éviter ce genre de blocage. Ce graphe possède a priori un nombre infini de nœuds pour deux raisons : la sémantique d'un TA a elle-même un nombre infini d'états, et la taille du buffer n'est pas bornée. En réalité, il est possible de borner le nombre de nœuds de ce graphe de jeu en se basant sur une représentation symbolique du TA, comme par exemple le graphe des régions [6] ou encore un graphe de zones [22], et de réduire l'ensemble des mots (non temporisés) du buffer à un ensemble borné Σ_c^n comme vu à la partie 4.3.2 pour le cas non temporisé. La définition précise du graphe de jeu est disponible dans [193], nous nous contenterons ici de l'illustrer au travers d'un exemple.

Exemple 25. *Le graphe de jeu associé à la propriété φ_5, dans lequel les nœuds gagnants ont été calculés, est représenté figure 4.14. Les étiquettes des arêtes n'ont pas été mises pour ne pas alourdir la figure. L'action* `Write` *est abrégée en* `w`*. Les nœuds de Büchi sont doublement cerclés, et les nœuds gagnants du joueur P_0 (l'EM), i.e. W_0, sont en bleu. Les arêtes utilisent la convention de couleurs suivante :*

- *les arêtes bleues (⟶) signifient que l'EM passe son tour*
- *les arêtes vertes (⟶) signifient que l'EM émet l'événement en tête de son buffer*
- *les arêtes oranges (⟶) signifient qu'un événement contrôlable est reçu*
- *les arêtes rouges (⟶) signifient qu'un événement incontrôlable est reçu, ou qu'il n'y a plus d'événement à recevoir*
- *les arêtes violettes (⟶) symbolisent l'écoulement du temps.*

Supposons par exemple que le nœud $(l_1, \{x < 2\}, -, 1)$ soit atteint. C'est donc le tour de l'Environnement. Pour sortir de ce nœud, on identifie les cas suivants :

- *réception de l'événement incontrôlable LockOn qui mène au nœud $(l_2, T, -, 0)$ qui correspond bien à la localité l_2 du TA.*
- *réception de l'événement incontrôlable LockOff ou Auth qui mène au nœud $(l_1, \{ x < 2\}, -, 0)$.*
- *réception de l'événement contrôlable* `Write` *qui mène au nœud $(l_1, \{x < 2\}, \mathtt{w}, 0)$, ce qui signifie que l'événement* `Write` *a été ajouté au buffer.*

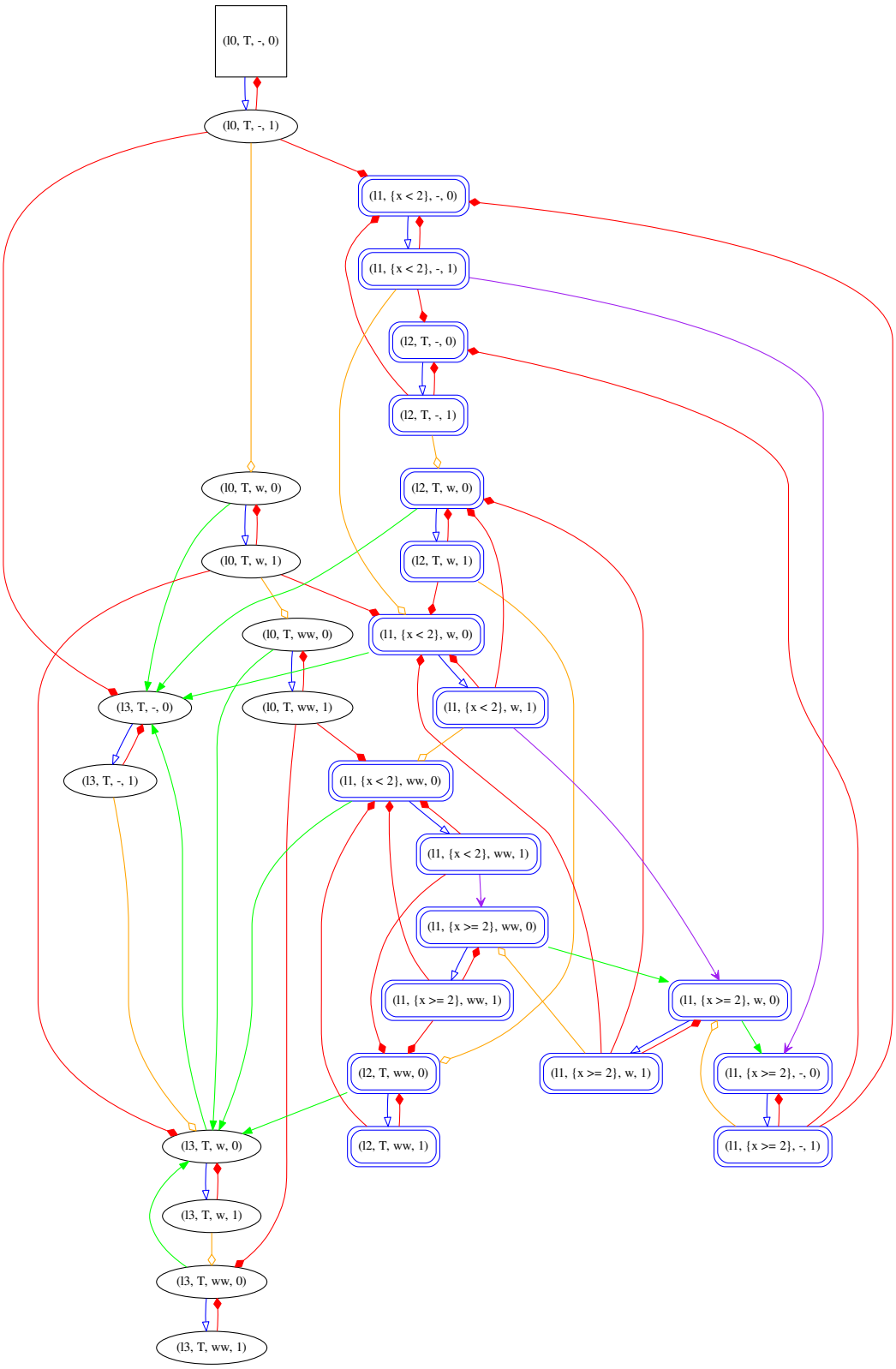


Figure 4.14 – Graphe de jeu associé à la propriété φ_5

— *écoulement du temps, qui mène au nœud* $(l_1, \{x \geq 2\}, -, 0)$*, mettant ainsi à jour l'état symbolique.*

Dans ces quatre cas, l'Environnement rend la main à l'EM.

Finalement, le graphe de jeu est utilisé pour le calcul de Safe de façon similaire à la version non temporisée, comme expliqué dans la partie 4.3.2.

La version fonctionnelle de l'*EM* reprend le même principe que la définition 12 : une fonction store qui renvoie deux valeurs correspondant à la sortie σ_s et à l'état du buffer σ_{buf}, le tout en calculant la "meilleure" séquence de Safe. Sur le même principe que nous avons utilisé dans [J6] et dans la partie 4.2.3 : il est nécessaire d'adapter ces notions dans un cadre temporisé : σ_s correspond aux événements (contrôlables) prêts à être sortis (quand la date sera atteinte), alors que σ_{buf} correspond aux événements (contrôlables) stockés ne permettant pas (encore) d'atteindre un état acceptant. La différence notable par rapport à [J6] et la partie 4.2.3, c'est que lorsqu'un événement est dans σ_s il n'y pas de certitude qu'il finisse réellement par être émis par l'*EM* tant que sa date d'émission n'est pas atteinte. En effet, si un événement incontrôlable survient entre-temps, ce dernier doit être émis immédiatement, et l'état sur le TA de la propriété est mis à jour. A ce moment, il est nécessaire de revérifier si les événements de σ_s peuvent encore être émis sans violer la propriété. Finalement, store va donc renvoyer deux mots : σ_∞ un mot temporisé correspondant à la sortie de l'*EM* à un temps infini, i.e. les événements déjà émis par l'*EM* suivis des événements contrôlables en attente d'être émis sous réserve qu'il n'y ait pas d'événement incontrôlable qui survienne entre-temps, et σ_c un mot non temporisé correspondant aux événements contrôlables retenus pour ne pas violer la propriété. La fonction store est définie de façon inductive en mettant à jour ces deux mots lorsque survient un événement, en séparant le cas d'un événement incontrôlable et d'un événement contrôlable. Dans les deux cas, il est nécessaire de calculer la meilleure séquence correcte à l'aide de Safe. La définition détaillée de store se trouve dans [J7, 193]. [13]

Exemple 26. *Considérons à nouveau la propriété* φ_5 *modélisée par le TA* $\mathcal{A}_{\varphi_5} = (L, l_0, X, \Sigma, \Delta, G)$ *de la figure 4.13. Le tableau 4.2 montre l'évolution des mots* σ_∞*,* σ_c *et de la sortie de l'EM dans* store *pour l'entrée* $\sigma = (1, \textit{Auth}) \,.\, (1, \textit{LockOn}) \,.\, (2, \texttt{Write}) \,.\, (1, \textit{LockOff}) \,.\, (1, \textit{LockOn}) \,.\, (1, \texttt{Write}) \,.\, (1, \textit{LockOff})$ *comme dans l'exemple précédent. Comme nous l'avons vu, au temps* $t = 4$*, l'événement contrôlable* `Write` *survient et ne peut être émis. Il est donc ajouté à* σ_c*. En* $t = 5$*, l'occurrence de l'événement incontrôlable LockOff amène le TA dans la localité* l_1*, et permet d'envisager d'émettre* `Write`*. Ainsi, la valeur "*$(3, \textit{LockOff}) \,.\, (2, \texttt{Write})$*" est ajoutée à la suite de* σ_∞*, ce qui correspond à l'émission instantanée de LockOff suivie de l'émission prévue de* `Write` *au bout de 2 unités de temps. L'occurrence en* $t = 6$ *de LockOn amène le TA en* l_2*. L'émission de* `Write` *n'est donc plus envisageable, et ce dernier est donc remis dans* σ_c*.*

La version opérationnelle (sous forme de système de transitions) de l'*EM* est une simple extension temporisée de la définition 13, en utilisant le même principe que celui de [J6] décrit dans la partie 4.2.3. Une configuration est un quadruplet $(\sigma_b, \sigma_c, q, \delta)$ où σ_b représente les événements contrôlables en attente d'être émis (à un temps infini), ce qui correspond à σ_∞ vu ci-dessus sans les événements déjà émis ; σ_c représente les événements contrôlables retenus pour ne pas violer la propriété (comme dans la version fonctionnelle de l'*EM*), q correspond à l'état courant dans

13. Le lecteur trouvera dans [J7] une version à base de dates, alors qu'une version à base de délais, plus proche de celle décrite ici, est disponible dans [193].

Tableau 4.2 – Tableau montrant l'évolution de σ_∞, σ_c et de la sortie de l'*EM* dans store pour l'entrée $\sigma = ((1, \mathit{Auth}) \,.\, (1, \mathit{LockOn}) \,.\, (2, \texttt{Write}) \,.\, (1, \mathit{LockOff}) \,.\, (1, \mathit{LockOn}) \,.\, (1, \texttt{Write}) \,.\, (1, \mathit{LockOff}))$.

t	Sortie de l'*EM* au temps t	σ_∞	σ_c
1	$(1, \mathit{Auth})$	$(1, \mathit{Auth})$	ϵ
2	$(1, \mathit{Auth}) \,.\, (1, \mathit{LockOn})$	$(1, \mathit{Auth}) \,.\, (1, \mathit{LockOn})$	ϵ
4	$(1, \mathit{Auth}) \,.\, (1, \mathit{LockOn})$	$(1, \mathit{Auth}) \,.\, (1, \mathit{LockOn})$	Write
5	$(1, \mathit{Auth}) \,.\, (1, \mathit{LockOn}) \,.\, (3, \mathit{LockOff})$	$(1, \mathit{Auth}) \,.\, (1, \mathit{LockOn}) \,.\, (3, \mathit{LockOff}) \,.$ $(2, \texttt{Write})$	ϵ
6	$(1, \mathit{Auth}) \,.\, (1, \mathit{LockOn}) \,.\, (3, \mathit{LockOff}) \,.$ $(1, \mathit{LockOn})$	$(1, \mathit{Auth}) \,.\, (1, \mathit{LockOn}) \,.\, (3, \mathit{LockOff}) \,.$ $(1, \mathit{LockOn})$	Write
7	$(1, \mathit{Auth}) \,.\, (1, \mathit{LockOn}) \,.\, (3, \mathit{LockOff}) \,.$ $(1, \mathit{LockOn})$	$(1, \mathit{Auth}) \,.\, (1, \mathit{LockOn}) \,.\, (3, \mathit{LockOff}) \,.$ $(1, \mathit{LockOn})$	Write . Write
8	$(1, \mathit{Auth}) \,.\, (1, \mathit{LockOn}) \,.\, (3, \mathit{LockOff}) \,.$ $(1, \mathit{LockOn}) \,.\, (2, \mathit{LockOff})$	$(1, \mathit{Auth}) \,.\, (1, \mathit{LockOn}) \,.\, (3, \mathit{LockOff}) \,.$ $(1, \mathit{LockOn}) \,.\, (2, \mathit{LockOff}) \,.\, (2, \texttt{Write}) \,.$ $(0, \texttt{Write})$	ϵ
10	$(1, \mathit{Auth}) \,.\, (1, \mathit{LockOn}) \,.\, (3, \mathit{LockOff}) \,.$ $(1, \mathit{LockOn}) \,.\, (2, \mathit{LockOff}) \,.\, (2, \texttt{Write}) \,.$ $(0, \texttt{Write})$	$(1, \mathit{Auth}) \,.\, (1, \mathit{LockOn}) \,.\, (3, \mathit{LockOff}) \,.$ $(1, \mathit{LockOn}) \,.\, (2, \mathit{LockOff}) \,.\, (2, \texttt{Write}) \,.$ $(0, \texttt{Write})$	ϵ

le TA de la propriété, et δ le temps écoulé depuis le dernier événement émis. De façon naturelle, une règle supplémentaire symbolisant l'écoulement du temps a été ajoutée. Pour chacune de ces règles, les différents éléments composant une configuration sont mis à jour. Le lecteur pourra trouver la sémantique formelle complète dans [J7, 193].

Exemple 27. *La figure 4.15 présente un exemple d'évolution de l'état de l'EM et les règles appliquées pour enforcer la propriété φ_5 modélisée par le TA $\mathcal{A}_{\varphi_5}$ de la figure 4.13, avec la même séquence d'entrée que précédemment : $\sigma = ((1, \mathit{Auth}) \,.\, (1, \mathit{LockOn}) \,.\, (2, \texttt{Write}) \,.\, (1, \mathit{LockOff}) \,.$ $(1, \mathit{LockOn}) \,.\, (1, \texttt{Write}) \,.\, (1, \mathit{LockOff}))$. Le mot temporisé en rouge (resp. vert, bleu) correspond à la sortie (resp. l'entrée, σ_b) de l'EM. Le mot non temporisé en noir correspond à σ_c. L'état courant de $\mathcal{A}_{\varphi_5}$ ainsi que δ sont aussi en noir. Pour des raisons de place, LockOff est représenté par off, LockOn par on, et* Write *par* w.

Propriétés Un *EM* défini sous forme opérationnelle ou fonctionnelle comme décrit ci-dessus est correct, p-transparent et optimal. Le lecteur pourra consulter les preuves dans [J7, 193].

Nous avons développé un prototype, l'outil *GREP* [14], mettant en œuvre la synthèse de l'*EM* par la théorie des jeux, et ce pour des propriétés temporisées (ou non) en présence d'événements incontrôlables. L'architecture de cet outil, développé en C, est fournie figure 4.16. Nous allons décrire brièvement les éléments composant ce prototype. Le lecteur trouvera s'il le souhaite une description détaillée dans [C25]. Le moteur de l'*EM* est composé de deux éléments, le *Symbolic Computing Module (SCM)* qui gère tous les aspects liés à la gestion symbolique du temps, et tout ce qui doit être pré-calculé : chargement du TA, génération du graphe de zones, et construction du graphe de jeu avec les nœuds gagnants. Ensuite, l'*Enforcement Monitor (EM)* se charge

14. "Games for Runtime Enforcement of Properties", disponible à l'adresse https://github.com/matthieurenard/GREP.

$$
\begin{array}{ll}
\text{t = 0} & \epsilon/\langle\epsilon,\epsilon,(l_0,0),0\rangle/(1,\mathit{Auth}).(1,\mathit{on}).(2,\mathtt{w}).(1,\mathit{off}).(1,\mathit{on}).(1,\mathtt{w}).(1,\mathit{off}) \\
& \downarrow \text{elapse}(1) \\
\text{t = 1} & \epsilon/\langle\epsilon,\epsilon,(l_0,1),1\rangle/(0,\mathit{Auth}).(1,\mathit{on}).(2,\mathtt{w}).(1,\mathit{off}).(1,\mathit{on}).(1,\mathtt{w}).(1,\mathit{off}) \\
& \downarrow \text{pass-uncont}(\mathit{Auth}) \\
\text{t = 1} & (1,\mathit{Auth})/\langle\epsilon,\epsilon,(l_1,0),0\rangle/(1,\mathit{on}).(2,\mathtt{w}).(1,\mathit{off}).(1,\mathit{on}).(1,\mathtt{w}).(1,\mathit{off}) \\
& \downarrow \text{elapse}(1) \\
\text{t = 2} & (1,\mathit{Auth})/\langle\epsilon,\epsilon,(l_1,1),1\rangle/(1,\mathit{on}).(2,\mathtt{w}).(1,\mathit{off}).(1,\mathit{on}).(1,\mathtt{w}).(1,\mathit{off}) \\
& \downarrow \text{pass-uncont}(\mathit{on}) \\
\text{t = 2} & (1,\mathit{Auth}).(1,\mathit{on})/\langle\epsilon,\epsilon,(l_2,1),0\rangle/(2,\mathtt{w}).(1,\mathit{off}).(1,\mathit{on}).(1,\mathtt{w}).(1,\mathit{off}) \\
& \downarrow \text{elapse}(2) \\
\text{t = 4} & (1,\mathit{Auth}).(1,\mathit{on})/\langle\epsilon,\epsilon,(l_2,3),2\rangle/(2,\mathtt{w}).(1,\mathit{off}).(1,\mathit{on}).(1,\mathtt{w}).(1,\mathit{off}) \\
& \downarrow \text{store-cont}(\mathtt{w}) \\
\text{t = 4} & (1,\mathit{Auth}).(1,\mathit{on})/\langle\epsilon,\mathtt{w},(l_2,3),2\rangle/(1,\mathit{off}).(1,\mathit{on}).(1,\mathtt{w}).(1,\mathit{off}) \\
& \downarrow \text{elapse}(1) \\
\text{t = 5} & (1,\mathit{Auth}).(1,\mathit{on})/\langle\epsilon,\mathtt{w},(l_2,4),3\rangle/(1,\mathit{off}).(1,\mathit{on}).(1,\mathtt{w}).(1,\mathit{off}) \\
& \downarrow \text{pass-uncont}(\mathit{off}) \\
\text{t = 5} & (1,\mathit{Auth}).(1,\mathit{on}).(3,\mathit{off})/\langle(2,\mathtt{w}),\epsilon,(l_1,0),0\rangle/(1,\mathit{on}).(1,\mathtt{w}).(1,\mathit{off}) \\
& \downarrow \text{elapse}(1) \\
\text{t = 6} & (1,\mathit{Auth}).(1,\mathit{on}).(3,\mathit{off})/\langle(2,\mathtt{w}),\epsilon,(l_1,1),1\rangle/(1,\mathit{on}).(1,\mathtt{w}).(1,\mathit{off}) \\
& \downarrow \text{pass-uncont}(\mathit{on}) \\
\text{t = 6} & (1,\mathit{Auth}).(1,\mathit{on}).(3,\mathit{off}).(1,\mathit{on})/\langle\epsilon,\mathtt{w},(l_2,1),0\rangle/(1,\mathtt{w}).(1,\mathit{off}) \\
& \downarrow \text{elapse}(1) \\
\text{t = 7} & (1,\mathit{Auth}).(1,\mathit{on}).(3,\mathit{off}).(1,\mathit{on})/\langle\epsilon,\mathtt{w},(l_2,2),1\rangle/(1,\mathtt{w}).(1,\mathit{off}) \\
& \downarrow \text{store-cont}(\mathtt{w}) \\
\text{t = 7} & (1,\mathit{Auth}).(1,\mathit{on}).(3,\mathit{off}).(1,\mathit{on})/\langle\epsilon,\mathtt{w}.\mathtt{w},(l_2,2),1\rangle/(1,\mathit{off}) \\
& \downarrow \text{elapse}(1) \\
\text{t = 8} & (1,\mathit{Auth}).(1,\mathit{on}).(3,\mathit{off}).(1,\mathit{on})/\langle\epsilon,\mathtt{w}.\mathtt{w},(l_2,3),2\rangle/(1,\mathit{off}) \\
& \downarrow \text{pass-uncont}(\mathit{off}) \\
\text{t = 8} & (1,\mathit{Auth}).(1,\mathit{on}).(3,\mathit{off}).(1,\mathit{on}).(2,\mathit{off})/\langle(2,\mathtt{w}).(0,\mathtt{w}),\epsilon,(l_1,0),0\rangle/\epsilon \\
& \downarrow \text{elapse}(2) \\
\text{t = 10} & (1,\mathit{Auth}).(1,\mathit{on}).(3,\mathit{off}).(1,\mathit{on}).(2,\mathit{off})/\langle(2,\mathtt{w}).(0,\mathtt{w}),\epsilon,(l_1,2),2\rangle/\epsilon \\
& \downarrow \text{dump}((2,\mathtt{w})) \\
\text{t = 10} & (1,\mathit{Auth}).(1,\mathit{on}).(3,\mathit{off}).(1,\mathit{on}).(2,\mathit{off}).(2,\mathtt{w})/\langle(0,\mathtt{w}),\epsilon,(l_1,2),0\rangle/\epsilon \\
& \downarrow \text{dump}((0,\mathtt{w})) \\
\text{t = 10} & (1,\mathit{Auth}).(1,\mathit{on}).(3,\mathit{off}).(1,\mathit{on}).(2,\mathit{off}).(2,\mathtt{w}).(0,\mathtt{w})/\langle\epsilon,\epsilon,(l_1,2),0\rangle/\epsilon
\end{array}
$$

Figure 4.15 – Evolution de l'*EM* avec la séquence d'entrée $(1,\mathit{Auth}).(1,\mathit{LockOn}).(2,\mathtt{Write}).(1,\mathit{LockOff}).(1,\mathit{LockOn}).(1,\mathtt{Write}).(1,\mathit{LockOff})$ pour φ_5

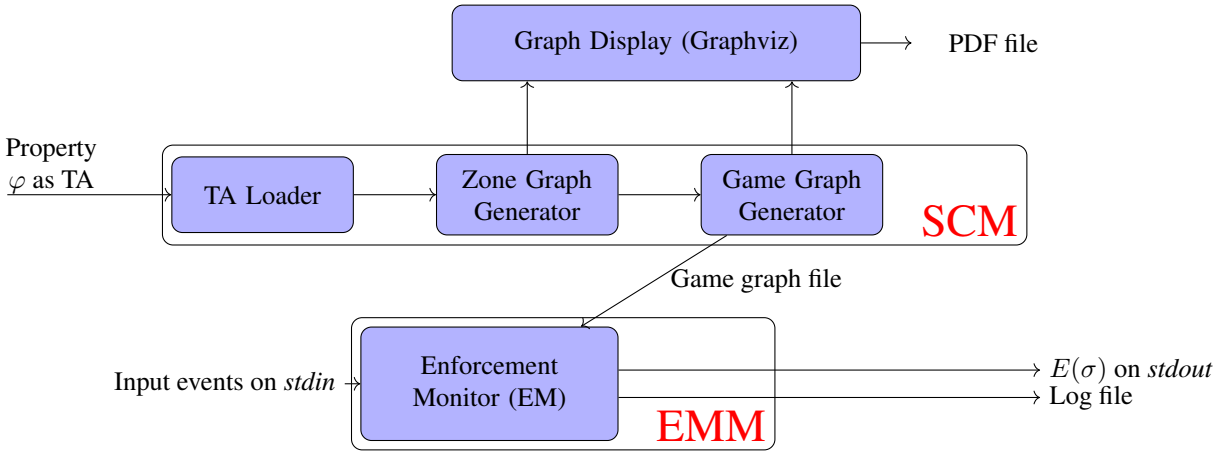


Figure 4.16 – Architecture globale de *GREP*

de la gestion de l'enforcement à la volée durant l'exécution. Pour cela, il utilise le graphe de jeu fourni par le SCM, et retient le nœud courant pendant l'exécution. *GREP* a été conçu dans le but d'être utilisé avec des applications externes : les événements d'entrée sont reçus dans le flux *stdin*, et la sortie de l'*EM* est envoyée dans *stdout*, ce qui permet ainsi de le "brancher" facilement à un système à enforcer. *GREP* fonctionne à la volée durant l'exécution (mode *online*) en sortant les événements au fur et à mesure, mais aussi en mode *offline* permettant de produire une séquence complète en sortie pour une entrée (complète) donnée. Afin d'évaluer les performances de *GREP*, nous l'avons comparé à l'outil *TiPEX* [186], qui à notre connaissance est le seul outil d'enforcement pour propriétés temporisées régulières quelconques. Il est difficilement envisageable de se comparer à d'autres outils comme Polymer [19] ou Mobile [119], car ils se basent sur la théorie de Schneider [200], et ne considèrent donc que des propriétés de safety. De plus, ils sont fortement liés à un langage de programmation, ce qui rend la comparaison encore plus difficile. *TiPEX* ne gère pas les événements incontrôlables, mais à notre connaissance, il n'existe aucun autre outil que *GREP* permettant de les prendre en compte. Nous avons donc comparé *TiPEX* et *GREP* en se basant sur un ensemble de propriétés (uniquement avec des événements contrôlables) fournies avec *TiPEX* (voir [186]) : une propriété de safety, une propriété de co-safety, et une propriété de response. Les résultats de cette étude sont disponibles dans [C25] . Ces derniers montrent notamment que non seulement l'approche par la théorie des jeux est viable dans le cadre d'un *EE*, les temps de réponse étant satisfaisants (généralement de quelques microsecondes sur les exemples étudiés). Sur les propriétés testées, le chargement en mémoire du graphe de jeu ne pose pas de problème. Notons qu'il pourrait être intéressant de tester sur des propriétés ayant un grand nombre d'états, mais rappelons que la philosophie de notre *EM* est d'enforcer des propriétés à haut niveau d'abstraction, et il nous semble peu probable qu'il soit amené à gérer des propriétés complexes, et donc probablement difficilement compréhensibles. D'autre part, les performances obtenues surpassent généralement celles de *TiPEX*.

4.4 Conclusion et perspectives

Ces travaux étant assez récents, il est assez difficile de mesurer leur impact sur la communauté scientifique. Cependant, à notre connaissance, nous avons été les premiers à proposer une approche d'*EE* pour tout type de propriété temporisée régulière (décrite sous forme de TA), et notre approche est généralement (la seule) citée dans les états de l'art du domaine de l'*EE* en ce qui concerne la prise en compte du temps. Notons que ces approches suscitent aussi l'intérêt de certains industriels, comme l'atteste notamment [66], article publié par des membres de Thalès Avionics, expliquant comment ils utilisent le principe de fonction d'enforcement (décrit dans ce chapitre) pour corriger à la volée des erreurs dans un écran tactile de cockpit.

Les perspectives liées à ces travaux sont nombreuses. Nous en listons quelques-unes ci-dessous. Certaines seront développées plus en détail au chapitre 5. Tout d'abord, quel serait l'impact sur le framework si nous décidions de borner la taille des buffers ? Ensuite, que signifie *enforçable* ? Et comment caractériser une telle propriété ? Comment généraliser ces approches, notamment en considérant les différentes règles possibles (suppression d'événement, retard, arrêt, etc...). Comment prendre en compte le non-déterminisme dans le modèle de description ? Peut-on envisager un autre langage de description de propriétés plus adapté aux besoins industriels ? Il reste aussi beaucoup d'améliorations possibles en ce qui concerne l'instrumentation de l'*EE* : même si l'outil *GREP* a été pensé pour s'interfacer avec d'autres applications via les flux *stdin* et *stdout*, son haut niveau d'abstraction nécessite généralement un travail d'instrumentation conséquent pour pouvoir l'utiliser. Enfin, de façon similaire au domaine du MBT vu au chapitre 2, d'autres perspectives sont liées au domaine d'application de l'*EE*. De nombreuses problématiques se présentent notamment dans le cas de systèmes cyber-physiques, systèmes concurrents ou distribués ou systèmes autonomes, ou dans les domaines de la sécurité ou du jeu vidéo. Ces aspects seront développés au chapitre 5.

Chapitre 5

Conclusion générale et perspectives

5.1 Conclusion

Dans ce document, nous avons présenté nos contributions autour du *MBT*, de la vérification de code, et de l'*EE* de propriétés temporisées. Nous y avons notamment décrit nos travaux sur le test de robustesse effectués dans le cadre de la thèse de Fares Saad-Khorchef (thèse dont j'ai assuré le co-encadrement), et en collaboration avec Ismail Berrada, Richard Castanet, Sébastien Salva. Nous avons présenté notre framework de test de robustesse consistant à intégrer de façon incrémentale des aléas à la spécification nominale pour finalement obtenir une spécification augmentée servant de base à la génération des tests. Quelques cas d'études ont aussi été évoqués. Nous avons aussi présenté nos travaux sur le test de systèmes temporisés à flot de données. Ces résultats sont essentiellement issus des travaux de postdoc d'Omer Nguena-Timo (dont j'ai assuré la direction) et d'une collaboration avec Hervé Marchand. Nous avons proposé le modèle *VDTA* pour décrire ce type de systèmes, et discuté la mise en place d'une méthodologie de test adaptée. Nous avons aussi présenté brièvement nos contributions au sujet du test à distance, effectuées dans le cadre du (deuxième) postdoc d'Omer Nguena-Timo (dont j'ai aussi assuré la direction), et issues aussi d'une collaboration avec l'équipe de Kim Larsen au Danemark : Alexandre David, Kim Larsen et Marius Mikucionis. Dans ces travaux, nous avons notamment proposé une sémantique permettant d'intégrer les délais de propagation dans la relation de conformité, et une condition suffisante décidable sur un TA pour garantir l'absence de phénomènes d'entrelacement dus aux problèmes d'observabilité du test. Nous avons ensuite décrit nos travaux autour du BMC effectués principalement lors d'un séjour en CRCT [1] à l'I3S de Sophia Antipolis, et en collaboration avec Hélène Collavizza, Vinh Nguyen, Olivier Ponsini et Michel Rueher. Nous y avons notamment décrit la méthode de recherche en *backjump* permettant d'accélérer la vérification d'assertions dans le code *C* et présenté une étude de cas industrielle temps-réel : le *FlashManager*. Ensuite, nous avons présenté nos travaux concernant l'*EE* de propriétés régulières temporisées décrites sous forme de TA. Ces travaux sont le fruit d'une collaboration avec Yliès Falcone, Thiérry Jéron, Hervé Marchand, Omer Nguena-Timo et Srinivas Pinisetty. Nous avons proposé un framework complet d'*EE*, à différents niveaux d'abstraction, permettant pour une propriété régulière quelconque, de synthétiser l'*EM* associé, et de garantir les propriétés classiques de ce type de mécanisme : correction, transparence, optimalité. Enfin, nous nous sommes intéressés au cas particulier de l'*EE* en présence d'événe-

1. Congé pour Recherches ou Conversions Thématiques.

ments incontrôlables. Ces résultats sont issus de la thèse de Matthieu Renard (dont j'ai assuré la co-direction), et d'une collaboration avec Yliès Falcone, Thiérry Jéron, Hervé Marchand, Omer Nguena-Timo et Srinivas Pinisetty. Nous avons discuté comment adapter nos travaux précédents dans ce nouveau cadre, et proposé un nouveau framework permettant de garantir les propriétés décrites précédemment, en les adaptant si besoin (la transparence est devenue la p-transparence). Enfin, nous avons proposé une modélisation de ce type d'*EE* sous forme de jeu à deux joueurs, permettant de simplifier le problème, et d'obtenir des temps de calcul de l'*EM* globalement meilleurs. L'outil *GREP*, implémentant cette approche, a aussi été décrit.

5.2 Perspectives

Nous allons maintenant donner ci-dessous quelques pistes de recherche en lien avec les travaux décrits précédemment dans ce document. Nous avons déjà proposé quelques perspectives disséminées à l'intérieur des différents chapitres. L'objectif ici est plutôt de présenter quelques thématiques auxquelles nous aimerions nous intéresser par la suite.

Enforcement à l'exécution : quelques aspects théoriques et pratiques. Concernant l'*EE*, il reste un certain nombre de questions théoriques à résoudre. Tout d'abord, nous avons considéré des buffers de taille non bornée (dans le cas temporisé ou non). Il serait donc utile d'étudier la possibilité de borner le buffer de l'*EM* et son impact sur le reste du framework. Sur certains types de propriétés, il semble difficile d'obtenir une borne (exemple d'une propriété de co-safety), mais l'idée serait de caractériser plus finement les propriétés et le lien par rapport au remplissage du buffer. Dans le même esprit, nous avons constaté dans nos travaux que certaines propriétés étaient enforçables et d'autres pas. Cette notion d'*enforçabilité* mériterait d'être précisée, et surtout associée à une caractérisation des propriétés par rapport à ce critère. Ces travaux se feraient dans le cadre d'une généralisation plus globale de la théorie de l'enforcement. En effet, dans nos travaux nous avons considéré des primitives d'enforcement particulières (retardement, arrêt), d'autres comme [3, 100] utilisent la suppression. Il semble donc opportun d'envisager une généralisation, dans laquelle les primitives d'enforcement pourraient être vues comme des paramètres à instancier. Il serait aussi judicieux d'associer à cette généralisation une notion de distance, sur le principe de Bloem et al. [30], permettant de définir une sorte de "scope" dans lequel l'*EM* doit maintenir le *SUS*. Une étude sur la notion de composition et de priorité entre plusieurs *EM* compléterait cette généralisation. Par ailleurs, la description des propriétés pourrait aussi faire l'objet d'investigations. Dans ces travaux, nous avons considéré uniquement des automates (temporisés) déterministes. Le fait d'autoriser des modèles non déterministes permettrait une plus grande abstraction. Il faudrait donc étudier dans quelle mesure nos travaux pourraient s'adapter. D'autres pistes concernant la description des propriétés seraient utiles, par exemple sous forme de logique, e.g. TLTL, ou éventuellement un modèle plus riche. Dans la perspective de développement d'un outil, il serait aussi souhaitable de travailler à la possibilité de décrire des propriétés sous forme de langage simple, compréhensible par un humain non spécialiste de méthodes formelles, un peu à la manière de *Stimulus* [134] qui génère des modèles formels à partir de morceaux de phrases assemblés par l'utilisateur, ou *ParTrap* [29] qui utilise des briques syntaxiques pour décrire des propriétés. Dans ce cas, il faudrait peut-être aussi réfléchir à la définition d'un langage de description "unifié" permettant de traduire d'autres langages vers celui-ci, à la manière de *RSML* dans [110]. Toujours en ce

qui concerne la description de propriétés, nous avons pour l'instant considéré des propriétés simples, décrites par des automates finis. Il serait intéressant d'étudier la possibilité d'ajouter des données dans les modèles, et ainsi d'utiliser des techniques symboliques pour pouvoir enforcer ces propriétés. De plus, comme discuté précédemment, la frontière entre *EE* et la théorie du contrôle [189] est en réalité assez ténue. Il serait donc intéressant d'étudier plus précisément ce qui distingue ces deux domaines, et ainsi voir dans quelle mesure il est possible de réutiliser les techniques de théorie du contrôle pour les appliquer à l'*EE*. Par ailleurs, l'*EE* recouvre aussi bien les aspects théoriques que pratiques. Le framework que nous avons proposé reste à un niveau abstrait. L'outil *GREP* a été conçu pour se "brancher" au *SUS* par les flux *stdin* et *stdout*. Cependant, il est généralement nécessaire d'opérer une phase d'instrumentation pour réellement le brancher sur la plupart des applications. Des efforts restent à faire globalement pour faciliter l'instrumentation, notamment en fournissant un librairie permettant d'interfacer plus facilement l'*EM* et le *SUS*.

Systèmes distribués, systèmes concurrents. Qu'il s'agisse de test, de *RV* ou d'*EE*, les systèmes distribués ou concurrents amènent de nouvelles problématiques, et demeurent un domaine de recherche très actif. Comme discuté au chapitre 2, les problèmes liés à l'observabilité (partielle), à la contrôlabilité, et la recherche d'architecture associée restent des enjeux importants d'investigation pour tester les systèmes distribués. En ce qui concerne le *MBT*, nous pensons que des techniques à base de théorie des jeux seraient prometteuses pour améliorer les résultats actuels, notamment dans le cas temporisé. En s'inspirant de la notion de distance de Henry et al. dans [121], l'idée serait de trouver une stratégie gagnante permettant de maintenir le testeur dans une sorte de zone, qui offre un compromis acceptable entre contrôlabilité et observabilité. Les problématiques d'architecture et d'instrumentation dans le cadre du test, de la *RV*, ou de l'*EE* de systèmes distribués sont assez proches. Si on considère que les testeurs dans une architecture distribuée sont tous passifs, cela revient dans ce cas à un problème de *RV*. Le lecteur trouvera une étude sur les problématiques liées au monitoring de systèmes distribués ainsi qu'une classification des différentes approches associées dans le livre de l'action COST ARVI [97]. Il y est notamment rappelé les enjeux liés à ce type de monitoring, que l'on peut résumer ainsi :

- les systèmes distribués possèdent de nombreuses sources de contrôle [2].
- l'existence de délais de propagation entre les différents nœuds d'un système distribué rend difficile le fait d'identifier un état global du système à tout moment
- le non-déterminisme inhérent à ce type de systèmes
- le fait de monitorer un système distribué peut altérer son comportement global
- la difficulté d'instrumentation (d'un point de vue ingénierie).

Des approches récentes de monitoring décentralisé ont été proposées dans [33, 89], dans lesquelles les moniteurs (ou testeurs passifs) ont une vue partielle du système (distribué), mais ont la possibilité de communiquer entre eux pour émettre un verdict. Les propriétés sont exprimées sous forme de logique temporelle [33] ou d'automate [89]. Il reste cependant pas mal de pistes, comme la prise en compte de fautes dans le modèle, l'observation globale du système (lien entre prédicats locaux observés et prédicats globaux), ou bien l'architecture et notamment les liens entre les différents moniteurs. A notre connaissance, il existe peu de travaux qui s'intéressent à l'*EE* de systèmes distribués. Nous aimerions étudier la possibilité d'étendre les travaux de

2. Traduction sûrement améliorable de *"foci of control"*.

Bonakdarpour et al. [33] dans un cadre d'*EE*. Il faudrait ainsi étudier commet décrire les propriétés à enforcer, quelles informations (minimales) doivent être échangées entre les *EM*, comment synthétiser automatiquement les différents *EM*, quelles primitives d'enforcement il serait raisonnable d'utiliser, et quel type de propriété distribuée il est possible d'enforcer. La prise en compte du temps ainsi que des délais de propagation dans ces modèles permettrait aussi de rendre ce framework plus réaliste. En ce qui concerne plus précisément les systèmes concurrents multithreadés, nous pensons que l'ajout d'un mécanisme d'*EE* et d'une instrumentation adaptée au niveau des *threads* pourrait permettre d'éviter certains phénomènes de *datarace*, voire même d'ordonnancer automatiquement des événements dans le programme pour garantir une propriété globale. L'idée serait de fournir à l'utilisateur une API en apparence similaire à *pthreads*[3] mais permettant de contraindre certaines exécutions pour garantir ces propriétés. Ce principe a déjà fait ses preuves dans [154] dans lequel Luo et Roşu génèrent des *EM* locaux capables de bloquer un thread particulier en cas de risque de violation de propriété dans un programme *Java*.

Vers des problèmes plus concrets, en lien avec les industriels. Les travaux présentés dans ce document sont en majorité des contributions théoriques. Cependant, il nous semble que le fossé apparent entre ces travaux et des applications industrielles réelles n'est pas si grand. Nous pensons donc qu'il serait très utile d'étudier dans quelle mesure une partie de ces contributions serait exportable dans un cadre industriel. Voici quelques pistes dans ce sens. Comme l'illustre [66], les travaux sur l'*EE* semblent intéresser certains industriels. L'idée paraît effectivement séduisante : rajouter après coup un mécanisme permettant de contraindre des comportements d'un système. De notre point de vue, l'utilisation de techniques d'*EE* semble assez prometteuse notamment dans des situations où il est possible d'identifier des situations globales incohérentes sur un système (un peu à la manière des *règles de protection du domaine de vol* dans les avions récents d'Airbus, dont les commandes vont contraindre les évolutions de l'avion dans son domaine de vol). Ces situations sont naturellement assez fréquentes dans les domaines aéronautique, spatial et automobile, et globalement pour tous les systèmes critiques. Ceci étant dit, il reste souvent un gros travail à fournir pour passer de la problématique réelle de l'industriel à la modélisation abstraite fournie par les outils académiques. Il est donc nécessaire globalement de réfléchir à des formalismes de modélisation plus adaptés aux besoins industriels, à la manière de *Stimulus* [134] ou *ParTrap* [29] et de fournir des environnements graphiques permettant de faciliter la saisie de propriétés pour des non spécialistes en méthodes formelles. Ce besoin est valable à la fois en ce qui concerne le test et l'*EE*. Dans la même optique, et toujours dans le domaine du test aussi bien que de l'*EE*, il est nécessaire de fournir des outils permettant de faciliter l'*instrumentation* d'un programme. Par instrumentation, on entend ici tout ce qui permet de passer d'une méthodologie (plus ou moins) abstraite, à sa mise en place concrète dans le cadre d'un banc de test ou d'un réel procédé d'*EE* embarqué. Certains chercheurs considèrent que ce travail est plutôt dévolu aux utilisateurs de la méthodologie proposée, mais la réalité montre qu'en général les partenaires industriels manquent d'expertise pour adapter nos méthodes à leurs cas d'utilisation. Un des enjeux liés à ces travaux, c'est de trouver des solutions pour ne pas tomber dans le cas par cas. L'idée serait donc de mettre au point des techniques les plus génériques possibles, et paramétrables, pour qu'elles puissent ensuite s'instancier dans diverses situations. Cette problématique est très présente globalement dans le domaine de la *RV*, dans lequel on trouve parfois des bibliothèques fournies clés en main

3. Threads sous POSIX.

(e.g. Java-MOP [57]). De façon globale, un de nos objectifs est de mener plus de collaborations avec des partenaires industriels. En effet, lors des divers projets auxquels nous avons participé, ces échanges (notamment avec Dassault Systèmes et EDF) ont toujours été très enrichissants. Usuellement, ces derniers amènent de nouvelles problématiques liées à la validation de leurs applications, qui sont souvent complexes, et de notre côté, nous pouvons apporter notre vision globale du domaine, et la possibilité de consacrer du temps pour résoudre ces problèmes, voire ensuite exporter ces solutions dans d'autres domaines. Ces collaborations sont envisageables notamment par le biais de thèses CIFRE, ou via d'autre projets plus génériques (région, ANR, FUI, etc...).

(Cyber)-sécurité. La sécurité des systèmes informatiques est récemment devenue un enjeu de société majeur, et notamment en ce qui concerne les systèmes de supervision et d'acquisition de données (*SCADA*). Une des pistes intéressantes pour progresser dans ce domaine, c'est sans doute l'apport des méthodes formelles. D'un point de vue *RV* et test, il s'agit d'un domaine spécifique d'application, ouvrant de nombreuses perspectives de recherche. Remarquons qu'une des techniques actuelles les plus populaires de cyber-sécurité est le *test de pénétration*, qui peut être vu comme une variante de test de robustesse décrit au chapitre 2, dans lequel les aléas seraient des éléments spécifiques à la sécurité. Le test de sécurité, y compris à base de modèle, a déjà fait l'objet de nombreuses contributions, dont le lecteur trouvera un résumé très complet (mais un peu ancien) dans [195] et plus récent (et aussi très complet) dans [103]. Il est aussi possible de trouver des travaux sur cette thématique qui se basent sur la vérification à l'exécution (souvent appelés *monitoring* dans ce cadre là). Généralement, ces techniques consistent à vérifier à la volée qu'une propriété donnée est bien respectée lors de l'exécution du *SUS*. Une application possible de nos travaux serait d'utiliser des techniques d'*EE* pour rendre une application plus robuste aux attaques. Ainsi, en plaçant un ou des *EM* de façon stratégique aux points sensibles d'une architecture réseau, et en considérant un ensemble de règles de politique de sécurité comme des propriétés à enforcer (e.g. propriétés de confidentialité, intégrité, disponibilité, authentification, autorisation ou non-répudiation), cela permettrait de garantir certaines propriétés sensibles, en intégrant au *SUS* des mécanismes externes (des *EM*), pouvant être développés bien après la conception initiale du système. Ces règles pourraient être décrites soit sous forme de formules logiques ou d'automates (comme dans [180, 181]), ou par exemple sous forme de patrons de sécurité, à la manière de Obeid et Dhaussy dans [178]. Ces patrons pourraient faire aussi l'objet d'une forme de composition, qu'il serait aussi intéressant d'étudier. Des premières approches prometteuses d'*EE* de propriétés décrites par des automates pour garantir certains aspects de confidentialité dans un navigateur Internet et dans un réseau social ont été discutées dans [180, 181], mais n'ont pas encore été implémentées à notre connaissance, et ne prennent pas en compte les aspects distribués ou distants inhérents aux applications de type *SCADA*.

Mondes virtuels, jeux vidéos. Même si le test en conditions réelles (test *Hardware In the Loop* ou *HIL*) est a priori indispensable, il peut être à la fois difficile et coûteux à mettre en place. Par exemple, on comprend aisément que la mise en situation "réelle" d'une sonde spatiale (quasiment impossible), ou encore d'un drone aérien (potentiellement dangereuse) s'avère compliquée. Il est donc utile de procéder en amont à des tests en simulation (*Software In the Loop* ou *SIL*) afin de détecter au plus tôt le plus de bugs possibles. Cette technique assez ancienne est utilisée depuis longtemps dans le domaine spatial (par exemple par la NASA ou l'Agence

Spatiale Européenne). Cependant, le développement d'environnements ouverts de simulation pour robot, comme *MORSE* [88]) ou *Gazebo* [140] a permis récemment à des laboratoires plus modestes de s'intéresser aussi à cette question du test en simulation. Par exemple, Sotiropoulos et al. proposent dans [206, 207] de tester le système de navigation d'un robot en utilisant une technique de *génération procédurale*, i.e. en l'immergeant virtuellement dans un monde en 3D simplifié généré automatiquement, avec des obstacles aléatoires. Ils définissent une mesure de difficulté de chemin (liée au nombre d'obstacles) et lancent ensuite un grand nombre de simulations en vérifiant à chaque fois si le robot atteint son objectif ou rencontre un problème (erreur de logiciel, collision ou timeout). Ils montrent l'aspect prometteur de cette approche qui a permis de détecter certaines erreurs dans les robots testés. Dans ces travaux, on retrouve finalement des problématiques usuelles du test de logiciel. Il existe un très grand nombre de scénarios de test possibles, qui sont générés aléatoirement. Ainsi, il serait utile de réfléchir à des solutions pour réduire le nombre de données de test tout en obtenant une confiance suffisante. Des techniques par classe d'équivalence permettraient par exemple de conserver un aspect aléatoire dans la génération des tests, tout en réduisant le domaine des données d'entrées, et si besoin, des techniques de type "pairwise", pourraient diminuer un peu le nombre de cas de tests. Une autre piste d'amélioration serait de tenter de modéliser formellement le monde virtuel généré, et ainsi utiliser des techniques de *MBT* pour générer des tests et assurer une couverture suffisante. De plus, le travail nécessaire pour exporter une méthode de test d'un système autonome à un autre est assez fastidieux même quand les deux semblent proches (c'est le cas par exemple dans la thèse de Soutiropoulos [205], dans laquelle deux robots différents sont étudiés). Rendre ces approches plus génériques permettrait de faciliter leur utilisation. Enfin, les expériences actuelles utilisent une échelle de temps réelle. Il serait donc opportun de réfléchir à des possibilités "d'accélérer" le test tout en faisant attention à ne pas l'éloigner trop de la réalité. Par ailleurs, l'arrivée à Bordeaux de l'entreprise Ubisoft est une opportunité pour s'intéresser plus spécifiquement au domaine du jeu vidéo, et pour réfléchir aux apports potentiels des méthodes formelles dans ce domaine. Le principe de *génération procédurale* est aussi utilisé dynamiquement dans certains jeux (dont ceux d'Ubisoft) pour générer à la volée un niveau de jeu, et placer des adversaires ou d'autres personnages dedans. Le placement de ces divers éléments doit respecter des propriétés précises afin de garantir une jouabilité correcte (e.g. placement géographique, déplacement, type de comportement, etc...). Un objectif de recherche serait donc d'étudier la possibilité de décrire ces propriétés à l'aide d'un langage de haut niveau, et de vérifier à l'exécution que ces propriétés sont bien respectées. De façon assez naturelle, l'apport de techniques d'*EE* est aussi envisagé pour contraindre la génération procédurale et s'assurer que les propriétés requises restent vérifiées tout au long de l'exécution du jeu. [4]

Systèmes cyber-physiques (SCP), systèmes autonomes. Ces dernières années, les SCP ont connu un développement très important, notamment dans le domaine médical (on parle de systèmes cyber-physiques médicaux, ou SCPM). Ce sont des systèmes à forte interaction physique avec leur environnement. L'exemple le plus connu de SCPM est le pacemaker, appareil dont l'objectif est de mesurer l'activité cardiaque d'une personne, et d'envoyer un signal électrique si nécessaire pour corriger une arythmie, ou encore le système de contrôle d'une pompe à insuline. Ce type de systèmes est lié à des enjeux vitaux, par conséquent, il est indispensable de le vérifier de façon soigneuse. L'apport des méthodes de vérification statiques permet d'aider à augmenter la confiance envers ces systèmes, cependant leur comportement est généralement

4. Un projet de recherche incluant ces problématiques a été déposé auprès de la région Nouvelle-Aquitaine.

caractérisé par l'évolution de données physiques (généralement continues), ajoutées à des événements discrets, avec des contraintes temps-réel. Par conséquent les modèles utilisés pour les décrire peuvent être parfois compliqués (e.g. des automates hybrides), faisant intervenir par exemple des notions de vitesse ou d'accélération, rendant la vérification difficilement réalisable (voire indécidable [5]). De plus, ce type de systèmes n'est pas (encore) soumis à des normes de développement logiciel strictes qu'il est possible de trouver par exemple en aéronautique. Le point positif, c'est qu'il est généralement assez simple d'accéder aux traces d'exécution de ces systèmes, voire même de les instrumenter, ce qui en fait de bons candidats pour les techniques de *RV*. Le lecteur trouvera une étude sur les problématiques liées au monitoring de SCP et quelques pistes pour les résoudre, notamment en partant de la logique *Signal Temporal Logic (STL)*, dans le livre de l'action COST ARVI [97]. On peut citer aussi une approche récente pour vérifier à l'exécution un SCPM proposée par Blein et al. dans [29], pour analyser les traces (après coup) lors d'actes de chirurgie. Pour compléter ces travaux, il serait intéressant d'étudier dans quelle mesure il serait possible de vérifier ces propriétés à la volée, pour faire remonter des alertes si besoin. Dans le même esprit, il serait utile d'identifier des situations dans lesquelles un SCPM (e.g. un pacemaker) donné ne devrait jamais se trouver (e.g. un délai minimum à respecter entre deux messages d'entrée sur un port donné), d'étudier comment décrire ce type de propriété à l'aide d'un langage adapté, et d'étudier la possibilité d'intégrer un mécanisme d'enforcement pour éviter que cette situation ne se produise. Dans un registre assez similaire, les systèmes autonomes représentent de bons cas d'application pour les techniques de *RV* et d'*EE*. Prenons par exemple le cas d'une voiture autonome, cette dernière va être soumise à un environnement imprévisible, voire hostile, et certaines défaillances peuvent provoquer de graves dommages. Même si les processus de développement de ce genre de systèmes sont généralement soumis à de nombreux procédés de validation, la complexité des situations dynamiques ne permet pas d'envisager une vérification formelle exhaustive, même en utilisant un haut niveau d'abstraction. Ainsi, les méthodes de *RV* permettent d'envisager de monitorer le véhicule et de cibler certaines propriétés de safety (e.g. sortie des configurations admissibles) ou de liveness [6] (progression garantie vers la destination), et de faire remonter des alertes en cas de problème. De plus, les concepteurs essaient souvent de multiplier les mécanismes de sécurité pour les fonctionnalités critiques. C'est dans ce cadre que les méthodes d'*EE* interviennent. Elles permettent de mettre en place de façon additionnelle au système initial des mécanismes de blocage ou de correction en cas de situation considérée comme interdite ou catastrophique (e.g. bras robot qui dépasse une zone de sécurité, angle d'une articulation de robot trop aigu, ou voiture autonome qui franchit une ligne continue). Par exemple, ce type de problème peut survenir lorsqu'une précondition d'un composant n'est en réalité pas respectée (phénomène difficile à mettre en évidence lors des tests d'intégration). Ainsi, ajouter des mécanismes de sécurité revient à identifier un ensemble de propriétés qui peuvent être globales, ou locales à un composant, de voir dans quelle mesure il est possible d'ajouter un ensemble de mécanismes de correction ou d'évitement, et si oui, de synthétiser et d'intégrer ces mécanismes au système. Notons toutefois qu'un *EM* ne devrait effectuer que des actions correctives simples (e.g. stopper le système, freiner, corriger une trajectoire, retarder un événement) et c'est ce qui le rend utilisable de façon efficace. En outre, il est nécessaire d'assurer l'indépendance des différents *EM* intégrés au système, pour éviter des actions correctives contradictoires qui pourraient s'avérer catastrophiques. Ce point n'est pas vraiment évoqué dans la littérature sur le domaine, mais mé-

5. Notons que certains automates hybrides peuvent être vérifiés avec l'outil Hytech [122] ; cette notion de décidabilité des automates hybrides est discutée par Henzinger et al. dans [123].

6. Au sens de Lamport dans sa classification *Safety-Liveness* : "quelque-chose de bon finit par arriver".

riterait d'être étudié. Notons que le LaBRI constitue un bon environnement pour expérimenter nos techniques d'*EE* sur des systèmes autonomes. En effet, nous prévoyons de collaborer avec l'équipe Rhoban qui développe des robots, et d'étudier la possibilité d'intégrer un *EM* dans un premier temps sur un robot "low-cost" de type *Metabot*, puis sur un modèle de cobot humanoïde de type *Ultraban*. Dans les deux cas, il s'agirait dans un premier temps d'intégrer des mécanismes de protection au niveau des articulations pour éviter qu'elles ne cassent.

Bibliographie personnelle

Ma bibliographie personnelle complète et mise à jour régulièrement est disponible à l'adresse : `http://www.labri.fr/perso/rollet/publications.html`

Chapitres de livre :

[B1] Yliès Falcone, Leonardo Mariani, Antoine Rollet and Saikat Saha. Runtime Failure Prevention and Reaction. In *Lectures on Runtime Verification - Introductory and Advanced Topics.* volume 10457 of LNCS, pages 103–134, Springer, 2018

Revues internationales :

[J7] Matthieu Renard, Yliès Falcone, Antoine Rollet, Thierry Jéron and Hervé Marchand. Optimal Enforcement of (Timed) Properties with Uncontrollable Events. *Mathematical Structures in Computer Science (MSCS)* : 1-46 (2017)

[J6] Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, Hervé Marchand, Antoine Rollet and Omer Nguena-Timo. Runtime Enforcement of Timed Properties Revisited. *Formal Methods in System Design (FMSD)* 45(3) : 381-422 (2014)

[J5] Hélène Collavizza, Nguyen Le Vinh, Olivier Ponsini, Michel Rueher and Antoine Rollet. Constraint-based BMC : a Backjumping Strategy. *International Journal on Software Tools for Technology Transfer (STTT)* 16(1) : 103-121 (2014)

[J4] Sébastien Salva and Antoine Rollet. A Pragmatic Approach for Testing Stateless and Stateful Web Service Robustness. *Studia Informatica Universalis*, 10(2) : 139-179 (2012).

[J3] Hacène Fouchal, Antoine Rollet and Abbas Tahrini. Robustness Testing of Composed Real-time Systems. *Journal of Computational Methods in Science and Engineering (JCMSE)*, 10(1) :135-148 (2010).

[J1] Fares Saad Khorchef and Antoine Rollet. A Formal Framework for Robustness Testing of Embedded Systems. *International Journal of Computer and Information Science* (IJCIS), 8(2) :290-299 (2007).

Revues nationales :

[J2] Sébastien Salva and Antoine Rollet. Testabilité des services web. *Ingénierie des Systèmes d'Information RSTI série ISI*, 13(3) :35-58 (2008).

Conférences internationales :

[C25] Matthieu Renard, Antoine Rollet and Yliès Falcone. GREP : Games for the Runtime Enforcement of Properties. In *29th IFIP International Conference on Testing Software and Systems ICTSS 2017*, volume 10533 of LNCS, Springer, pages 259-275, 2017.

[C24] Matthieu Renard, Antoine Rollet and Yliès Falcone. Runtime Enforcement using Büchi Games. In *24th International Symposium on Model Checking Software SPIN 2017*, Co-located with ISSTA 2017, ACM Press, pages 70-79, 2017.

[C23] Matthieu Renard, Yliès Falcone, Antoine Rollet, Srinivas Pinisetty, Thierry Jéron, Hervé Marchand. Enforcement of (Timed) Properties with Uncontrollable Events. In *12th International Colloquium on Theoretical Aspects of Computing ICTAC 2015*, volume 9399 of LNCS, Springer, pages 542-560, 2015.
Selected paper for publication in "Mathematical Structures in Computer Science (MSCS)"

[C22] Alexandre David, Kim G. Larsen, Marius Mikučionis, Omer Nguena-Timo and Antoine Rollet. Remote Testing of Timed Specifications. In *25th IFIP International Conference on Testing Software and Systems ICTSS 13*, volume 8254 of LNCS, Springer, pages 65-81, 2013.

[C21] Srinivas Pinisetty, Yliès Falcone, Thierry Jéron, Hervé Marchand, Antoine Rollet and Omer Nguena-Timo. Runtime Enforcement of Timed Properties. In *Third International Conference on Runtime Verification RV 2012*, volume 7687 of LNCS, Springer-Verlag, pages 229-244, 2012.

[C20] Omer Nguena-Timo and Antoine Rollet. A Zone-based Reachability Analysis of Variable Driven Timed Automata. In *3rd International Conference on Advances in System Testing and Validation Lifecycle VALID 2011*, pages 51-57, 2011.
Best Paper Award.

[C19] Sébastien Salva and Antoine Rollet. Automatic Web Service Testing from Wsdl Descriptions. In *11th International Conference on Innovative Internet Community Services I2CS 2011*, volume 186 of Lecture Notes in Informatics, GI, pages 217-226, 2011.

[C18] Omer Nguena-Timo and Antoine Rollet. Test Selection for Data-flow Reactive Systems Based on Observations. In *7th Workshop on Advances in Model Based Testing A-MOST 2011*, IEEE, pages 1-8, 2011.

[C17] Omer Nguena-Timo, Hervé Marchand and Antoine Rollet. Automatic Test Generation for Data-flow Reactive Systems with Time Constraints. In *22nd IFIP International Conference on Testing Software and Systems ICTSS 2010, (ex Testcom/Fates)*, pages 25-30, 2010. Short Paper, long version available at `https://hal.archives-ouvertes.fr/hal-00503000`.

[C16] Fares Saad-Khorchef, Ismail Berrada, Antoine Rollet and Richard Castanet. Automated Robustness Testing for Reactive Systems : Application to Communicating Protocols. In *10th International Conference on Innovative Internet Community Services I2CS, Jubilee Edition 2010*, volume 165 of *Lecture Notes in Informatics*, pages 409-421. GI, 2010.

[C15] Omer Nguena Timo and Antoine Rollet. Conformance Testing of Variable Driven Automata. In *8th IEEE International Workshop on Factory Communication Systems Communication in Automation WFCS 2010*, pages 241-248, 2010.

[C14] Sébastien Salva and Antoine Rollet. Test purpose generation for timed protocol testing. In *2nd International Conference on Communication Theory, Reliability, and Quality of Service CTRQ 2009,* pages 8-14, France, July 2009.

[C13] Antoine Rollet and Sébastien Salva. Testing Robustness of Communicating Systems using Ioco-based Approach. In *1st IEEE Workshop on Performance evaluation of communications in distributed systems and Web based service architectures, in conjunction*

with IEEE ISCC 2009, pages 67-72, 2009.

[C12] Hacène Fouchal, Antoine Rollet and Abbas Tarhini. Robustness Testing on Composed Timed Systems. In *18th International Conference on Software Engineering and Data Engineering SEDE 2009*, pages 161-167, 2009.
Selected paper for publication in Journal of Computational Methods in Science and Engineering (JCMSE)

[C11] Antoine Rollet and Fares Saad-Khorchef. A Formal Approach to Test the Robustness of Embedded Systems using Behaviour Analysis. In *5th International Conference on Software Engineering Research, Management and Applications-TOC SERA 2007*, pages 667-674, 2007.
Selected paper for publication in IJCIS international journal

[C10] Fares Saad-Khorchef, Antoine Rollet and Richard Castanet. A Framework and a Tool for Robustness Testing of Communicating Software. In *ACM International Symposium on Applied Computing SAC 2007*, pages 1461-1466, 2007.

[C9] Fares Saad Khorchef, Ismail Berrada, Antoine Rollet and Richard Castanet. Cadre Formel pour le Test de Robustesse. Application au Protocole SSL. In *Colloque Francophone sur l'Ingénierie des Protocoles - CFIP 2006*, Hermès, pages 191-202, 2006.

[C8] Hacène Fouchal, Antoine Rollet and Abbas Tarhini. Robustness of Composed Timed Systems. In *31st Annual Conference on Current Trends in Theory and Practice of Informatics SOFSEM 2005*, volume 3381 of LNCS, Springer, pages 155-164, 2005.

[C7] Abbas Tarhini, Antoine Rollet and Hacène 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 2005*, pages 143-149, 2005.

[C6] Hacène Fouchal, Laurent Pierre, Sébastien Gruson, Cyril Rabat and Antoine Rollet. Integrated Tool for Testing Timed Systems. In *Fifth IEEE International Symposium and School on Advance Distributed Systems ISSADS 2005*, volume 3563 of LNCS, Springer, pages 153-166, 2005.

[C5] Hacène Fouchal, Cyril Rabat, Antoine Rollet and Abbas Tarhini. Experimental Results on Testing Real-time Systems. In *13th International Conference on Intelligent and Adaptive Systems and Software Engineering IASSE 2004*, pages 284-289, 2004.

[C4] Hacène Fouchal and Antoine Rollet. A Simple Testing Technique for Embedded Systems. In *7th International Conference on Principles of Distributed Systems OPODIS 2003*, volume 3144 of LNCS, Springer, pages 159-170, 2003.

[C3] Antoine Rollet and Hacène Fouchal. Testing Protocol Robustness. In *Innovative Internet Community Systems I2CS 2003*, volume 2877 of LNCS, Springer, pages 201-215, 2003.

[C2] Antoine Rollet. Testing Robustness of Real-time Embedded Systems. In *Workshop On Testing Real-Time and Embedded Systems WTRTES, Satellite Workshop of Formal Methods FM 2003 Symposium*, pages 63–74, 2003.

[C1] Sébastien Salva, Antoine Rollet and Hacène Fouchal. Temporal and Behavior Characterization of States in Timed Systems. In *Annual International Conference on Computer and Information Science, ICIS 2002*, pages 687–692, 2002.

Ecoles jeunes chercheurs :

[I3] Antoine Rollet. Model Based Testing : Principes et Applications dans le Cadre Temporisé. In *Ecole d'Eté Temps Réel ETR 2011*, Brest, France, pages 63–76, August 2011.

[I2] Antoine Rollet. Tutorial on testing techniques : a research point of view. In *12th TAROT Summer School 2016 on Software Testing, Verification and Validation*, Paris, France, July, 2016.

Papiers invités :

[I1] Antoine Rollet and Sébastien Salva. Two Complementary Approaches to Test Robustness of Reactive Systems. In *IEEE international conference on Automation, Quality and Testing, Robotics, AQTR 2008*, pages 47-53, 2008.
Invited paper.

Bibliographie

[1] UML 2.0 testing profile, v1.0. Technical report, Object Management Group, July 2005.

[2] S. Abramsky. Observation equivalence as a testing equivalence. *Theoretical Computer Science*, 53(2) :225 – 241, 1987.

[3] L. Aceto, I. Cassar, A. Francalanza, and A. Ingólfsdóttir. On runtime enforcement via suppressions. In *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*, pages 34 :1–34 :17, 2018.

[4] S. B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, C-27(6) :509–516, 1978.

[5] B. Alcalde, A. Cavalli, D. Chen, D. Khuu, and D. Lee. Network protocol system passive testing for fault management : A backward checking approach. In *International Conference on Formal Techniques for Networked and Distributed Systems*, pages 150–166. Springer, 2004.

[6] R. Alur and D. Dill. A theory of timed automata. *Theoretical Comput. Sci.*, 126 :183–235, 1994.

[7] R. Anido and A. Cavalli. *Guaranteeing full fault coverage for UIO-based testing methods*, pages 215–231. Springer US, 1996.

[8] A. Anier. *Model Based Framework for Distributed Control and Testing of Cyber-Physical Systems*. PhD thesis, Tallinn University of Technology, 2016.

[9] A. Anier, J. Vain, and L. Tsiopoulos. Dtron : a tool for distributed model-based testing of time critical applications. *Proceedings of the Estonian Academy of Sciences*, 66(1) :75, 2017.

[10] A. Arnold, G. Point, A. Griffault, and A. Rauzy. The Altarica formalism for describing concurrent systems. *Fundamenta Informaticae*, 40(2, 3) :109–124, 1999.

[11] G. Audemard and L. Simon. On the Glucose SAT solver. *International Journal on Artificial Intelligence Tools*, 27(01) :1840001, 2018.

[12] Y. Bar-Hillel, M. Perles, and E. Shamir. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14 :143–172, 1961.

[13] R. Barbuti and L. Tesei. Timed automata with urgent transitions. *Acta Informatica*, 40(5) :317–347, 2004.

[14] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. *Information Processing Letters*, 93(6) :281–288, 2005.

[15] E. Bartocci, Y. Falcone, B. Bonakdarpour, C. Colombo, N. Decker, K. Havelund, Y. Joshi, F. Klaedtke, R. Milewicz, G. Reger, G. Rosu, J. Signoles, D. Thoma, E. Zalinescu, and Y. Zhang. First international competition on runtime verification : rules, benchmarks, tools, and final results of CRV 2014. *International Journal on Software Tools for Technology Transfer*, 2017.

[16] D. Basin, V. Jugé, F. Klaedtke, and E. Zălinescu. Enforceable security policies revisited. *ACM Trans. Inf. Syst. Secur.*, 16(1) :3 :1–3 :26, June 2013.

[17] D. Basin, F. Klaedtke, and E. Zalinescu. Algorithms for monitoring real-time properties. In S. Khurshid and K. Sen, editors, *Proceedings of the 2nd International Conference on Runtime Verification (RV 2011)*, volume 7186 of *LNCS*, pages 260–275. Springer-Verlag, 2011.

[18] P. Baudin, J.-C. Filliâtre, T. Hubert, C. Marché, B. Monate, Y. Moy, and V. Prevosto. ACSL : ANSI/ISO C Specification Language. preliminary design, version 1.4, 2008.

[19] L. Bauer, J. Ligatti, and D. Walker. Composing security policies with Polymer. In V. Sarkar and M. W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 305–314. ACM, 2005.

[20] G. Behrmann, A. David, and K. G. Larsen. *A Tutorial on Uppaal*, pages 200–236. Springer Berlin Heidelberg, 2004.

[21] M. Bekkouche, H. Collavizza, and M. Rueher. Locfaults : A new flow-driven and constraint-based error localization approach. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, SAC '15, pages 1773–1780. ACM, 2015.

[22] J. Bengtsson and W. Yi. Timed automata : Semantics, algorithms and tools. *LNCS*, 3098 :87–124, 2004.

[23] N. Benharrat, C. Gaston, R. M. Hierons, A. Lapitre, and P. Le Gall. Constraint-based oracles for timed distributed systems. In N. Yevtushenko, A. R. Cavalli, and H. Yenigün, editors, *Testing Software and Systems*, pages 276–292. Springer International Publishing, 2017.

[24] I. Berrada, R. Castanet, and P. Félix. Testing communicating systems : a model, a methodology, and a tool. In F. Khendek and R. Dssouli, editors, *Testing of Communicating Systems*, pages 111–128. Springer Berlin Heidelberg, 2005.

[25] G. Berry and G. Gonthier. The Esterel synchronous programming language : Design, semantics, implementation. *Science of computer programming*, 19(2) :87–152, 1992.

[26] N. Bertrand, T. Jéron, A. Stainer, and M. Krichen. Off-line test selection with test purposes for non-deterministic timed automata. In P. A. Abdulla and K. R. M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 96–111. Springer Berlin Heidelberg, 2011.

[27] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu, et al. Bounded model checking. *Advances in computers*, 58(11) :117–148, 2003.

[28] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *TACAS*, pages 193–207. Springer, 1999.

[29] Y. Blein, Y. Ledru, L. du Bousquet, and R. Groz. Extending specification patterns for verification of parametric traces. In *FormaliSE 2018*. ACM, 2018.

[30] R. Bloem, B. Könighofer, R. Könighofer, and C. Wang. Shield synthesis : - runtime enforcement for reactive systems. In *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 533–548, 2015.

[31] G. V. Bochmann and A. Petrenko. Protocol testing : Review of methods and relevance for software testing. In *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '94, pages 109–124. ACM, 1994.

[32] H. Bohnenkamp and A. Belinfante. Timed testing with TorX. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM 2005 : Formal Methods*, pages 173–188. Springer Berlin Heidelberg, 2005.

[33] B. Bonakdarpour, P. Fraigniaud, S. Rajsbaum, D. A. Rosenblueth, and C. Travers. Decentralized Asynchronous Crash-Resilient Runtime Verification. In J. Desharnais and R. Jagadeesan, editors, *27th International Conference on Concurrency Theory (CONCUR 2016)*, volume 59 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16 :1–16 :15. Schloss Dagstuhl–Leibniz-Zentrum fur Informatik, 2016.

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

[35] M. Bozga, J.-C. Fernandez, L. Ghirvu, S. Graf, J.-P. Krimm, and L. Mounier. IF : An intermediate representation and validation environment for timed asynchronous systems. In *Proc. of FM '99*, pages 307–327. Springer, 1999.

[36] J. Bradley and N. Davies. Analysis of the SSL protocol. Technical Report CSTR-95-021, Department of Computer Science, University of Bristol, 1995.

[37] E. Brinksma and J. Tretmans. *Testing Transition Systems : An Annotated Bibliography*, pages 187–195. Springer Berlin Heidelberg, 2001.

[38] E. Brinskma. A theory for the derivation of tests. *Proc. 8th Int. Conf. Protocol Specification,Testing and Verification*, pages 63–74, 1988.

[39] 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.

[40] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*, volume 3472 of *LNCS*. Springer Verlag, 2005.

[41] J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking : 1020 states and beyond. *Information and Computation*, 98(2) :142 – 170, 1992.

[42] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of jml tools and applications. *International journal on software tools for technology transfer*, 7(3) :212–232, 2005.

[43] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe : Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2) :10 :1–10 :38, Dec. 2008.

[44] D. Cansell and D. Méry. *The event-B Modelling Method : Concepts and Case Studies*, pages 47–152. Springer Berlin Heidelberg, 2008.

[45] D. Cansell and D. Méry. Foundations of the B method. *Computing and informatics*, 22(3-4) :221–256, 2012.

[46] D. Cansell, D. Méry, and J. Rehm. Time constraint patterns for event B development. In J. Julliand and O. Kouchnarenko, editors, *B 2007 : Formal Specification and Development in B*, pages 140–154. Springer Berlin Heidelberg, 2006.

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

[48] F. Cassez, A. David, K. G. Larsen, D. Lime, and J.-F. Raskin. Timed control with observation based and stuttering invariant strategies. In *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis*, volume 4762 of *LNCS*, pages 192–206. Springer, 2007.

[49] R. Castanet and H. Waeselynck. Techniques avancées de test de systèmes complexes : Test de robustesse. Technical report, Action spécifique 23 du CNRS, 2003.

[50] K. C. Castillos, H. Waeselynck, and V. Wiels. Show me new counterexamples : A path-based approach. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–10, 2015.

[51] A. Cavalli, C. Gervy, and S. Prokopenko. New approaches for passive testing using an extended finite state machine specification. *Information and Software Technology*, 45(12) :837–852, 2003.

[52] A. Cavalli, S. Maag, and E. M. De Oca. A passive conformance testing approach for a manet routing protocol. In *Proceedings of the 2009 ACM Symposium on Applied Computing*, SAC '09, pages 207–211. ACM, 2009.

[53] E. Chang, Z. Manna, and A. Pnueli. The Safety-Progress Classification. Technical report, Stanford University, Dept. of Computer Science, 1992.

[54] H. Charafeddine, K. El-Harake, Y. Falcone, and M. Jaber. Runtime enforcement for component-based systems. In R. L. Wainwright, J. M. Corchado, A. Bechini, and J. Hong, editors, *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pages 1789–1796. ACM, 2015.

[55] X. Che, F. L. Rojas, and S. Maag. A logic-based passive testing approach for the validation of communicating protocols. In *ENASE'12 : 7th International Conference on Evaluation of Novel Approaches to Software Engineering*, 2012.

[56] F. Chen, M. d'Amorim, and G. Roşu. Checking and correcting behaviors of Java programs at runtime with Java-MOP. *Electronic Notes in Theoretical Computer Science*, 144(4) :3–20, 2006.

[57] F. Chen and G. Roşu. Java-MOP : A monitoring oriented programming environment for Java. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 546–550. Springer, 2005.

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

[59] H.-N. Chu, J. Arlat, M.-O. Killijian, B. Lussier, and D. Powell. Robustness Testing of Robot Controller Software. In H. Waeselynck, editor, *12th European Workshop on Dependable Computing, EWDC 2009*, page 2 pages, 2009.

[60] W. Chung and P. Amer. Improved on UIO Sequence Generation and Partial UIO Sequences. In R. Linn and M. Uyar, editors, *Protocol Specification, Testing, and Verification, XII, Lake Buena Vista, Florida, USA*, June 1992.

[61] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV : a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4) :410–425, Mar 2000.

[62] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG : A symbolic test generation tool. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 470–475, 2002.

[63] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer Berlin Heidelberg, 2004.

[64] E. M. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT press, 1999.

[65] M. Clatin, R. Groz, M. Phalippou, and R. Thummel. *Two approaches linking a test generation tool with verification techniques*, pages 151–166. Springer US, 1996.

[66] P. Coni, J. Berthon, J.-N. Perbet, Y. Sontag, J. C. Abadie, C. Dubau, and C. Balihaut. P-166 : Touchscreen system architecture for safety critical applications. In *SID Symposium Digest of Technical Papers*, volume 45, pages 1600–1603. Wiley Online Library, 2014.

[67] L. Cordeiro and B. Fischer. Verifying multi-threaded software using SMT-based context-bounded model checking. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 331–340. ACM, 2011.

[68] L. Cordeiro, P. Kesseli, D. Kroening, P. Schrammel, and M. Trtik. JBMC : A bounded model checking tool for verifying Java bytecode. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification*, pages 183–190. Springer International Publishing, 2018.

[69] O. Coudert and J. C. Madre. *A Unified Framework for the Formal Verification of Sequential Circuits*, pages 39–50. Springer US, 2003.

[70] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[71] C. Csallner and Y. Smaragdakis. Jcrasher : an automatic robustness tester for Java. *Software : Practice and Experience*, 34(11) :1025–1050, 2004.

[72] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C. In G. Eleftherakis, M. Hinchey, and M. Holcombe, editors, *Software Engineering and Formal Methods*, pages 233–247. Springer Berlin Heidelberg, 2012.

[73] F. Cuppens, N. Cuppens-Boulahia, and T. Sans. Nomad : A security model with non atomic actions and deadlines. In *Computer Security Foundations, 2005. CSFW-18 2005. 18th IEEE Workshop*, pages 186–196. IEEE, 2005.

[74] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4) :451–490, 1991.

[75] A. David, K. G. Larsen, S. Li, and B. Nielsen. A game-theoretic approach to real-time system testing. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '08, pages 486–491. ACM, 2008.

[76] A. David, K. G. Larsen, S. Li, and B. Nielsen. Timed testing under partial observability. In *2nd IEEE International Conference on Software Testing, Verification, and Validation*, pages 61–70. IEEE Computer Society, 2009.

[77] H. P. De León, S. Haar, and D. Longuet. Conformance relations for labeled event structures. In *Tests and Proofs*, pages 83–98. Springer, 2012.

[78] L. de Moura and N. Bjørner. Z3 : An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer Berlin Heidelberg, 2008.

[79] J. DeVale, P. Koopman, and D. Guttendorf. The Ballista software robustness testing service. In *Testing Computer Software Conference (TCSC99)*, June 1999.

[80] T. Dierks and C. Allen. Rfc 2246. *The TLS protocol, version*, 1, 1999.

[81] E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3) :174–186, 1968.

[82] E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8) :453–457, 1975.

[83] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, pages 197–212. Springer Berlin Heidelberg, 1990.

[84] R. Dorofeeva, K. El-Fakih, S. Maag, A. R. Cavalli, and N. Yevtushenko. FSM-based conformance testing methods : A survey annotated with experimental evaluation. *Information and Software Technology*, 52(12) :1286 – 1297, 2010.

[85] V. D'Silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7) :1165–1178, 2008.

[86] R. Dssouli, A. Khoumsi, M. Elqortobi, and J. Bentahar. Testing the control-flow, data-flow, and time aspects of communication systems : A survey. In *Advances in Computers*, volume 107, pages 95–155. Elsevier, 2017.

[87] B. Dutertre. Yices 2.2. In A. Biere and R. Bloem, editors, *Computer Aided Verification*, pages 737–744. Springer International Publishing, 2014.

[88] G. Echeverria, N. Lassabe, A. Degroote, and S. Lemaignan. Modular open robots simulation engine : Morse. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 46–51, 2011.

[89] A. El-Hokayem and Y. Falcone. Monitoring decentralized specifications. In T. Bultan and K. Sen, editors, *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 125–135. ACM, 2017.

[90] J. Eloff and M. B. Bella. *Software Failures : An Overview*, pages 7–24. Springer International Publishing, 2018.

[91] E. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3) :241 – 266, 1982.

[92] A. EnNouaary, R. Dssouli, and F. Khendek. Timed Wp-method : Testing real-time systems. *IEEE Transactions on Software Engineering (TSE)*, 28(11) :1023–1038, 2002.

[93] D. Estrin, S. Gupta, and A. Helmy. Stress : Systematic testing of robustness by evaluation of synthesized scenarios, 1998.

[94] F.-B.-F. F1211008Z. Homere : Hardware trOjans : Menaces et robustEsse des ciRcuits intEgrés.

[95] Y. Falcone. *Etude et mise en œuvre de techniques de validation à l'exécution*. PhD thesis, Université Joseph Fourier, Grenoble 1, 2009.

[96] Y. Falcone. You should better enforce than verify. In H. Barringer, Y. Falcone, B. Finkbeiner, K. Havelund, I. Lee, G. J. Pace, G. Rosu, O. Sokolsky, and N. Tillmann, editors, *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *LNCS*, pages 89–105. Springer, 2010.

[97] Y. Falcone and E. Bartocci, editors. *Lectures on Runtime Verification*. LNCS. Springer, Cham, 2018.

[98] Y. Falcone, J.-C. Fernandez, and L. Mounier. What can you verify and enforce at runtime ? *International Journal on Software Tools for Technology Transfer*, 14(3) :349–382, 2012.

[99] Y. Falcone, K. Havelund, and G. Reger. A tutorial on runtime verification. *Engineering Dependable Software Systems*, 34 :141–175, 2013.

[100] Y. Falcone, T. Jéron, H. Marchand, and S. Pinisetty. Runtime enforcement of regular timed properties by suppressing and delaying events. *Systems & Control Letters*, 123 :2–41, 2016.

[101] Y. Falcone, L. Mounier, J. Fernandez, and J. Richier. Runtime enforcement monitors : composition, synthesis, and enforcement abilities. *Formal Methods in System Design*, 38(3) :223–262, 2011.

[102] S. Falke, F. Merz, and C. Sinz. The bounded model checker LLBMC. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 706–709. IEEE, 2013.

[103] M. Felderer, M. Büchler, M. Johns, A. D. Brucker, R. Breu, and A. Pretschner. Security testing : A survey. In A. Memon, editor, *Advances in Computers*, volume 101, pages 1 – 51. Elsevier, 2016.

[104] J.-C. Fernandez, L. Mounier, and C. Pachon. A model-based approach for robustness testing. In F. Khendek and R. Dssouli, editors, *Testing of Communicating Systems*, pages 333–348. Springer Berlin Heidelberg, 2005.

[105] J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *Programming Languages and Systems*, pages 125–128. Springer Berlin Heidelberg, 2013.

[106] R. W. Floyd. Assigning meanings to programs. *Mathematical aspects of computer science*, 19(19-32) :1, 1967.

[107] H. Fouchal, E. Petitjean, and S. Salva. Testing timed systems with timed purposes. In *Proceedings Seventh International Conference on Real-Time Computing Systems and Applications*, pages 166–171, 2000.

[108] L. Frantzen, J. Tretmans, and R. de Vries. Towards model-based testing of web services. In A. Bertolino and A. Polini, editors, *In Proceedings of International Workshop on Web Services Modeling and Testing (WS-MaTe2006)*, pages 67–82, 2006.

[109] 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.

[110] F. Galinie, S. Ebersold, and J.-M. Bruel. Requirements specific modeling language : un langage formel d'expression d'exigences. In *7ème Conférence en IngénieriE du Logiciel - CIEL2018 Conference*, 2018.

[111] M. K. Ganai and A. Gupta. Accelerating high-level bounded model checking. In *Proceedings of the 2006 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '06, pages 794–801. ACM, 2006.

[112] C. Gaston, R. M. Hierons, and P. Le Gall. An implementation relation and test framework for timed distributed systems. In H. Yenigün, C. Yilmaz, and A. Ulrich, editors, *Testing Software and Systems*, pages 82–97. Springer Berlin Heidelberg, 2013.

[113] P. Godefroid, N. Klarlund, and K. Sen. DART : Directed Automated Random Testing. *SIGPLAN Not.*, 40(6) :213–223, 2005.

[114] G. Gonenc. A method for the design of fault detection experiment. *IEEE transactions on Computers*, C-19 :551–558, 1970.

[115] J. Grabowski, D. Hogrefe, and R. Nahm. *A method for the generation of test cases based on SDL and MSCs*. Universität Bern. Institut für Informatik und Angewandte Mathematik, 1993.

[116] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock. An introduction to the testing and test control notation (TTCN-3). *Computer Networks*, 42(3) :375–403, 2003.

[117] W. Grieskamp. Microsoft's protocol documentation program : A success story for model-based testing. In L. Bottaci and G. Fraser, editors, *Testing – Practice and Research Techniques*, pages 7–7. Springer Berlin Heidelberg, 2010.

[118] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, 1991.

[119] K. W. Hamlen, G. Morrisett, and F. B. Schneider. Certified in-lined reference monitoring on .NET. In V. C. Sreedhar and S. Zdancewic, editors, *Proceedings of the 2006 Workshop on Programming Languages and Analysis for Security, PLAS 2006, Ottawa, Ontario, Canada, June 10, 2006*, pages 7–16. ACM, 2006.

[120] O. Henniger. On test case generation from asynchronously communicating state machines. In M. Kim, S. Kang, and K. Hong, editors, *Testing of Communicating Systems*, IFIP - The International Federation for Information Processing, pages 255–271. Springer, 1997.

[121] L. Henry, T. Jéron, and N. Markey. Control strategies for off-line testing of timed systems. In M. d. M. Gallardo and P. Merino, editors, *Model Checking Software*, pages 171–189. Springer International Publishing, 2018.

[122] T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. Hytech : A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1-2) :110–122, 1997.

[123] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata ? In *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, STOC '95, pages 373–382. ACM, 1995.

[124] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and computation*, 111(2) :193–244, 1994.

[125] A. Hessel, K. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. Testing real-time systems using Uppaal. In R. Hierons, J. Bowen, and M. Harman, editors, *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 77–117. Springer Berlin / Heidelberg, 2008.

[126] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2) :9 :1–9 :76, Feb. 2009.

[127] R. M. Hierons, M. G. Merayo, and M. Núnez. Using time to add order to distributed testing. In *International Symposium on Formal Methods*, pages 232–246. Springer, 2012.

[128] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10) :576–580, 1969.

[129] C. Hutchison, M. Zizyte, P. E. Lanigan, D. Guttendorf, M. Wagner, C. L. Goues, and P. Koopman. Robustness testing of autonomy software. In *Proceedings of the 40th International Conference on Software Engineering : Software Engineering in Practice*, ICSE-SEIP '18, pages 276–285. ACM, 2018.

[130] IBM. Telelogic tau. `http://www-01.ibm.com/support/docview.wss?uid=swg21380572`.

[131] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-soft : Software verification platform. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification*, pages 301–306. Springer Berlin Heidelberg, 2005.

[132] C. Jard and T. Jéron. TGV : theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer*, 7(4) :297–315, Aug 2005.

[133] C. Jard, T. Jéron, L. Tanguy, and C. Viho. Remote testing can be as powerful as local testing. *Formal Desciption Techniques and Protocol Specification, Testing and Verification FORTE XI/PSTV XVIII*, 99, 1999.

[134] B. Jeannet and F. Gaucher. Debugging embedded systems requirements with Stimulus : an automotive case-study. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, 2016.

[135] A. S. Kalaji, R. M. Hierons, and S. Swift. Generating feasible transition paths for testing from an extended finite state machine (EFSM). In *2009 International Conference on Software Testing Verification and Validation*, pages 230–239, 2009.

[136] A. Kerbrat and I. Ober. Automated test generation from SDL/UML specifications. In *12th International Software Quality Week*, 1999.

[137] A. Khoumsi. A temporal approach for testing distributed systems. *IEEE Transactions on Software Engineering*, 28(11) :1085–1103, 2002.

[138] A. Khoumsi, A. EnNouaary, R. Dssouli, and M. Akalay. A new method for testing real time systems. In *Proceedings Seventh International Conference on Real-Time Computing Systems and Applications*, pages 441–450, 2000.

[139] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C : A software analysis perspective. *Formal Aspects of Computing*, 27(3) :573–609, 2015.

[140] N. P. Koenig and A. Howard. Design and use paradigms for Gazebo, an open-source multi-robot simulator. In *IROS*, volume 4, pages 2149–2154, 2004.

[141] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In *Model Checking Software, 11th International SPIN Workshop, Barcelona, Spain, April 1-3, 2004*, volume 2989 of *LNCS*, pages 109–126. Springer, 2004.

[142] R. Langerak. A testing theory for LOTOS using deadlock detection. In *Proceedings of the IFIP WG6.1 Ninth International Symposium on Protocol Specification, Testing and Verification IX*, pages 87–98, 1990.

[143] K. G. Larsen, M. Mikucionis, B. Nielsen, and A. Skou. Testing real-time embedded software using Uppaal-Tron : An industrial case study. In *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, pages 299–306. ACM, 2005.

[144] K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International journal on software tools for technology transfer*, 1(1-2) :134–152, 1997.

[145] J.-C. Léchenet, N. Kosmatov, and P. Le Gall. Cut branches before looking for bugs : certifiably sound verification on relaxed slices. *Formal Aspects of Computing*, 30(1) :107–131, 2018.

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

[147] B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In L.-H. Eriksson and P. A. Lindsay, editors, *FME 2002 :Formal Methods—Getting IT Right*, pages 21–40, 2002.

[148] M. Leucker and C. Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5) :293–303, 2009.

[149] J. Ligatti, L. Bauer, and D. Walker. Enforcing non-safety security policies with program monitors. In S. D. C. di Vimercati, P. F. Syverson, and D. Gollmann, editors, *Computer Security - ESORICS 2005, 10th European Symposium on Research in Computer Security, Milan, Italy, September 12-14, 2005, Proceedings*, volume 3679 of *LNCS*, pages 355–373. Springer, 2005.

[150] J. Ligatti, L. Bauer, and D. Walker. Run-time enforcement of nonsafety policies. *ACM Trans. Inf. Syst. Secur.*, 12(3) :19 :1–19 :41, Jan. 2009.

[151] J. A. Ligatti. *Policy enforcement via program monitoring*. PhD thesis, Princeton University, 2006.

[152] D. Lugato, C. Bigot, and Y. Valot. Validation and automatic test generation on UML models : the AGATHA approach. *Electronic Notes in Theoretical Computer Science*, 66(2) :33 – 49, 2002. FMICS'02, 7th International ERCIM Workshop in Formal Methods for Industrial Critical Systems (ICALP 2002 Satellite Workshop).

[153] G. Luo, A. Petrenko, and G. v. Bochmann. *Selecting Test Sequences for Partially-Specified Nondeterministic Finite State Machines*, pages 95–110. Springer US, 1995.

[154] Q. Luo and G. Roşu. EnforceMOP : A runtime property enforcement system for multithreaded programs. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 156–166. ACM, 2013.

[155] M. Machin, F. Dufossé, J.-P. Blanquart, J. Guiochet, D. Powell, and H. Waeselynck. Specifying safety monitors for autonomous systems using model-checking. In A. Bondavalli and F. Di Giandomenico, editors, *Computer Safety, Reliability, and Security*, pages 262–277. Springer International Publishing, 2014.

[156] M. Machin, F. Dufossé, J. Guiochet, D. Powell, M. Roy, and H. Waeselynck. Model-checking and game theory for synthesis of safety rules. In *High Assurance Systems Engineering (HASE), 2015 IEEE 16th International Symposium on*, pages 36–43. IEEE, 2015.

[157] B. Marre and A. Arnould. Test sequences generation from LUSTRE descriptions : GATEL. In *Proceedings ASE 2000. Fifteenth IEEE International Conference on Automated Software Engineering*, pages 229–237, Sept 2000.

[158] F. Martinell and I. Matteucci. Through modeling to synthesis of security automata. *Electronic Notes in Theoretical Computer Science*, 179 :31–46, 2007.

[159] K. L. McMillan. *Symbolic Model Checking*, pages 25–60. Springer US, 1993.

[160] G. H. Mealy. A method for synthesizing sequential circuits. *Bell System Technical Journal*, 34(5) :1045–1079, 1955.

[161] B. Meyer. *Object-oriented software construction*, volume 2. Prentice hall New York, 1988.

[162] L. Michel and P. V. Hentenryck. The Comet programming language and system. In P. van Beek, editor, *Proceedings of the 11th International Conference on Principles and Practice of Constraint Programming (CP'05)*, volume 3709 of *LNCS*, pages 881–881. Springer, 2005.

[163] Z. Micskei, H. Madeira, A. Avritzer, I. Majzik, M. Vieira, and N. Antunes. Robustness testing techniques and tools. In K. Wolter, A. Avritzer, M. Vieira, and A. van Moorsel, editors, *Resilience Assessment and Evaluation of Computing Systems*, pages 323–339. Springer Berlin Heidelberg, 2012.

[164] Z. Micskei, Z. Szatmári, J. Oláh, and I. Majzik. A concept for testing robustness and safety of the context-aware behaviour of autonomous systems. In G. Jezic, M. Kusek, N.-T. Nguyen, R. J. Howlett, and L. C. Jain, editors, *Agent and Multi-Agent Systems. Technologies and Applications*, pages 504–513. Springer Berlin Heidelberg, 2012.

[165] Z. Micskei and H. Waeselynck. The many meanings of UML 2 sequence diagrams : a survey. *Software & Systems Modeling*, 10(4) :489–514, 2011.

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

[167] B. P. Miller, D. Koski, C. P. Lee, V. Maganty, R. Murthy, A. Natarajan, and J. Steidl. Fuzz revisited : A re-examination of the reliability of unix utilities and services. Technical report, Technical Report CS-TR-1995-1268, University of Wisconsin, 1995.

[168] M. T. Mnad, C. Deleuze, and I. Parissis. Synchronous programs testing language (SPTL). In B. Murgante, S. Misra, A. M. A. C. Rocha, C. Torre, J. G. Rocha, M. I. Falcão, D. Taniar, B. O. Apduhan, and O. Gervasi, editors, *Computational Science and Its Applications – ICCSA 2014*, pages 683–695. Springer International Publishing, 2014.

[169] M. T. Mnad, C. Deleuze, I. Parissis, J. Launay, and J. B. Gning. Automated test generation for synchronous controllers. In *2016 IEEE/ACM 11th International Workshop in Automation of Software Test (AST)*, pages 1–7, 2016.

[170] R. Moraes, H. Waeselynck, and J. Guiochet. UML-based modeling of robustness testing. In *2014 IEEE 15th International Symposium on High-Assurance Systems Engineering*, pages 168–175, 2014.

[171] G. Morales, S. Maag, A. Cavalli, W. Mallouli, E. M. De Oca, and B. Wehbi. Timed extended invariants for the passive testing of web services. In *2010 IEEE International Conference on Web Services (ICWS)*, pages 592–599. IEEE, 2010.

[172] S. Musacchio. Le nanosatellite Picsat ne répond plus. *CNRS Le Journal*, april 2018.

[173] S. Naito and M. Tsunoyama. Fault detection for sequential machines by transition-tours. *Proceedings of Fault Tolerant Computer Systems*, pages 238–243, 1981.

[174] R. D. Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1) :83 – 133, 1984.

[175] B. Nielsen and A. Skou. Automated Test Generation from Timed Automata. In T. Margaria and W. Yi, editors, *Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, volume 2031 of *LNCS*, pages 343–357. Springer, april 2001.

[176] N. Noroozi, R. Khosravi, M. R. Mousavi, and T. A. Willemse. Synchronizing asynchronous conformance testing. In *Software Engineering and Formal Methods*, pages 334–349. Springer, 2011.

[177] M. Núñez and I. Rodríguez. Conformance testing relations for timed systems. In *Formal Approaches to Software Testing, 5th International Workshop, FATES 2005, Edinburgh, UK, July 11, 2005, Revised Selected Papers*, volume 3997 of *LNCS*, pages 103–117. Springer, 2005.

[178] F. Obeid and P. Dhaussy. Validation formelle d'architectures logicielles basée sur des patrons de sécurité. In A. Rollet and A. Lanoix, editors, *Approches Formelles dans l'Assistance au Développement de Logiciels - AFADL'18*, pages 35–40, 2018.

[179] I. S. G. of Software Engineering Terminology 610.12-1990. Customer and terminology standards. *In IEEE Standards Software Engineering, IEEE Press*, 1, 1999.

[180] G. J. Pace, R. Pardo, and G. Schneider. On the runtime enforcement of evolving privacy policies in online social networks. In T. Margaria and B. Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation : Discussion, Dissemination, Applications*, pages 407–412. Springer International Publishing, 2016.

[181] R. Pardo, P. Picazo-Sanchez, G. Schneider, and J. Tapiador. A runtime monitoring system to secure browser extensions. *Security Principles and Trust Hotspot 2017*, 2017.

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

[183] A. Petrenko, S. Boroday, and R. Groz. Confirming configurations in EFSM testing. *IEEE Transactions on Software Engineering*, 30(1) :29–42, 2004.

[184] M. Phalippou. *Relation d'implantation et hypothèses de test sur des automates à entrées et sorties*. PhD thesis, Univ. of Bordeaux, Sept. 1994.

[185] S. Pinisetty, Y. Falcone, T. Jéron, and H. Marchand. Runtime enforcement of parametric timed properties with practical applications. In J. Lesage, J. Faure, J. E. R. Cury, and B. Lennartson, editors, *12th International Workshop on Discrete Event Systems, WODES 2014, Cachan, France, May 14-16, 2014.*, pages 420–427. International Federation of Automatic Control, 2014.

[186] S. Pinisetty, Y. Falcone, T. Jéron, and H. Marchand. Tipex : a tool chain for timed property enforcement during execution. In *Runtime Verification*, pages 306–320. Springer, 2015.

[187] J. Postel. Transmission Control Protocol specification. *RFC 793*, 1981.

[188] D. Powell, J. Arlat, H. N. Chu, F. Ingrand, and M. O. Killijian. Testing the input timing robustness of real-time control software for autonomous systems. In *2012 Ninth European Dependable Computing Conference*, pages 73–83, 2012.

[189] P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1) :81–98, 1989.

[190] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE transactions on software engineering*, SE-11(4) :367–375, 1985.

[191] P. Raymond, X. Nicollin, N. Halbwachs, and D. Waber. Automatic testing of reactive systems. In *Proceedings of the 1998 IEEE Real-Time Systems Symposium, RTSS'98, Madrid, Spain*, pages 200–209. IEEE Computer Society Press, December 1998.

[192] M. Renard. GREP. `https://github.com/matthieurenard/GREP`, 2017.

[193] M. Renard. *Runtime Enforcement of (Timed) Properties with Uncontrollable Events*. PhD thesis, Université de Bordeaux, 2017.

[194] A. Reynolds and V. Kuncak. Induction for SMT solvers. In D. D'Souza, A. Lal, and K. G. Larsen, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 80–98. Springer Berlin Heidelberg, 2015.

[195] J.-L. Richier. Security testing techniques. Technical report, DIAMONDS Consortium, 2011. Deliverable ID : D1.WP2.

[196] M. Rodríguez, F. Salles, J.-C. Fabre, and J. Arlat. Mafalda : Microkernel assessment by fault injection and design aid. In J. Hlavička, E. Maehle, and A. Pataricza, editors, *Dependable Computing — EDCC-3*, pages 143–160. Springer Berlin Heidelberg, 1999.

[197] F. Saad-Khorchef. *Un cadre formel pour le test de robustesse des protocoles de communication*. PhD thesis, Université Bordeaux 1, 2006.

[198] K. Sabnani and A. Dahbura. A protocol test generation procedure. *Computer Networks and ISDN Systems*, 15 :285–297, 1988.

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

[200] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1) :30–50, Feb. 2000.

[201] B. Seljimi and I. Parissis. Using CLP to automatically generate test sequences for synchronous programs with numeric inputs and outputs. In *ISSRE '06 : Proceedings of the 17th International Symposium on Software Reliability Engineering*, pages 105–116. IEEE Computer Society, 2006.

[202] K. Sen, D. Marinov, and G. Agha. Cute : a concolic unit testing engine for C. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 263–272. ACM, 2005.

[203] A. Shahrokni and R. Feldt. A systematic review of software robustness. *Information and Software Technology*, 55(1) :1 – 17, 2013. Special section : Best papers from the 2nd International Symposium on Search Based Software Engineering 2010.

[204] A. Simao and A. Petrenko. Generating asynchronous test cases from test purposes. *Information and Software Technology*, 53(11) :1252–1262, 2011.

[205] T. Sotiropoulos. *Test aléatoire de la navigation de robots dans des mondes virtuels*. PhD thesis, Université Toulouse 3 Paul Sabatier, 2018.

[206] T. Sotiropoulos, J. Guiochet, F. Ingrand, and H. Waeselynck. Virtual worlds for testing robot navigation : A study on the difficulty level. In *2016 12th European Dependable Computing Conference (EDCC)*, pages 153–160, 2016.

[207] T. Sotiropoulos, H. Waeselynck, J. Guiochet, and F. Ingrand. Can robot navigation bugs be found in simulation ? an exploratory study. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, pages 150–159, 2017.

[208] J. Springintveld, F. Vaandrager, and P. R. D'Argenio. Timed Testing Automata. *Theoretical Comput. Sci.*, 254(254) :225–257, 2001.

[209] R. S. Streett. Propositional dynamic logic of looping and converse. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 375–383. ACM, 1981.

[210] J. Tretmans. Testing labelled transition systems with inputs and outputs. In *8th International Workshop on Protocols Test Systems, Evry, France*, 1995.

[211] J. Tretmans. Test generation with inputs, outputs, and repetitive quiescence. *Software-Concepts and Tools*, 17 :103–120, 1996.

[212] J. Tretmans and E. Brinksma. TorX : Automated model based testing. In *First European Conference on Model-Driven Software Engineering*, 2003.

[213] 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.

[214] H. Waeselynck, Z. Micskei, N. Rivière, Á. Hamvas, and I. Nitu. Termos : a formal language for scenarios in mobile computing systems. In *International Conference on Mobile and Ubiquitous Systems : Computing, Networking, and Services*, pages 285–296. Springer, 2010.

[215] M. T. B. Waez, J. Dingel, and K. Rudie. A survey of timed automata for the development of real-time systems. *Computer Science Review*, 9 :1 – 26, 2013.

[216] C. J. Wang and M. T. Liu. Generating test cases for EFSM with given fault models. In *INFOCOM '93, 12th Annual Joint Conference of the IEEE Computer and Communications Societies. Networking : Foundation for the Future, IEEE*, pages 774–781 vol.2, 1993.

[217] Wikipedia. List of software bugs. `https://en.wikipedia.org/wiki/List_of_software_bugs`. vérifié le 19 juin 2018.

[218] M. Wu, H. Zeng, and C. Wang. Synthesizing runtime enforcer of safety properties under burst error. In S. Rayadurgam and O. Tkachuk, editors, *NASA Formal Methods - 8th International Symposium, NFM 2016, Minneapolis, MN, USA, June 7-9, 2016, Proceedings*, volume 9690 of *LNCS*, pages 65–81. Springer, 2016.

[219] M. Yannakakis and D. Lee. An efficient algorithm for minimizing real-time transition systems. In *International Conference on Computer Aided Verification*, pages 210–224. Springer, 1993.

[220] I.-T. R. Z.140-142. The testing and test control notation", version 3 (TTCN-3), Rec. Z.140 : TTCN-3 core language, Rec. Z.141 : Tabular presentation format for TTCN-3 (TFT), Rec. Z.142 : Graphical presentation format for TTCN-3 (GFT). ITU-T, Geneva (Switzerland), 2002.