Environnement de test pour un système temps-réel de performance en musique mixte

Clément Poncelet Sanchez

▶ To cite this version:

Clément Poncelet Sanchez. Environnement de test pour un système temps-réel de performance en musique mixte. Autre [cs.OH]. 2013. hal-00920028

HAL Id: hal-00920028
https://inria.hal.science/hal-00920028

Submitted on 17 Dec 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

hal-00868618, version 1 - 1 Oct 2013

Environnement de test pour un système temps-réel de performance en musique mixte

Clement Poncelet Sanchez
clement.poncelet@gmail.com

1er septembre 2013

Université Pierre et Marie Curie
Mémoire de Master STL-APR

Résumé

Les systèmes temps-réel ne sont pas nouveaux en informatique et sont généralement développés dans les secteurs critiques : - avionique, automobile, ferroviaire ... - Ils sont de plus soumis à de lourdes responsabilités c'est pourquoi ils ont été presque directement concernés par les tests. Ce document est un mémoire de fin de Master2 Informatique STL de l' Université Pierre et Marie Curie. Il s'est déroulé à l'IRCAM (Institut de Recherche et coordination Acoustique/Musique) où la recherche informatique est très présente. Ici, nous allons aborder le sujet du test sur un système temps réel embarqué. Les tests en temps réel ne sont pas récents mais la génération automatique de tests basée sur modèle pour systèmes temps-réel est un problème relativement nouveau. Une grande partie de cette littérature porte donc sur des systèmes non temporisés. - La spécification par automates temporisés, l'exécution des tests grâce à des outils du logiciel UPPAAL, la gestion d'une conformité pour rendre le bon verdict selon un environnement en temps réel - sont des points étudiés lors de ce stage. Avant de les aborder et d'énoncer les contributions en détail, un récapitulatif des notions utilisées et du contexte (notamment du système Anstecofo) est fait.

hal-00868618, version 1 - 1 Oct 2013

Table des matières

I Introduction **4**

1 Contexte général **4**
1.1 IRCAM 4
1.2 Musique Mixte 5
1.3 Chronologie 6
1.3.1 MAX 7
1.3.2 Antescofo un système musical interactif 8
1.4 État de l'art 9
1.5 Sujet de Stage 11

II Existant **14**

2 Antescofo : Langage haut niveau **14**
2.1 Syntaxe 14
2.2 Tempi, synchronisation et erreurs 15
2.3 ...et d'autres instructions 17

3 Langage "bas niveau" **18**
3.1 Les Automates à entrées/sorties 18
3.2 Automates temporisés à entrées/sorties 21

4 Traduction d'automate **23**

III Contribution **24**

5 Principes de compilation **24**
5.1 Vision temporisée d'UPPAAL 25

6 Le temps **27**

7 Génération de test Offline **29**
7.1 coVer 30
7.2 Utilisation 31

8 Génération de test Online **33**
8.1 Tron 34
8.2 Utilisation de Tron 35
8.3 Version Ad'Hoc 36

IV Conclusion **37**

9 Bilan **37**
9.1 Avantages et plus 37
9.2 Difficultés 38

10 Conclusion **39**

V Remerciements **40**

VI annexes **40**

hal-00868618, version 1 - 1 Oct 2013

hal-00868618, version 1 - 1 Oct 2013

Première partie

Introduction

De nombreux domaines utilisent l'informatique, aussi éloignés qu'ils puissent être. Les croisements de ces domaines avec l'informatique apportent de nouveaux points de vues qui s'en concluent par de nouvelles avancées, aussi bien dans ces domaines que dans l'informatique. C'est sur un de ces croisements que ce sujet de stage porte, lorsque deux grands domaines en plein essor se croisent. Le premier, culturel mais pas moins lié à l'informatique : *la musique mixte*. Cette musique allie des sons d'instruments traditionnels et des sons électroacoustiques. Valorisant le temps avant tout, la musique mixte est sans cesse en développement, nécessitant des logiciels de plus en plus complexes pour innover et satisfaire les attentes des compositeurs. Le second, la génération de tests fondée sur modèle, qui concerne tout logiciel temps-réel complexe voulant prouver sa valeur et fiabilité. Ce croisement, qui est de faire une génération de test temps réel sur un système de musique mixte embarqué, pointe plusieurs sujets de recherche récents et est un excellent moyen de développer cette recherche tout en testant un logiciel utilisé.

Nous allons commencer **Partie 1** par présenter le contexte de ce stage, c'est à dire l'IRCAM et son système temps réel que nous devons tester (Antescofo) ainsi que la problématique et la méthode générale de test choisie. Nous étudierons ensuite, **Partie 2**, les réflexions et les faits accomplis avant le commencement de mon stage. Nous parlerons enfin **partie 3** de la contribution au sujet effectuée durant ce stage et de la manière dont on a procédé à nos tests. Pour finir la dernière partie est consacrée à la conclusion et énonce les points positifs et négatifs rencontrés pendant le déroulement des recherches.

1 Contexte général

C'est à l'IRCAM que ce stage a été réalisé : Un emplacement idéal pour étudier des environnements de test sur un système temps-réel de performance en musique mixte. Mais qu'est ce que l'IRCAM ? Et ce système testé ? Le contexte général va présenter brièvement ces deux mots pour introduire clairement où et dans quel but cette recherche est traitée. Il présentera ensuite un historique de la musique mixte pour décrire Antescofo, le système cible de nos tests. Après cette description nous aborderons un état de l'art sur les tests et les automates pour mixer les deux sujets et parler de la problématique et de notre solution.

1.1 IRCAM

L'Institut de Recherche et Coordination Acoustique/Musique est l'un des plus grands centres de recherche publique au monde se consacrant à la création musicale et à la recherche scientifique. Fondé en 1977 par Pierre Boulez, l'institut est dirigé depuis

hal-00868618, version 1 - 1 Oct 2013

FIGURE 1: url : http ://www.ircam.fr/

2006 par Frank Madlener et réunit plus de cent soixante collaborateurs. Ses trois axes principaux : création, recherche et transmission, sont développés au cours d'une saison parisienne, d'un festival annuel, de tournées en France et à l'étranger. L'icram est associé au Centre Pompidou sous la tutelle du ministère de la Culture et de la Communication. Depuis 1995, le ministère de la Culture et de la Communication, l'ircam et le CNRS sont associés dans le cadre d'une unité mixte de recherche STMS (Science et technologies de la musique et du son - UMR 9912) rejoint en 2010 par l'université Pierre et Marie Curie (UPMC).

C'est sur Antescofo, un logiciel développé par l'IRCAM que le sujet du stage porte. Avec l'équipe INRIA de développement et de recherche MUTANT, unifiant le CNRS, l'INRIA et l'IRCAM, le problème de tester ce logiciel distribué en temps réel s'est pausé. C'est dans cette équipe que j'ai effectué ce stage et aidé mon encadrant, Florent Jacquemard, chercheur à l'INRIA.

1.2 Musique Mixte

Avant d'introduire Antescofo il nous faut commencer par parler de musique mixte. La musique mixte est une musique alliant des sons instrumentaux (générés par des instruments traditionnels) et des sons électroacoustiques (diffusés par des haut-parleurs), elle se définit donc lorsqu'une pièce de musique comporte une partie ou des effets joués par un système électronique. C'est à dire que parmi les musiciens se trouve des systèmes, comme un ordinateur, et que ceux-ci vont jouer en suivant la partition de musique mixte (comme un musicien supplémentaire). Cela implique des **interactions** entre ces systèmes électroniques et les musiciens.

Écrire une partition, revient à poser sur papier **la performance idéale** pensée par un compositeur. Cette performance est humainement irréalisable et englobe **plusieurs performances** possibles pouvant toutes être considérées comme juste. Deux points de vue diffèrent dans la musique entre la composition où l'on crée la partition, et la performance oú on **la réalise en temps réel**. Cela ajoute au contexte, en plus des aspects

temps réel de la performance, **embarqué** de l'interaction système-musiciens, un aspect d'**unité de temps**. En effet une partition ne comprend que du **temps relatif**, que l'on appel des 'pulsations' (beats), hors c'est bien en **temps absolu** (seconde) qu'un musicien joue cette même partition.

1.3 Chronologie

Antescofo est la suite d'une longue histoire, celle de la musique mixte. Cette histoire s'est principalement déroulée à l'IRCAM où a été créé Antescofo. La connaître est primordiale pour comprendre le contexte, l'évolution de la musique mixte et le système Antescofo. Cette rapide chronologie présente quelques étapes de la musique mixte et se termine par la présentation de ce nouveau système.

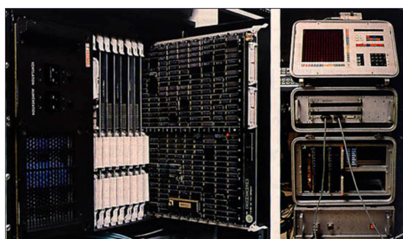
Avant 1950s : Enregistrement sur bandes magnétiques. La première œuvre mixte pour instrument et dispositif électroacoustique est souvent attribuée au compositeur *Bruno Maderna* pour sa pièce ≪ *Musica su due dimensioni* ≫ *pour flûte et bande* (1952). L'idée étant de ramener les dispositifs électroacoustiques sur scène et de les considérer en ***interaction*** avec des instruments acoustiques.

Année 1970s : L'Ircam a, depuis sa création en 1977, travaillé sur le temps réel, c'est-à-dire sur des dispositifs matériels et informatiques permettant la création et le travail du son numérique sans délai de calcul perceptible. Parmi les compositeurs intégrant ces technologies dans leurs œuvres, le premier à avoir exploré de manière systématique les possibilités du temps réel est sans doute *Philippe Manoury*.

Année 1980s-90s : Des machines nécessaires à l'électronique temps réel voient le jour, elles sont volumineuses et les logiciels développés pour les utiliser récents. Voici en exemple la pièce **Pluton**, de Manoury, Lippe et Puckette pour piano et électronique temps-réel (live electronics). Elle tournait sur le **4x** avec un environnement de programmation peu confortable, comme montré images 2.
Ses œuvres mises à part, *Philippe Manoury* est surtout connu dans le monde de l'informatique musicale pour sa contribution, dès 1988 (aux côtés de l'informaticien Miller Puckette), au développement du logiciel ***Max***, dont le nom est un hommage à Max Matthews, le père de l'informatique musicale (décédé en avril 2011).

Maintenant : Les logiciels se sont améliorés aussi bien dans la performance qu'au niveau ergonomique et tournent sur des ordinateurs portables ou de bureau. Voici, image 3 le logiciel MAX utilisé pour la pièce **Hist Wist** en 2009 de Stroppa et Cont.

hal-00868618, version 1 - 1 Oct 2013



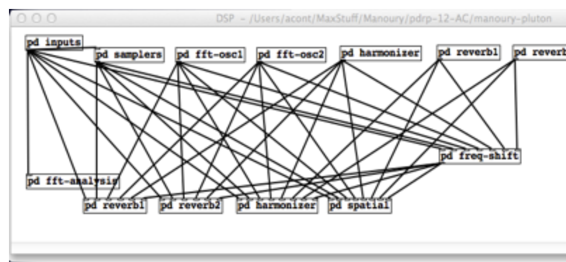


FIGURE 2: Musique-mixte-1980-90 (Pluton)

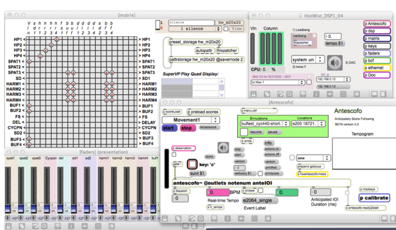


FIGURE 3: Musique-mixte-2009 (Hist-Wist)

1.3.1 MAX

MAX/MSP est un logiciel musical dédié à la synthèse sonore, l'analyse, l'enregistrement et le contrôle d'instrument MIDI (Musical Instrument Digital Interface) en temps réel. Développé par l'IRCAM dans les années 1980 il est l'un des logiciels musicaux les plus utilisés aussi bien par les musiciens professionnels qu'amateurs (il est notamment pratique pour les performances). C'est le fruit d'une association de deux logiciels :

MAX : Un logiciel de calculs mathématiques permettant, par extension, de contrôler en temps réel les instruments MIDI

MSP : Une bibliothèque de fonction qui, ajoutée à MAX, permet de travailler en temps réel avec le signal audio (DSP)

Cet environnement de programmation graphique fonctionne par le langage des 'patchs' que l'on associe visuellement. L'on peut, plus tard, rendre nos patchs indépendants de MAX (on l'appelle la version *stand alone* du patch).

Pure Data : Est aussi un langage de programmation graphique et permet de faire des programmes sans toucher une ligne de code. Il est utilisé pour générer du son, des vidéos, des graphiques 2D/3D, des interfaces de senseur, des entrés de système, et du MIDI.

hal-00868618, version 1 - 1 Oct 2013

1.3.2 Antescofo un système musical interactif

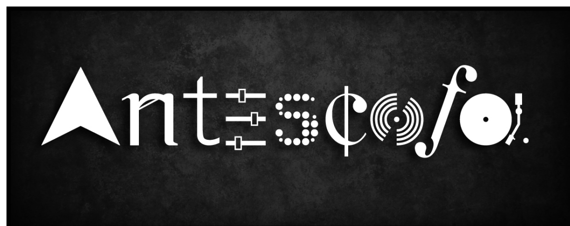


Figure 4: url : http ://repmus.ircam.fr/antescofo
http ://forumnet.ircam.fr/product/antescofo/

Antescofo est un système modulaire polyphonique, suiveur de partition et peut être vu comme un langage synchrone pour composition musicale. Il permet une reconnaissance automatique du tempo et de la position sur partition d'un flux audio en temps réel venant d'un ou de musicien(s). Il synchronise de plus une performance instrumentale avec des éléments réalisés par ordinateur. Il se veut 'plug and play' et facilement utilisable.

Antescofo est né d'une collaboration entre un chercheur (Arshia Cont), un compositeur (Marco Stroppa) et un saxophoniste (Claude Delangle) pour son premier mouvement ... de *Silence* en 2007. Depuis cette date de nombreux compositeurs et musiciens informatiques ont augmenté les rangs, toujours actifs, d'Antescofo : Pierre Boulez, Philippe Manoury, Gilbert Nouno, Serge Lemouton, Larry Nelson et bien d'autres ... Ce système essaie d'étendre le paradigme de synchronisation et le suivi de partition, il devient un véritable outil pour l'écriture du temps et l'interaction musicale en informatique (aussi bien pour la composition que pour la performance musicale), et n'a pas fini d'évoluer...

Fonctionnement : Antescofo est un musicien électronique et a besoin au préalable d'une partition (de musique mixte) en entrée. Il se comporte ensuite comme un humain en performance et pour une partition courante, écoute les musiciens qui l'accompagnent pour 'jouer' sa partie électronique. Cette partition spécifie le comportement d'Antescofo attendu par le compositeur selon toutes performances possibles.

But et Architecture : Antescofo fût développé en **C++** comme un module externe pour MAX/MSP et des environnements de programmation comme PureData. Il représente une avancée dans la musique mixte en ajoutant la notion de **temps relatif** (le temps en 'pulsations' : croches, noires, blanches...). Cette notion permet le **calcul du tempo** et est importante pour la réaction d'Antescofo selon une 'performance' spécifique d'un musicien en temps réel. De nombreux traitements supplémentaires sont

hal-00868618, version 1 - 1 Oct 2013

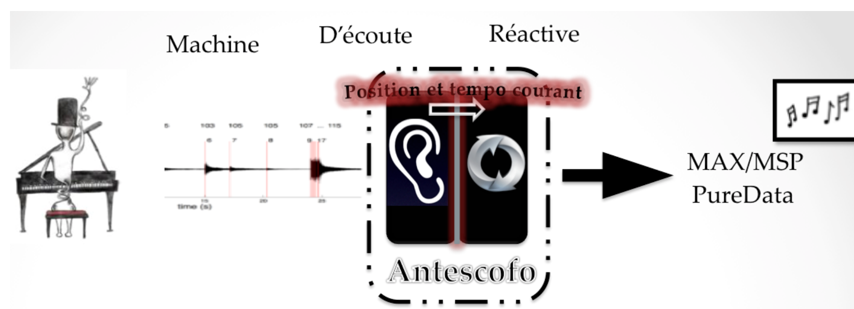


FIGURE 5: Architecture d'Antescofo

possibles et décrits dans la section **II.2** présentant le langage utilisé pour écrire une partition mixte.
Nous pouvons voir (image 5) l'architecture d'Antescofo divisée en deux modules : **une machine d'écoute**, se focalisant sur le flux audio qui arrive en entrée (l'oreille du musicien). Ce module est chargé de **détecter** la position (dans la partition) et le tempo courants du musicien nécessaire au deuxième module. Ce second module, **la machine réactive**, 'joue' selon la partition courante et les informations de la machine d'écoute, la partie électronique. En réalité Antescofo ne joue pas mais se charge **d'envoyer au bon moment des 'messages'** en sortie vers un environnement électronique, ne contraignant pas une compatibilité en sortie.

1.4 État de l'art

Concentrons nous maintenant sur le domaine de la génération de test basée sur modèle. Dans ce domaine les automates ont fait leur preuve et sont très efficaces pour décrire des systèmes complexes ainsi que pour les modéliser rigoureusement. Plus précisément ce sont les Automates temporisés qui ont été choisi pour spécifier formellement Antescofo, nous allons faire ici un état de l'art sur quelques papiers de la littérature scientifique concernant la génération de test discret basée sur modèle et la spécificité qu'a le temps réel ***à donner des durées en sortie*** :

Une description formelle des automates à entrées/sorties labellisés (IOLTS) ainsi qu'une méthode de génération de test basée sur ces modèles sont présentées par une étude d'Antoine Rollet ([Rol11]). Cette méthode est utilisée pour tester en 'boite noire' des systèmes réactifs, c'est-à-dire, interagissant avec leur environnement. Elle présente également une description pour les automates temporisés et leur relation de conformité (tioco) pour une spécification non bloquante et une implémentation non bloquante et complète en entrées. L'idée étant que toutes sorties (y compris l'écoulement du temps) possibles en un état de l'**IUT** (*Implantation sous test*) après avoir observé une certaine séquence sur le modèle doivent être autorisées par ce modèle (i.e spécifiées).

hal-00868618, version 1 - 1 Oct 2013

hal-00868618, version 1 - 1 Oct 2013

Comme abordé plus haut, Antescofo a une relation primordiale avec le temps. Les automates temporisés ont plusieurs relations de conformité vis-à-vis d'une IUT. Elles sont formellement décrites dans la littérature, selon la définition donnée pour l'écoulement du temps et le traitement de l'environnement extérieur. Ce papier [JS08] définit et compare les différentes formes de cette relation appelée *tioco*.

FIGURE 6: url : http ://www.uppaal.org/

UPPAAL est un logiciel de test de modèle (model checking) pour automates temporisés, développé par les universités d'Uppsala (Suisse) et d'Aalborg (Danemark). C'est un outil fournissant un environnement (graphique) pour modéliser, valider et vérifier les systèmes temps-réels spécifiés comme des réseaux d'automates temporisés étendus par des types de données.

Uppaal fournit une étude ([AHS08]) sur les automates temporisés où les auteurs insistent fortement sur le principe d'environnement et de temps pour l'exécution de tests temporisés. Selon eux, un système temps réel ne doit pas seulement rendre une réponse avant un certain temps mais doit la donner au bon moment (ni après ni avant). Pour ces systèmes le temps de réaction est aussi important que le type de réaction. Cette étude présente en particulier deux points de vues distincts de génération de tests pour des systèmes temps-réel en utilisant UPPAAL (ces points de vues vont être expliqués ci-après). En revanche l'exécution des tests sur une IUT possède le même principe, qui est d'appliquer un cas ou une suite de test sur le système sous test. Ce cas de test étant une trace qui comporte une séquence alternative d'actions (entrées sorties) et de délais.

Offline : Dans cette approche la suite de tests est pré-calculée depuis la spécification, puis exécutée sur l'implantation sous test (IUT). Ces suites de tests peuvent être automatiquement (re)-générées même pour un changement de modèle. Les cas de tests sont simples, à bas prix et sont exécutés rapidement car la résolution des contraintes de temps a été faite lors du pré-calcul. Pendant ce pré-calcul, il est possible d'ajouter des critères de couvertures pour augmenter la fiabilité des suites de test générées automatiquement. Le désavantage est qu'il est nécessaire d'analyser entièrement le modèle (explosion d'états), cela implique le déterminisme du modèle de spécification et de l'implantation. Les contributions apportées par cette méthode sont données section **III.7**

note : Des travaux ont été fait dans le but de supprimer cette condition de déterminisme, comme ceux de Nathalie Bertrand [NB12].

Online : Cette deuxième approche consiste à combiner la génération et l'exécution des cas de tests, on l'appelle aussi test 'à la volée' (on-the-fly). Le générateur de test

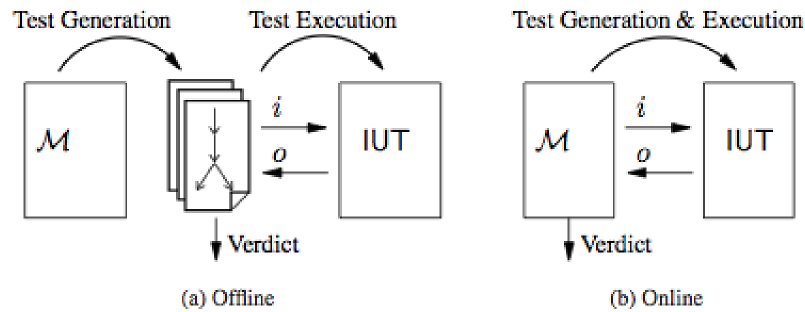


FIGURE 7: Génération de tests Offline et Online

interprète le modèle puis simule et observe directement l'IUT. Les cas de tests s'opèrent un par un, pendant cette exécution le générateur va choisir aléatoirement les entrées et délais puis vérifier sur le modèle les sorties (si existantes) de l'IUT. Ces tests peuvent facilement durer longtemps (des heures ou des jours) et trouver ainsi le cas pointant un dysfonctionnement. Il n'y a plus de problème d'explosion d'état et la spécification et l'implantation peuvent être non-déterministes. Le problème est que la génération et la vérification en temps réel peuvent demander beaucoup de puissance de calcul, un test trop long peut également être difficile à diagnostiquer (bien que des guides sont possibles pour la génération de tests). Les contributions apportées par cette méthode sont données section **III.8**

1.5 Sujet de Stage

Ce sujet : Environnement de test pour un système temps-réel de performance en musique mixte, consiste à appliquer les techniques de génération de test en temps réel aux systèmes de musique mixte. Innovant dans le domaine des automates temporisés et apportant des tests pour le système Antescofo. Cette nouveauté s'explique par la notion de **temps relatif** et **de calcul de tempo** de la musique mixte qui n'est pas présent dans les automates temporisés. La ***problématique*** majeure est la suivante :

*Comment procéder à des tests en **temps réel** sur le système embarqué de musique mixte qu'est Antescofo ?*

Choix : Pour commencer, nous allons définir les points importants concernant la méthode de test choisie :
- Nous parlerons de **test "boite noire"** : Ce qui signifie que le code source de l'implantation testée est inconnu et que seul l'utilisation des ses entrées sorties est faite pendant les tests.
- Ces tests seront **basés sur modèle** (model driven) et les automates (temporisés) ont été choisis pour spécifier et tester Antescofo.

hal-00868618, version 1 - 1 Oct 2013

– Nous avons utilisé les outils développés par la série **UPPAAL** (ainsi que 2 extensions **coVer** et **Tron**) présentés dans la partie **III** (contribution).

Notre problématique englobe une question primordiale pour des tests : *Comment spécifier strictement Anstescofo en un modèle et y générer les tests ?* La réponse est simple, par **la partition**. Celle-ci comprenant le comportement attendu, y générer un modèle strict permet d'avoir indépendamment de l'implantation d'Antescofo une bonne spécification. Comme notre but est de tester le programme pour s'assurer de la réaction d'Antescofo pour n'importe quelle performance, nous allons tester ce comportement en fonction d'une partition, dite 'de test'.

Environnement : Antescofo étant un système embarqué en temps-réel, son environnement ne peut être dissocié. Ce point est souligné lors de l'article UPPAAL [AHS08] : une fois le système spécifié, il est nécessaire de pouvoir contrôler le modèle et les 'entrées/sorties' permettant d'interagir avec son environnement sont elles aussi spécifiées. Cet environnement doit donc être présent dans la spécification pour modéliser l'environnement extérieur d'Antescofo. Un modèle étant généralement un réseau d'automates temporisés, cette spécification d'environnement est un automate du réseau. Ce point est très important lors des tests car cette spécification sera l'environnement de test pour le modèle et la génération des tests. Il est inutile de spécifier un environnement trop général, contenant des situations souvent impossibles voire absurdes (porteuses d'erreurs que l'on pourrait se passer). En revanche, s'il est trop spécifique, aucun ou un unique cas de test sera possible lors de la génération de la suite de test (ce qui peut être inutile du point de vue d'un critère de couverture, ou utile pour un cas de test pertinent ou spécifique voulu).

L'image 8 présente l'architecture générale des tests sur Antescofo. C'est la partition de test qui est prise comme base pour créer automatiquement **le modèle du système** comprenant le modèle de l'environnement (partie musicien) et la spécification (qui spécifie le comportement d'Antescofo pour cette partition électronique). Nous faisons ainsi **des tests pour une partition donnée**.

C'est pour cela que le schéma image 8 est fait d'un point de vue 'machine réactive' : la spécification correspond à la modélisation de la machine réactive d'Antescofo pour la partition de test. Le schéma illustre deux parties : La partie supérieure présente une exécution réelle d'Antescofo, on y voit : ses entrées, qui sont les informations de la machine d'écoute en fonction d'une performance de musicien, et ses sorties, les messages vers un environnement audio pour appliquer l'action voulue. Attention les jargons sont différents dans les deux mondes, une entrée est nommée un 'événement' (simplement une note pour les cas simples) dans Antescofo et décrit avec un '?' en fin de label dans un automate temporisé. Les sorties sont des 'messages' pour Antescofo et décrit avec un '!' en fin de label dans un automate temporisé.

hal-00868618, version 1 - 1 Oct 2013

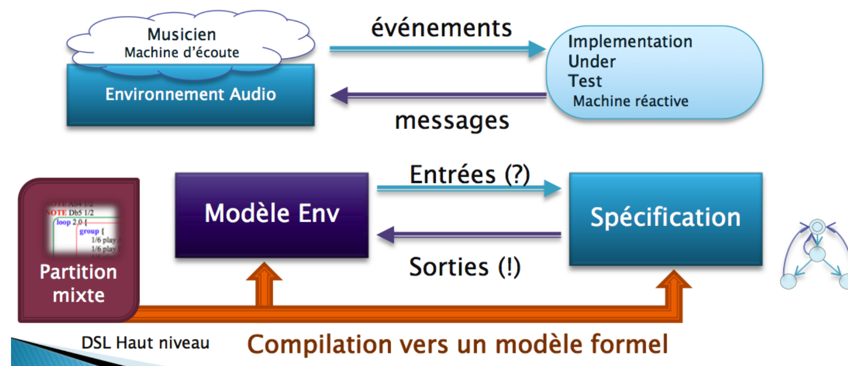


FIGURE 8: Schéma de test sur Antescofo

En dessous est schématisée la méthode de test choisie. Elle consiste à partir d'une partition de musique mixte pour construire automatiquement un modèle grâce à **un compilateur**. Ce modèle étant un réseau de machines à état fini.

Le contexte et l'orientation du stage ont été exposés, ceci pour comprendre pourquoi et comment ce sujet de recherche est traité. Pour continuer, la partie **II** va exposer les avancés existantes au début de mon stage.

hal-00868618, version 1 - 1 Oct 2013

Deuxième partie
Existant

De longues études ont été faites antérieurement à mon stage, pour créer un langage de partition mixte, étudier la stratégie de test, s'approcher d'un langage modèle plus simple pour opérer la génération de test ... La complexité d'Antescofo le rend difficile à spécifier, même pour une partition, ce système réagit pour n'importe quelle performance possible et l'on veut tester toutes ces réactions. La partition de musique mixte donnée en entrée est de plus (de part la complexité du système) écrite dans un langage haut niveau, propre à Antescofo et à son contexte.

Dans cette partie nous allons énoncer ce qui a été fait avant le début de mon stage. Nous commencerons par présenter une partie restreinte de ce langage haut niveau, puis décrirons la méthode choisie pour créer nos modèles de tests spécifiant le comportement d'Antescofo.

2 Antescofo : Langage haut niveau

L'écriture de partition musicale est une ancienne pratique. Elle a beaucoup évolué et comprend une multitudes d'annotations, permettant aux compositeurs d'être le plus précis et concis possible. Il en est de même pour la musique mixte qui n'est qu'une extension de la musique instrumentale. Le langage Antescofo utilisé pour écrire une partition de musique mixte se doit et permet donc aux compositeurs d'écrire une partition avec ces mêmes avantages.

2.1 Syntaxe

Commençons par donner une partie de la grammaire d'Antescofo à laquelle on se limitera dans ce mémoire. L'explication est tirée du papier JDEDS où toute la sémantique et ce qu'il peut être écrit sur Antescofo est inscrit [JE11] :

score	:=	*event* | *event score* | (d *group*) *score* | (d *loop*) *score*
event	:=	(e c)
group	:=	**group** l *synchro error* (d *action*)+
loop	:=	**loop** l *synchro error* p n (d *action*)+
action	:=	a | *group* | *loop*
synchro	:=	**loose** | **tight**
error	:=	**local** | **global**

Description rapide : Une partition peut contenir des événements (entrées d'Antescofo depuis le(s) musicien(s)), ou des actions (sorties vers l'environnement électronique). Un événement possède un événement instrumental (e : une hauteur de note) suivi de

hal-00868618, version 1 - 1 Oct 2013

sa durée (c). Une action atomique est notée a (un message de sortie), d représente un délai et l un label. Pour les boucles, p représente la période de celle-ci en durée (2 pulsations ou 2 secondes) et n le nombre de répétition. Les mots italiques sont des mots *abstraits* (non terminaux) et les mots en gras sont les mots clés du langage.

2.2 Tempi, synchronisation et erreurs

```
BPM 60
NOTE a4 1
NOTE g4 0.5
NOTE f4 0.5
NOTE e4 2
```

- that is

Figure 9: Parallèle Antescofo - Partition traditionnelle (Partie musicien)

Voyons ce qu'il est possible de faire par des exemples, pour commencer voici image 9, une partition simple. Cette partition Antescofo comporte uniquement la partie instrumentale, c'est à dire la performance idéale du musicien pensée par le compositeur. On y voit décrit le tempo qui devrait être pris (60 bpm) et 4 notes. Aucun problème pour Antescofo, qui ne doit ici qu'écouter l'environnement sans devoir jouer en retour. Lors d'une performance, il ne serait ici qu'une personne du public, partition en main, qui listerait les notes entendues et calculerait le tempo réellement pris par le musicien, ceci pour chacune des notes. *A relever la notation relative des notes : 1 temps correspondant à une noire, 0.5 à une croche et 2 à une blanche.* La syntaxe d'une note est donc ainsi : NOTE hauteur durée (Ceci est une version explicative et simplifiée, beaucoup plus d'informations et de manières d'écrire une note sont possibles)

L'image 10 nous présente une partition plus intéressante. Une partie spécifie (comme la première partie pour le musicien) la réaction électronique. Cette partie électronique sera exécutée par la machine réactive d'Antescofo chargée d'envoyer les messages de sorties (appelés actions). *A noter qu'une partie musicien est obligatoire pour permettre à Antescofo de se synchroniser et de savoir à tous moments où est le(s) musicien(s) dans la partition jouée (comme un humain connaîtrait ou lirait la partie des autres musiciens).*

Image 10 : Partition : Trois portées sont illustrées pour ce morceau. La première (la plus haute), est le jeu du musicien. Les deux suivantes constituent le jeu électronique, séparé en deux instruments électroniques (un pour chaque portée). Langage Antescofo : Ceci est représenté par une suite de NOTE, décrivant la portée du musicien. Les actions d'Antescofo sont listées dans des groupes, se déclenchant lorsque la note correspondante du musicien est détectée. Ici une première boucle (de deux pulsations), jouant infiniment son corps, est lancée lors de la seconde note détectée. Ce corps est constitué de deux parties, un groupe et une deuxième boucle d'une pulsation. Cette écriture représente le même effet que les notes illustrées sur les portées électroniques de la partition.

hal-00868618, version 1 - 1 Oct 2013

hal-00868618, version 1 - 1 Oct 2013

Figure 10: Parallèle Antescofo - Partition traditionnelle (Partition mixte)

Une action se décrit comme une note, elle possède un nom (en fonction de l'environnement électronique extérieur) et un temps. **Attention** : Ce temps (relatif dans notre exemple image 10) représente le **délai** à attendre avant d'envoyer l'action et non la durée de celle-ci. Aussi pour la petite boucle bleue, la première note ne dure pas 0 temps mais est envoyée dans 0 temps (au même moment que la détection de la note du musicien), la suivante sera envoyée un demi temps plus tard (que le premier envoie de message).

Deux points importants sont illustrés figure 10 : **la synchronisation** avec la partie du musicien (il faut qu'Antescofo envoie l'action au bon moment (ni trop tôt ni trop tard) par rapport à la note du musicien sur le même temps). Visuellement, cela correspond à l'alignement vertical des notes sur les différentes portées, un même alignement est équivalent au même temps du morceau. Ainsi que **la signification** d'une action ou d'un groupe, ces deux boucles sont écrites après la deuxième note de la partie instrumentale : Ce qui signifie qu'une fois cette note détectée Antescofo enverra, aussitôt, la réaction électronique en ne tenant compte que des variations de tempo. Tout délai de note dans ce groupe sera instancié avec le précédent tempo (obtenu avec la dernière note détectée du musicien), ce qui veut dire que pour la 6eme note du groupe vert (le premier accent *(un accent est un > au dessus d'une note)*) son délai sera 1 pulsation après le déclenchement du groupe (2eme note du musicien). Hors il n'est pas dit (et voire impossible) que ce musicien garde exactement le même tempo, qu'il ne ralentisse ou n'accélère pas. Dans ce cas cet accent ne sera jamais aligné (joué au même moment) avec la 4eme note du musicien, ce qui peut être exigé par le compositeur.

(Nous pouvons voir ici, les différentes possibilités d'écrire une même partition et la complexité qu'elle peut représenter)

L'exemple figure 11 reprend la dernière question, celle des stratégies de synchroni-

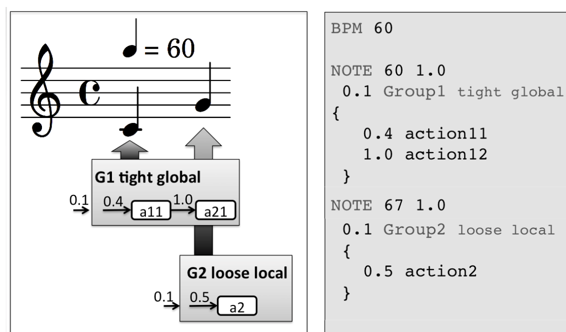


Figure 11: Parallèle Représentation Antescofo sur partition - Antescofo (Traitement d'erreur)

sation. La partition est plus courte mais les groupes ont des attributs définissants la **stratégie de synchronisation** et **la gestion d'erreur**. Cette première notion **(tight ou loose)** permet de décomposer un groupe en fonction de sa synchronisation. Chaque action d'un groupe **tight** se verra rattachée directement à l'événement le plus proche (comme si elle était écrit dans un groupe différent et lié à cette note) automatiquement (cela permet une cohérence d'écriture pour le compositeur en maintenant la synchronisation désirée). L'effet obtenu est l'exécution de cette action lors de la reconnaissance de ce nouvel événement, et non après une attente sur un précédent tempo. Si cet attribut *tight* n'est pas mis le groupe est **loose** et laissé comme dans l'exemple image 10.

La seconde notion découle directement de la musique mixte. Un musicien peut aisément comprendre (voire même entendre) qu'une note est manquée ou mal jouée par le musicien en cours de performance, un ordinateur non. La gestion d'erreur permet de laisser le choix au compositeur de la réaction d'Antescofo face à ce genre d'événements. Il y a également deux possibilités **global ou local**, la première joue immédiatement le groupe d'action une fois la note manquée détectée, alors que la seconde ignore simplement le groupe et ne joue aucunes actions de celui-ci.

Ces deux notions sont indépendantes l'une de l'autre, nous avons donc quatre combinaisons possibles, voici les différents comportements décrient par l'image 12. Elle décrit les réactions survenues lorsqu'une note (la première) est ratée :

2.3 ...et d'autres instructions

Ce langage devient très complet, avec cela s'ajoute les instructions de contrôles, les boucles, les macros ... Le langage d'Antescofo est finalement un vrai langage de programmation, une partition devient donc un programme en elle même. Il est donc nécessaire de lui donner une sémantique claire afin de pouvoir générer nos tests rigoureusement.

hal-00868618, version 1 - 1 Oct 2013

hal-00868618, version 1 - 1 Oct 2013

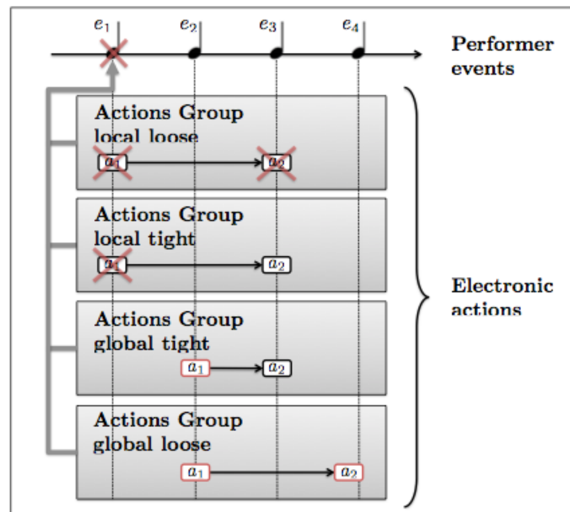


FIGURE 12: Stratégies d'attributs de groupe

A titre d'exemple une partition 'exotique' pour les tests a été créée et est illustrée image 13. Sa syntaxe est compliquée et ne ressemble presque plus aux partitions mixtes du départ.

3 Langage "bas niveau"

Ce langage "bas niveau" désigne les automates temporisés, le but étant d'avoir un langage précis et simple pour spécifier proprement un modèle du comportement du système sous test. Un compilateur a été codé pour transformer une partition mixte d'Antescofo en un réseau d'automates temporisés. Le langage haut niveau étant riche, il est possible de *spécifier rigoureusement une partie* de ce langage pour pouvoir le tester proprement.

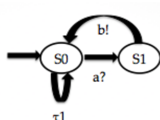
Nous allons ici décrire les automates et automates temporisés ainsi que le principes de compilation depuis une partition.

3.1 Les Automates à entrées/sorties

Cette partie est basée sur le papier d'Antoine Rollet qui définit formellement des automates et automates temporisés. Nous commencerons par énoncer sa description d'automates à entrées/sorties pour définir ensuite nos propres automates temporisés par une variante de sa définition et de l'utilisation que nous en faisons. Voici les IOLTS, des systèmes de transitions (automates) à entrées sorties labellisées :

```
// Score-driver genered from a coVer trace 1
// ----------------------------------------------------------------
Group Driver
    {
    // Traces in a file (output file)
    openoutfile traceout_1"/tmp/13-Output_1.txt"

     //      NOTE e1 1.25
     1.25 antescofo::nextevent

     5 closefile traceout_1
    }


// Score under test generated from the original score
// Score with a simple form and automata's labels
traceout_1 "START of traces at " $NOW " tempo = " $RT_TEMPO "\n\n"
// ----------------------------------------------------------------
NOTE 6200 1.5 e1
 traceout_1 e1 " now=" $NOW " tempo=" $RT_TEMPO " rtime=" $RNOW " \n"

    0.5 group g0 @tight{
    0.25 traceout_1 "    " a0 " now=" $NOW " tempo=" $RT_TEMPO " rtime=" $RNOW " \n"
    0.25 group g1 @tight{
        0.1 traceout_1 "    " a1 " now=" $NOW " tempo=" $RT_TEMPO " rtime=" $RNOW " \n"
        0.1 traceout_1 "    " a2 " now=" $NOW " tempo=" $RT_TEMPO " rtime=" $RNOW " \n"
        }

    0.25 traceout_1 "    " a3 " now=" $NOW " tempo=" $RT_TEMPO " rtime=" $RNOW " \n"
    0.5 traceout_1 "    " a4 " now=" $NOW " tempo=" $RT_TEMPO " rtime=" $RNOW " \n"
    }


NOTE 6700 0.25 e5
 traceout_1 e5 " now=" $NOW " tempo=" $RT_TEMPO " rtime=" $RNOW " \n"

NOTE 6800 0.25 e10
 traceout_1 e10 " now=" $NOW " tempo=" $RT_TEMPO " rtime=" $RNOW " \n"

NOTE 6900 0.5 e12
 traceout_1 e12 " now=" $NOW " tempo=" $RT_TEMPO " rtime=" $RNOW " \n"

 traceout_1 "STOP ##### " $NOW " \n"
```

Figure 13: Illustration d'une partition de test



Les automates d'entrées sorties ordinaires sont généralement comme celui-ci. Ils comportent **une transition initiale** (ou un état initial), représentant l'emplacement du premier état (montré par une flèche sans prédécesseur sur l'exemple (état S0)). **Des états**, dans lesquels est fait une attente d'un ou d'événement(s) (lorsque celui-ci arrive une transition est empruntée). Et pour finir **des transitions** partant d'un état source pour aller sur un état cible (possiblement lui même), elles possèdent différents types (selon leur label) : **les entrées**, marquées par un '?' représentant une interaction venant de l'environnement extérieur (par exemple une pression de bouton), un état ayant une transition d'entrée veut dire qu'il peut réagir à cet événement. **Les sorties**, marquées d'un '!' sont des envois du système vers l'environnement extérieur (un affichage ou une boisson pour une machine à café ...). Un état ayant ce type de transition veut dire que le système *peut* émettre une sortie. Les IOLTS peuvent aussi avoir des transitions **internes** au système, elles sont représentées par un 'τ'.

hal-00868618, version 1 - 1 Oct 2013

Definition : Un IOLTS est un quadruplet M = $(Q^M, A^M, - >_M, q_0^M)$ tel que :

- Q^M soit un ensemble fini non vide d'états
- q_0^M soit l'état initial
- A^M soit un alphabet d'actions partitionné en trois ensembles disjoints tel que $A^M = A_I^M \cup A_O^M \cup I^M$ avec :
  - A_I^M l'alphabet d'entrées (notées avec un ?)
  - A_O^M l'alphabet de sorties (notées avec un !)
  - I^M l'alphabet des actions internes (notées $\tau_k, k \in \mathbb{N}$)
- $- >_M \subseteq Q^M$ x A^M x Q^M la relation de transition.

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

Des notations usuelles sont aussi fournit pour utiliser cette écriture : Soit en plus de M un IOLTS, $\mu_k \in A^M$ des actions quelconques, $a_k \in A_{VIS}^M$ des actions visibles, $\tau_k \in I^M$ des actions internes, $\sigma \in A_{VIS}^M{}^*$ une séquence d'actions visibles, et q, q' et $q_k \in Q^M$ des états. On note :

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

Les tests résonnent essentiellement sur les traces, nous avons donc en plus :
- q $\xRightarrow{\epsilon}$ q' $\triangleq q = q' \vee q \xrightarrow{\tau_1 \tau_2 .. \tau_n}$ q'
- q $\xRightarrow{a} q' \triangleq \exists q_1, q_2 : q \xRightarrow{\epsilon} q1 \xrightarrow{a} q2 \xRightarrow{\epsilon}$ q' (ϵ étant la séquence vide)

L'on étend aux séquences d'actions visibles :
- q $\xRightarrow{a_1 ... a_n}$ q' $\triangleq \exists q_0, ..., q_n : q = q_0 \xRightarrow{a_1} q_1 ... \xRightarrow{a_n} q_n =$ q'
- q $\xRightarrow{\sigma} \triangleq \exists q' : q \xRightarrow{\sigma} q'$.

Et les traces sont maintenant définissables à partir d'un état :
$Traces(q) \triangleq \{ \sigma \in A_{VIS}^M * | q \xRightarrow{\sigma} \}$, et par extension
$Traces(M) \triangleq Traces(q_0^M)$
En prenant l'exemple de l'image 14 nous pouvons écrire :
- $\Gamma(s_0) = \{$?d, ?r$\}$
- Out$(s_0) = \phi$
- In$(s_0) = \{$?d, ?r$\}$
- $s_1 \xRightarrow{\epsilon} s_2$
- $s_1 \xRightarrow{?d} s_3$
- $s_1 \xRightarrow{?d.!o} s_0$
- $s_2 after?d.!o = \{s_0, s_4\}$
- $s_0 after?d.!a = \phi$
- $\{s_0, s_2\} after?d = \{s_1, s_2, s_3\}$

hal-00868618, version 1 - 1 Oct 2013

hal-00868618, version 1 - 1 Oct 2013

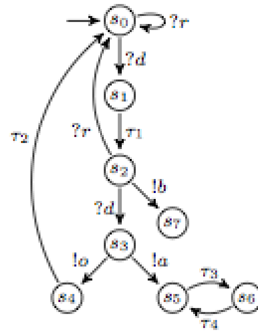


FIGURE 14: S : Un automate avec entrées sorties

– $\mathit{Traces}(\mathrm{S}) = \mathit{Traces}(s_0) = \{\epsilon, ?d, ?r, ?d.?r, ?r.?d, ?d.!b, ...\}$

De cette base, nous pouvons continuer sur les automates temporisés utilisés pour notre recherche.

3.2 Automates temporisés à entrées/sorties

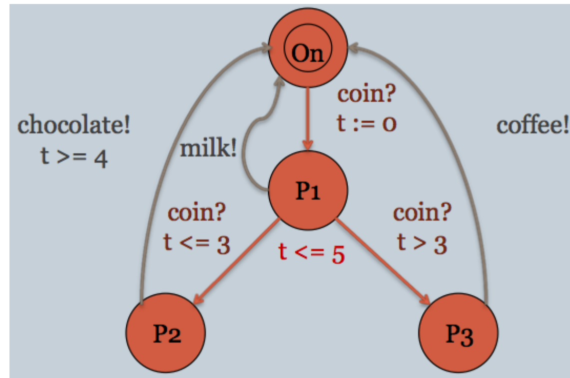


FIGURE 15: Un automate temporisé

Dans son article, Antoine Rollet s'inspire des automates temporisés d'Alur et Dill, ce qui diffère un peu de la version prise pour notre stage. En temps réel la notion d'horloge (de temps écoulé) entre en jeu, l'on trouve toujours les entrées/sorties mais une ou des variable(s) sont ajoutées pour mesurer le temps. Voici la définition prise pour notre stage :

Definition : Un TAIO est un quintuplet A = $(S, s00, T, Act_\tau, E)$ tel que :

- S soit un ensemble fini de localités
- $s00 \in S$ soit la localité initiale
- T soit un ensemble fini d'horloges
- $Act_\tau = Act_I \cup Act_O \cup \{\tau\}$ l'ensemble des actions
- E soit l'ensemble des transitions (s, s', ψ, r, a) avec :
  - s, s' $\in$ S les localités source et destination
  - ψ la garde (une conjonction de contraintes t#c, avec t $\in$ T, c $\in \mathbb{N}$ et # $\in \{<, \leqslant, \geqslant, >\}$)
  - r $\subseteq$ T : l'ensemble des horloges à réinitialiser
  - a $\in Act_\tau$ l'action

Le temps ajoute beaucoup de problèmes sur comment spécifier les modèles, car si l'on considère que le temps peut passer sur tout état de l'automate une infinité de possibilité peut apparaître. (Une sortie peut être émise au temps t mais aussi t+1 ou t+2 ...). On peut donc trouver des contraintes supplémentaires dans les automates temporisés. Pour Antoine Rollet, les transitions de sorties possèdent une deadline, on trouve :

- *eager* : Est une transition **forcée**, dès que la sortie est possible elle est prise immédiatement. On peut aussi dire que le temps ne peut s'écouler sur un état qui possède une transition de sortie 'eager' valide.
- *delayed* : Est une transition que l'on peut **retarder**. Le temps peut s'écouler **si elle est toujours possible** après.
- *lazy* : Est une transition paresseuse, elle peut être prise ou laissée.

Les transitions de types eager ne peuvent pas avoir des gardes de forme : x > c. **Attention** : souvent des invariants sont placés sur des localités (forçant la prise d'une transition de sortie delayabled ou lazy par exemple)

Une première différence est la *non utilisation* de deadlines pour nos transitions. Pour notre étude, les actions sont ***toutes eagers*** et doivent être lancées dès que leur garde est valide. Les localités ne possèdent donc pas, non plus, d'invariants.

Nous pouvons maintenant définir formellement les deux types de transitions possibles pour notre automate temporisé : discrètes (entrées ou sorties) et temporelles (écoulement du temps) :

- E_A un ensemble fini d'états e = (s, v) avec s $\in$ S une localité, v : T $\rightarrow \mathbb{R}^+$ une valuation d'horloges
- $e_0^A = (s_0, \vec{0})$ l'état initial ($\vec{0}$ étant la valuation d'horloges les affectant toutes à 0)
- Tr_d les transitions discrètes : (s, v) $\xrightarrow{a} (s', v')$ ssi $\exists$(s, s', ψ, r, a) $\in$ E, v $\vDash \psi \wedge v'$ = reset(r) dans v (v' est la valuation obtenue à partir de v en réinitialisant toutes les horloges de r)
- Tr_t les transitions temporelles : (s, v) $\xrightarrow{t} (s, v + t)$ pour t $\in \mathbb{R}^+$ ssi il n'existe pas de transition (s, s', ψ, r, a) $\in$ E telle que v $\vDash \psi$

Il est important de bien définir le langage formel utilisé lors de la spécification,

hal-00868618, version 1 - 1 Oct 2013

voyons maintenant pour terminer le compilateur de partition, chargé de faire le lien entre une partition et les automates temporisés.

4 Traduction d'automate

Nous allons parler ici du compilateur, déjà implanté au commencement de mon stage. Il est basé sur le parseur d'Antescofo, ce qui lui permet de lire et d'avoir la même partition-AST que celui-ci. Pour le moment, il ne traite qu'une partie restreinte du langage haut niveau (à savoir ce qui est listé plus haut dans la grammaire). Le papier JDEDS ([JE11]) informe plus précisément des traductions apportées pour certaines instructions de la partition. ([LF13]) Ce papier informe aussi de la formalisation d'une partition mixte en automates temporisés. Voici un exemple, image 16, de Florent Jacquemard pour un cas de groupe "tight" : La première partition est la partition comprise après remaniement du groupe *tight*, le compositeur ayant écrit la partition de droite.

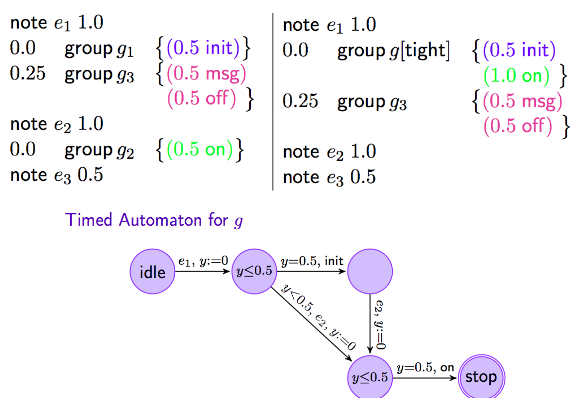


FIGURE 16: Traduction d'un groupe tight

hal-00868618, version 1 - 1 Oct 2013

hal-00868618, version 1 - 1 Oct 2013

Troisième partie
Contribution

Avant ce stage, les tests sur Antescofo se déroulaient directement par bandes sons pré-enregistrées ou performances de musicien(s). Ils visaient principalement la machine d'écoute du système. Les tests de la machine réactive basés sur modèle, sont nouveaux et n'ont pas d'antécédents.

Les contributions au sujet de recherche ont été apportées sur le compilateur, et par une génération de test basée sur des logiciels existants. Nous avons grâce à ces automates, une spécification précise d'un comportement du système via une partition donnée. La question qui nous vient maintenant est *"Comment les utiliser pour appliquer des tests sur le système Antescofo en temps réel ?"*

Les contributions vont répondre d'elles même à cette question, nous allons commencer par détailler ma contribution au compilateur et en profiterons pour décrire l'environnement UPPAAL ainsi que la vision des automates qu'il utilise. Nous expliquerons les utilisations d'UPPAAL et de ses extensions qui possèdent différentes visions de génération de test. Pour chacune de ces visions (dites offline ou online) nous allons présenter les extensions utilisées et la contribution (réellement implantée ou seulement pensée) que j'ai effectué durant le stage.

5 Principes de compilation

En ses débuts, le compilateur créait d'une (AST-)partition un réseau de machines à état fini. Il était nécessaire de créer un lien modélisant ces machines en automates temporisés compréhensibles par UPPAAL. Effectivement, d'après ses travaux, UPPAAL utilise une forme d'automates temporisés spécialement pour lui. Voici les contributions apportées au compilateur :

Contributions : Ajout, sous forme de design pattern Visiteur, d'une réécriture du réseau des machines à état fini (FSM) du compilateur. Ces visiteurs ont pour but d'afficher normalement dans le flux de sortie le réseau (nettoyage de code), de l'afficher en une version d'UPPAAL (.xta), ou en une version d'UPPAAL avec gestion de coordonnées graphiques dans les fichiers d'extension correspondante (.xta et .ugi). Ajout d'une option compilateur permettant de créer parmi le réseau de machines à état fini, une machine représentant la modélisation de l'environnement extérieur de test.

Pour expliquer plus concrètement, une présentation de la vision des automates temporisés d'UPPAAL va être faite. Son utilisation ainsi que des exemples en image de ce logiciel vont être présentés :

hal-00868618, version 1 - 1 Oct 2013

5.1 Vision temporisée d'UPPAAL

UPPAAL n'utilise pas totalement la même vision qu'Antoine Rollet pour ses automates temporisés. Les deadlines de ses transitions ne sont pas placées sur celles-ci mais ce sont ses états qui peuvent les caractériser (en plus de leurs invariants). Lorsqu'il parcours automatiquement l'automate, UPPAAL considère les transitions comme *eager*, un écoulement de temps est impossible si une transition (de sorties) est possible. Enfin pour un model checking, un environnement spécifié est nécessaire pour pouvoir tester et manipuler la spécification (modélisée par un réseau d'automates), ceci est un détail mais change la relation de conformité.

Etat : Un état peut être *Initial*, *Urgent* ou *Committed*. Cette première caractéristique précise qu'il est le premier état (S0). *Urgent* indique qu'aucun écoulement de temps n'est possible pour cet état mais autorise d'autres transitions dans le réseau d'automates temporisés que celles sortantes de l'état en question. *Committed* est identique mais n'autorise aucune autre transition que celles sortantes de lui même.

Automates temporisés d'UPPAAL : UPPAAL possède son langage pour décrire un réseau d'automates temporisés (Version qui a évoluée). La version la plus récente est sous forme *xml*, elle est un peu trop verbeuse pour l'écrire et la déchiffrer manuellement et comprend les coordonnées pour une présentation graphique de tous les composants du réseau. Une version antérieure appelée *xta* ne comprend que la définition du réseau et possède une verbosité raisonnable pour l'Homme. Elle contient : - les déclarations de variables globales et communes à tous les automates temporisés du réseau, la déclaration des automates et de leurs états, le types de ces états et les transitions entre ces états, puis pour finir, l'instanciation de ses automates dans le système (qui sont gérés comme des "templates" génériques) - . Une grande marge de manœuvre est ainsi autorisée pour la création de notre réseau d'automates temporisés.

Voici, image 17, à quoi ressemble le logiciel UPPAAL. Le premier écran présente la partie édition, il donne un aperçu graphique de chaque automate du réseau, ceux-ci sont listés dans la colonne de gauche. Le second écran est un simulateur et permet pour un réseau valide de simuler des traces (d'en charger et d'en créer également), ici aussi l'ergonomie graphique est importante. Le dernier écran concerne la partie vérification, où un langage de requêtes pour le test de modèle (model checking) est utilisé. Les requêtes sont traitées et validées ou non, en fonction des réponses possibles une trace est donnée comme verdict.

Un environnement modèle : Nous allons détailler plus précisément le modèle d'environnement. Il est un automate parmi le réseau de spécification du système, mais possède les sorties du systèmes comme entrées (côté "client" : prise de café, de monnaie, lecture d'information affichée ...) et les entrées du système comme ses propres

hal-00868618, version 1 - 1 Oct 2013

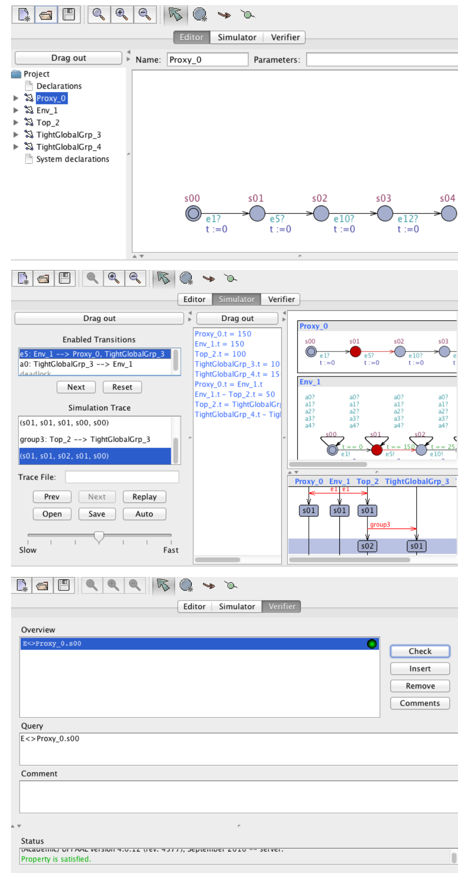


FIGURE 17: L'environnement graphique d'UPPAAL

sorties. Cette spécification est aussi importante que le reste car elle définie *les bornes et les comportements* que pourrait avoir **l'environnement extérieur**. Parmi les images figure 18, un système de machine à café avec 2 environnements possibles est présenté. Le premier environnement (2° image) est le plus général, tout est autorisé et à tout moment (plus un test est général plus il est lourd et plus de cas incohérents peuvent apparaître). Il est, par exemple, impossible d'obtenir une boisson avant d'avoir inséré au moins une pièce, et, une autre pièce n'est insérée qu'après un délai humainement possible. Ces deux notions sont appliquées sur le deuxième environnement, réduisant de ce fait les traces possibles et les cas de tests inutiles à générer.

Environnement Uppaal : Comme précisé dans les contributions, un automate supplémentaire a été ajouté dans les FSMs pour spécifier l'environnement de test. Celui-ci a été créé (exemple image 19 gauche), dans la plus simple des formes, c'est à dire que notre musicien joue à la suite les notes de la partition (avec au moins la

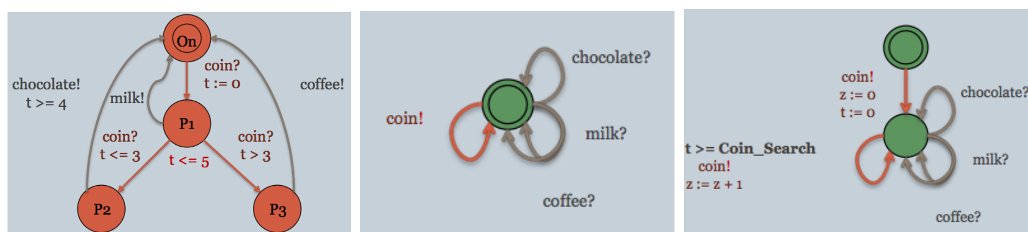


FIGURE 18: Système à café et ses environnements

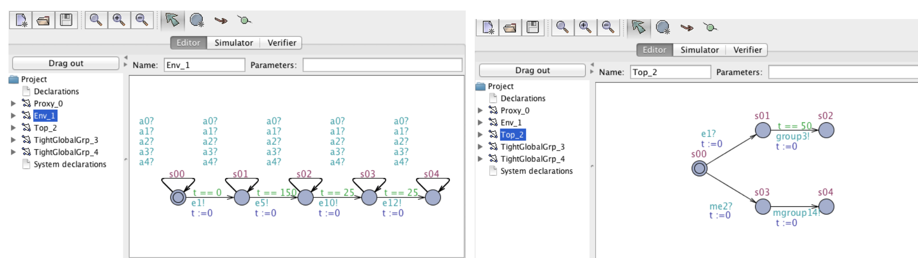


FIGURE 19: Environnement et graphique Uppaal

durée de la note) et qu'à tout moment l'environnement électronique peut recevoir un message de sortie venant d'Antescofo. Cet environnement peut avoir plusieurs formes et notamment peut modéliser le saut de note.

Calcul graphique : Le fichier *xta* peut avoir optionnellement son complément (fichier .ugi) indiquant les coordonnées graphiques des composants des automates. Cette contribution a consisté à calculer les coordonnées graphiques pour avoir une présentation ergonomique lors de l'affichage sur UPPAAL de notre réseau. En effet, les états n'étaient automatiquement pas bien placés. Ce fût pratique pour vérifier visuellement la création des réseaux de machines à état fini (le visiteur n'étant qu'un traducteur) et pour présenter nos travaux à nos collègues. (Comme exemples les automates affichés sur le logiciel UPPAAL sont directement ouverts depuis ces fichiers, images 17 et 19). Nous avons également un aperçu clarifié d'une partition à langage au niveau.

6 Le temps

Avant d'étudier concrètement nos générations de test, concentrons nous sur le temps. Sérieux problème pour nos implantations de test, il apporte des erreurs et quiproquos à éviter. De plus, Antescofo ajoute le temps relatif dans nos tests, ce qui place le temps comme un fondement du programme (pour Antescofo et donc nos programmes de test).

hal-00868618, version 1 - 1 Oct 2013

hal-00868618, version 1 - 1 Oct 2013

Temps relatif : Le temps est *relatif* lorsqu'il est écrit sur une partition ou sur un automate. Les valeurs de temps sont décrites par pulsations (beats) pour l'un et unités de temps (tic d'horloge) pour l'autre. Le premier problème se présentant est la distinction de ces ***deux unités de temps relatifs***. Pour pouvoir comparer ces deux valeurs la référence de ces temps relatifs doit être la même, l'automate doit avoir le même tempo d'horloge que sa partition (lors d'une performance) - le performeur (exécution d'un cas de test) doit avoir la même fréquence que l'horloge du réseau d'automates...

Antescofo : ... et les tempi d'Antescofo ? Antescofo se doit non seulement d'avoir son horloge de référence mais pire, cette référence change : A chaque événement *le tempo d'Antescofo est recalculé*. Cela reflète la complexité du temps absolu du monde réel (les secondes) qui n'est pas abstrait dans l'informatique.

Conséquence : Nous avons dans nos tests des comparaisons de traces dites 'patronnes' (attendues) avec une trace d'Antescofo correspondant au résultat d'une exécution de cas de test. Il faut avoir la même référence d'horloge pour la création de la trace 'patronne' et la comparaison de cette trace avec celle d'Antescofo. Celle-ci varie selon nos tests, des générations de tests utilisent l'Horloge d'Antescofo comme repère (le temps est écrit en relatif ainsi la conversion est faite par Antescofo (les détails sont énoncés lors de l'explication des tests)). D'autres une horloge arbitraire, représentant le tempo propre du musicien, modélisée par une courbe faisant varier (pour un temps relatif donné) le tempo de référence.

hal-00868618, version 1 - 1 Oct 2013

7 Génération de test Offline

Lors de la présentation d'UPPAAL dans l'introduction, deux méthodes de génération de suite de test différentes sont suivies. Elles comportent chacune des points de vues intéressants pour tester un système temps réel et comprennent des avantages et inconvénients. Nous allons commencer par la génération "différée" ou offline, qui génère premièrement depuis les modèles une suite de test pour ensuite l'exécuter sur le système testé : ***l'implantation sous test***. Voici le schéma général que nous avons suivi :

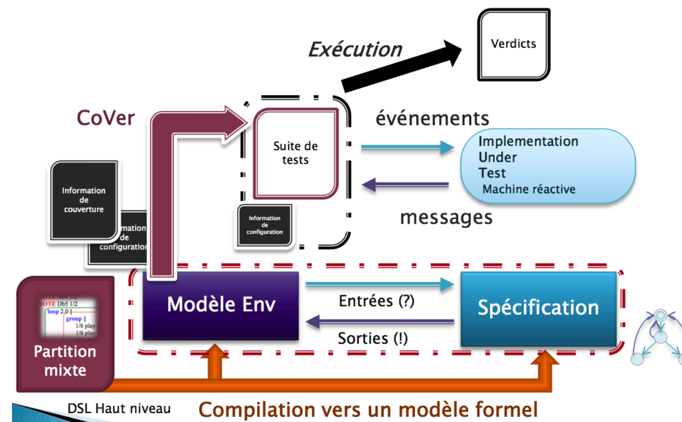


FIGURE 20: Notre méthode de test Offline utilisant coVer

Image 20 : *(Cette image se lit du bas gauche au haut droit)* Dans notre contexte nous commençons en bas à gauche par la partition de musique mixte choisie pour les tests. Le compilateur compile et modélise l'environnement ainsi que la spécification du comportement désiré. Depuis ces deux modèles (composants d'un même réseau d'automates UPPAAL), la génération des cas de test est opérée par coVer et ses fichiers de configuration. Le verdict est finalement obtenu, pour chaque cas de test, après l'exécution de cette suite sur l'implantation sous test. *On rappel que les modèles et l'implantation doivent être déterministes, cela ne nous contraint en rien car une partition possède un contexte déterministe.*

Nous allons décrire comment fonctionne coVer et ce qu'il demande comme configuration. La génération de suite de test comporte beaucoup de points critiques où des adaptations pour le contexte ou le système à testé sont requises, nous allons ensuite détailler plus précisément nos différentes contributions pour cette méthode de génération.

7.1 coVer

La méthode empruntée par uppaal pour ses tests est de transformer tout problème de couverture par un problème d'atteignabilité (ou à l'inverse de sécurité) se vérifiant par 'model checking'. Ainsi il est facile d'utiliser uppaal et d'obtenir (ou non) une trace de réponse. Les papiers concernant coVer nous expliquent comment transformer ces requêtes en formules compréhensibles par uppaal, ceci en utilisant : - des environnements de tests (précisant un cas de test), des formules d'atteignabilité simples (par le langage de requête d'uppaal) ... -.

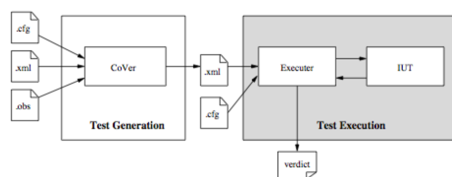


FIGURE 21: Configuration de coVer

L'image 21 présente le test différé par CoVer. Celui-ci ne se consacre qu'à la génération de test (le grand carré de gauche). Il se présente comme un exécutable et prend en entrée *plusieurs fichiers de configuration* pour donner en sortie un *nouveau fichier* contenant une suite de tests. Les informations d'entrées fournissent le modèle, le fichier de sortie et le critère de couverture pour la génération de suite. Les fichiers sont :

[input] <fichier>.["xml""xta"]	: Le réseau d'automates temporisés spécifiant le système.
[input] <fichier>.q	: Les requête(s) déterminant le critère de couverture
[output] (option -f) <fichier>.tr	: Le fichier de trace, contenant la suite de cas de test

.xml ou .xta : Le modèle du système. Il doit spécifier le système et l'environnement de test. C'est un réseau d'automates temporisés d'UPPAAL.

.q : Le langage de requête d'uppaal, étendu par un système "d'observateur" (fichier .obs). Les observateurs permettent de définir *le critère de couverture* voulu pour la suite de test générée automatiquement. Ils sont utilisés pour créer les cas de tests *les plus couvrants.* Un observateur est *un automate* avec pour chaque état un "objet observé", si l'on veut un critère de couverture sur toutes les transitions de notre modèle, chaque transition du modèle sera un état de l'observateur. Cet automate étant mis en parallèle avec le modèle pendant la génération de test, ses transitions se 'valident' lorsqu'une *trace générée* emprunte cette transition, la suite de cas de test se termine une fois que toutes les transitions ont été empruntées.

hal-00868618, version 1 - 1 Oct 2013

.tr : *Attention, le nom étant définissable en sortie, l'extension .tr est facultative mais recommandée* : Cover transforme ensuite les diagnostiques donnés par Uppaal en une suite de cas de test, un cas étant ni plus ni moins qu'une trace ordinaire. C'est à dire une simple suite d'événements discrets (une entrée ou une sortie) suivi optionnellement et alternativement d'une transition temporelle (un délai). La génération se fait par un algorithme choisissant le cas de test le plus couvrant possible, ainsi la longueur de la suite générée pour satisfaire le critère de couverture sera moindre.

7.2 Utilisation

Cette partie va présenter l'application des tests en utilisant la méthode différée. Deux méthodes ont été dissociées selon la référence d'Horloge prise pour les cas de tests : L'horloge même d'Antescofo ou une Horloge arbitraire. Ces deux méthodes changent jusqu'à l'organisation des tests, elles vont donc être expliquées l'une à la suite de l'autre :

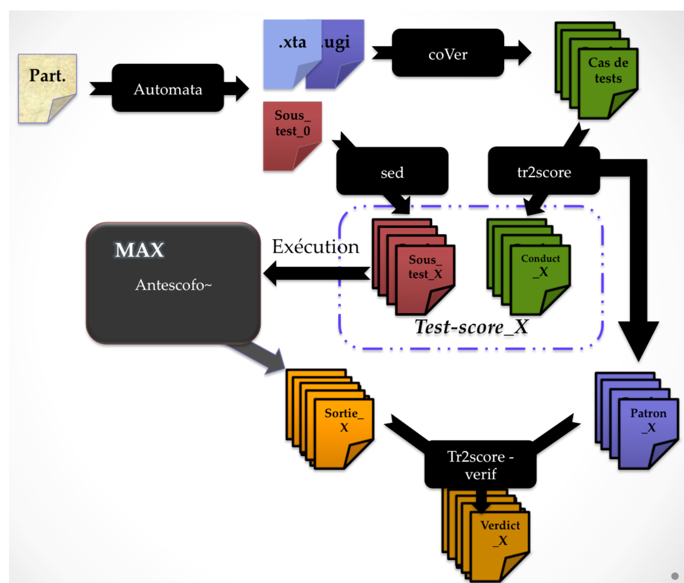


FIGURE 22: Tests différés 1

Génération de tests différés référencés sur l'Horloge d'Antescofo : Ces tests sont basés sur l'horloge d'Antescofo, c'est à dire qu'aucune conversion de temps (vers un temps absolu) n'est faite. Le temps est toujours considéré relatif au tempo d'Antescofo. Cependant ce ne sont pas les durées idéales (de la partition) qui sont jouées (ce serait tester simplement la machine réactive d'Antescofo sans problèmes) mais des durées générées automatiquement grâce à coVer.
Comme le montre l'image 22 l'on commence (en haut à gauche) par une partition.

hal-00868618, version 1 - 1 Oct 2013

Celle-ci est traduite par **le compilateur** (nommé "automata") en un réseau d'automates temporisés : le modèle, et une partition modifiée pour le passage de nos tests (partition sous tests) : cette modification concerne principalement l'insertion de traces de sortie. La génération de la suite est faite par coVer qui est configuré pour couvrir tous les états de l'automate environnement. Il jouera ainsi au minimum toute les performances voulues par notre environnement modèle. Nous obtenons une suite de plusieurs cas de tests traduits (par le programme créé à cet effet "tr2score") en plusieurs partitions conductrices et patronnes. La partition conductrice possède la performance du musicien pour un cas de test, aussi une partition conductrice est créée pour chaque cas de test présent dans la suite de coVer. De la même manière et pour ne pas mélanger les cas de test, une partition patronne, contenant la réaction que doit avoir Antescofo, est créée pour chaque cas de test.

L'étape "Génération de cas de test" est terminée, il faut maintenant les exécuter et comparer nos traces avec les réactions d'Antescofo obtenue en sortie d'exécution. Une option -verif de tr2score, compare la trace de sortie X au patron correspondant et retourne un verdict. Ici la comparaison est simple, tout temps étant relatif une vérification de la date relative des envois de messages est suffisante pour la réaliser.

Ces tests se concentrent sur la machine réactive, en vérifiant si pour une performance spéciale, les messages sont envoyés au bon moment. Rester en temps relatif permet d'omettre la complexité des horloges de notre monde réel, le temps absolue, et de se focaliser sur les tests. Les changements de tempo sont choisis implicitement par coVer.

Génération de tests différés référencés sur une horloge Arbitraire : Ces tests sont basés sur une Horloge arbitraire mimant une variation de tempo du ou des performeur(s). L'horloge est une fonction H prenant un temps relatif en entrée (depuis le début du morceau), par exemple le 5eme temps (5T : RT) et renvoie son temps absolu correspondant en milliseconde (T). Nous avons donc la fonction $H : RT -> T$.
La variation du tempo étant implicitement une performance de musicien la machinerie de coVer n'est plus nécessaire pour générer des cas de test. Ces conversions peuvent être fait directement depuis la performance idéale (de la partition, donc du modèle d'environnement).
L'image 23 présente le schéma général. La partition est toujours traduite par le compilateur "automata". L'automate est transformé en une partition conductrice ayant sa fonction de conversion H (utilisée pour lancer les événements du musicien en temps absolu). Une partition patronne est aussi créée en temps absolu, contenant les bons moments ('timetamps') où doivent être jouées les actions d'Antescofo. De la même manière l'exécution du cas de test sur le système sous test donne une trace de sortie des actions et événements d'Antescofo qui va être comparée avec la trace patronne.

hal-00868618, version 1 - 1 Oct 2013

hal-00868618, version 1 - 1 Oct 2013

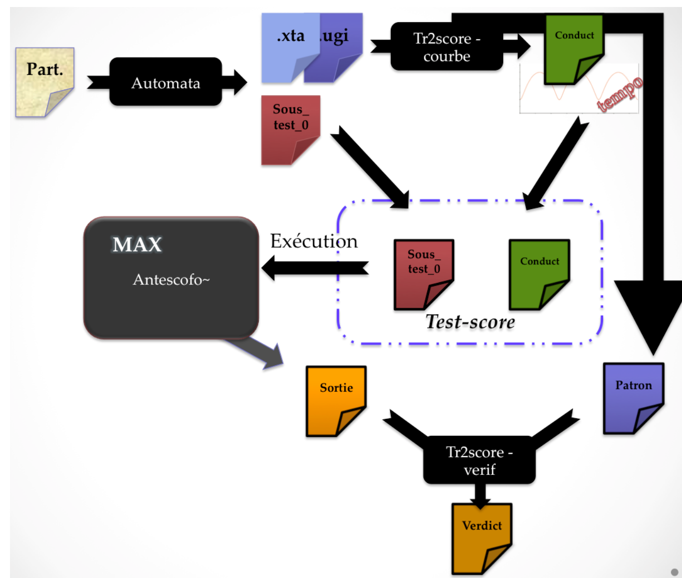


FIGURE 23: Tests différés 2

8 Génération de test Online

Cette partie se consacre aux générations de test dites ”à la volée” (on-the-fly). Elle présente les idées de contribution que l'on aurait pût développer avec plus de temps. Certains problèmes (voir partie **IV.9.2** Online) n'ont pas permis la réalisation de ces idées. La méthode de référence sur l'Horloge d'UPPAAL est impossible à réaliser par l'extension (Tron), nous préciserons pourquoi et comment, par une méthode personnalisée, nous aurions pût générer ce test :

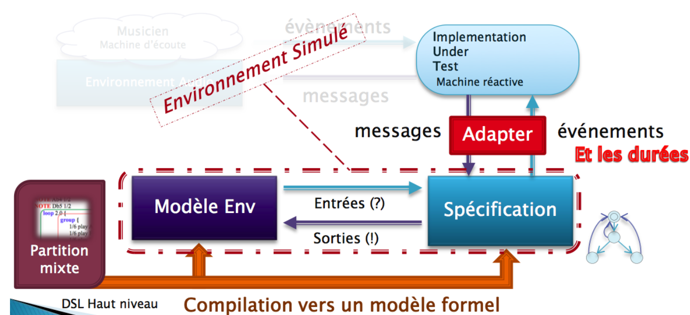


FIGURE 24: Notre méthode de test Online

Image 24 : *(Cette image se lit du bas gauche au haut droit)*. Elle nous montre le principe du test à la volée : Exécuter directement le cas de test pendant sa génération. Cela fait du système de test lui même un système temps réel. Il utilise également un modèle, généré par le compilateur depuis la partition choisie pour le test. En fonction de ce modèle, le calcule de l'ensemble *des états possibles après 'un pas'* ***(Z)*** est fait à chaque début de boucle dans laquelle le générateur : - Vérifie la validité d'une sortie du système sous test (∈ **Z**) ou, attend un délai ou simule une entrée sur le système sous test (∈ **Z**) - Tant que l'exécution continue c'est qu'aucune erreur n'est rencontrée, le verdict est donc positif, sinon c'est qu'un état du système de test n'est pas spécifié par le modèle.

Nous allons commencer par présenter Tron, ce qu'il demande et comment il fonctionne. Nous pourrons ensuite nous intéresser aux idées de contributions que nous avons eu.

8.1 Tron

Tron est également une extension d'UPPAAL, comme coVer il se présente sous forme d'un exécutable. A raison plus forte, car devant être lancé avec le système sous test, les adaptateurs sont plus importants et difficiles à concevoir.

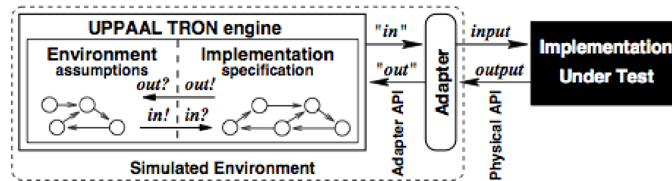


FIGURE 25: Configuration de Tron

Nous voyons ici (image 25) la génération à la volée de test. Tron est présenté partie gauche avec le modèle du système. Comme pour CoVer ce modèle doit comporter un modèle d'environnement, utilisé pour interagir avec la spécification. Tron s'en sert pour simuler un environnement extérieur au système sous test avec lequel il communique par un adaptateur. *Ceci induit implicitement une implantation d'adaptateur côté système sous test également.*

Lancement : L'architecture de Tron est présentée comme un client/serveur, le système sous test pouvant être serveur ou client. La communication se faisant par TCP/IP, les deux systèmes peuvent même être sur différentes machines. Au lancement Tron n'a besoin que **du modèle** : le réseau d'automates temporisés d'UPPAAL et d'un fichier de configuration indiquant les **entrées/sorties observables**. C'est à dire

hal-00868618, version 1 - 1 Oct 2013

les entrées sorties utilisées pour la communication modèle-environnement/spécification dans le réseau d'automates temporisés (les autres étant considérées comme transitions internes). La phase de configuration peut ensuite commencer :

Configuration : Lors de la configuration c'est le système sous test qui va 'paramétrer le cas de test' en informant à Tron :
- les entrées/sorties utilisées (Tron vérifie la cohérence des informations) ceci pour y lier un numéro lors du test
- les variables liées à ces entrées/sorties
- l'unité de temps du modèle d'automates temporisés [mtu] (*liaison du temps relatif du modèle (tic d'horloge d'automate) au temps absolue* : 1 mtu instanciera 1 tic par seconde)
- la longueur du test.

Exécution : Une fois configuré, le test peut commencer. A noter que les entrées/sorties ordinaires du système sous test sont toujours possibles. Ceci est prévu pour constater qu'une incohérence entraîne un échec du test (ou au contraire qu'une perturbation ne l'interrompt pas). Des études sont faites sur les problèmes de latences (aussi bien de communication que de concurrence système) et les solutions apportées sont expliquées en détail sur leur papier. Un système de temps virtuel a également été créé pour omettre ces problèmes lors des tests et ne faire tourner le temps que lorsque cela est nécessaire (Horloge-serveur). On trouve également beaucoup d'options pour la gestion des logs, des stratégies de génération du test...

8.2 Utilisation de Tron

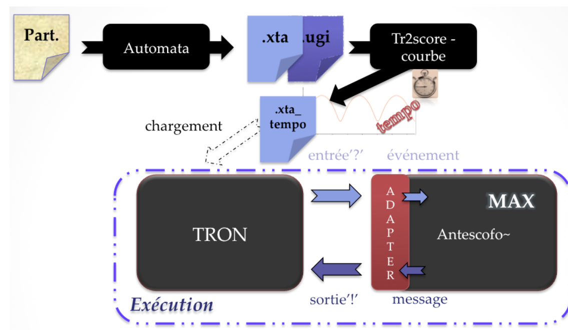


FIGURE 26: Tests à la volée 1

Génération de tests à la volée référencés sur une horloge Arbitraire : C'est effectivement la seule référence possible avec Antescofo et Tron car il est impossible de communiquer le temps relatif depuis Tron (instanciation par le mtu obligatoire). L'on

hal-00868618, version 1 - 1 Oct 2013

ne peut donc pas se baser sur l'Horloge d'Antescofo pour instancier les temps relatifs du modèle.

L'idée serait d'introduire la courbe de tempo dans Tron, pour qu'elle instancie en temps réel les valeurs relatives du modèle. Tron 'jouera' ainsi la partie musicien avec une instanciation des durées idéales pour un tempo aléatoire (ou par d'autres algorithmes).

8.3 Version Ad'Hoc

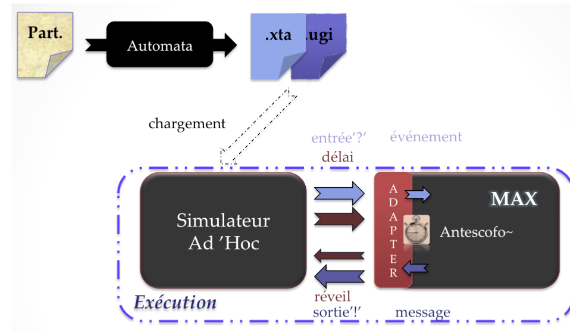


FIGURE 27: Tests à la volée 2

Génération de tests à la volée référencés sur l'horloge d'Antescofo : L'idée étant de créer un logiciel Ad'Hoc communiquant avec Antescofo. De commencer par calculer statiquement la trace idéale depuis Uppaal. Puis par un adaptateur dans Antescofo d'accéder au tempo courant pour instancier des durées relatives reçu par le logiciel de test en plus des événements discrets (identique à Tron). Ce logiciel de test implanté doit donc 'mimer' Tron mais aussi envoyer les temps relatifs à Antescofo.

hal-00868618, version 1 - 1 Oct 2013

hal-00868618, version 1 - 1 Oct 2013

Quatrième partie
Conclusion

Pour cette dernière partie un bilan va être établi, dans lequel sera rapidement résumé le stage. Nous commencerons par ses bons et mauvais côtés pour ensuite lister les avantages et inconvénients de chaque méthode de génération de test identifiée ci-dessus. Nous pourrons finalement terminer ce document.

9 Bilan

Ce stage concerne un contexte intéressant et touche également un point nouveau de l'informatique. Il a demandé beaucoup de pré-requis mais cela n'a que renforcé l'esprit de recherche et le défi du travail demandé. Il s'est en effet déroulé sous MAC-OS, XCode-git et l'environnement MAX, en plus des logiciels de la gamme UPPAAL. Pour les lectures, des articles sur Antescofo, les automates et automates temporisés ainsi que ceux concernant UPPAAL ont dû être étudiés. Pour continuer, de nombreux tests ont été fait ainsi que de recherches sur la méthode à employer pour la génération de test sur Antescofo. Ces difficultés ont permis une communication entre les membres de l'équipe et un approfondissement de ce sujet complexe et inachevé. Le problème du temps est donc à nouveau la limite de ce stage, qui laisse en suspend les implantations faites et à faire. Je vais cependant continuer ces travaux en thèse : ***Méthodes formelles d'analyse d'interactions Homme-système dans des scénarios temporels complexes*** pour approfondir ce stage et ses contributions.

9.1 Avantages et plus

Pour ce bilan une liste des avantages des méthodes étudiées est faite. Elle s'organise en tirant les avantages de la méthode de génération en général puis des implantations que nous en avons fait : **Offline** :
- Touche la conservation des œuvres (RIM)
- Contexte déterministe (contrainte non lourde pour nous)
- Adaptateurs simples

1) Horloge Antescofo :
- Génération automatique de plusieurs Cas de test (via coVer)
- Abstraction du temps (temps relatif uniquement)
- Utilisation de coVer.
- Option "play" possible pour l'exécution des cas de test (simule le temps sans attendre) : accélération des tests

2) Horloge Arbitraire :
- Système de test plus léger
- Utilisation de la trace idéale
- Courbe de tempo : Définition du tempo du musicien lors de la performance

– Test de l'Horloge d'Antescofo (car utilisation d'une Horloge arbitraire)

Online :
– Test en temps réel (plus adapté au contexte) : simulation d'un 'vrai' musicien
– Verdict et réaction d'Antescofo obtenus directement
– Tests illimités

1) Horloge Arbitraire :
– Possibilité d'intervenir dans les tests (ou de les faire soi-même (entrées ordinaires du logiciel toujours opérationnelles))
– Abstraction du temps possible (Serveur Horloge)
– Utilisation de Tron. (traite la gestion des problèmes de temps (notamment des latences))
– Courbe de tempo : Définition du tempo du musicien lors de la performance

2) Horloge Antescofo :
– Traduction 'personnalisée' de l'automate temporisé (gestion de l'algorithme)
– Passage de temps relatif (Abstraction du temps)

9.2 Difficultés

De la même manière les désavantages sont listés ci-après :
Offline :
– Tests en 'deux temps'.

1) Horloge Antescofo :
– Choix des délais de coVer inapproprié (le délai de la première prochaine sortie doit être choisie au maximum)
– Système de test assez complexe
– Problème d'automatisation du lancement de MAX

2) Horloge Arbitraire :
– Pas de génération d'une suite de test mais d'un cas de test

Online :
– Gestion de l'adaptateur via MAX compliquée
– Version stand-alone du système non terminée

1) Horloge Arbitraire :
– Pas d'envois de temps relatif pour Tron
– Intégration de la courbe de tempo à la volée ou dans l'automate

2) Horloge Antescofo :
– A créer entièrement (long)

hal-00868618, version 1 - 1 Oct 2013

hal-00868618, version 1 - 1 Oct 2013

10 Conclusion

La musique mixte se prête très bien au domaine du test temps-réel et la croisée des ces deux domaines constitue un véritable défi. Les automates temporisés sont bien pensés pour la gestion de systèmes distribués dans un contexte de deadlines mais ont encore quelques problèmes dans la gestion du temps relatifs (notamment pour Tron et son instanciation par Unité de Modèle de Temps (mtu)). CoVer gère très bien cela en gardant relatif les délais du modèle mais génère ses suites de tests avec un algorithme ne considérant pas le plus petit délai avant la première sortie possible (peut être faut-il ajouter des transitions eagers aux modèles d'UPPAAL *arrêtant l'écoulement du temps* dès qu'elle est valide).

Nous avons vu comment l'on peut procéder à des générations de tests en temps réel sur un système de musique mixte. Quels étaient l'état de l'art et solutions existantes pour essayer de les appliquer à notre contexte. Plusieurs points critiques ont été rencontrés et nous avons fini, dans la partie contribution, par décrire de nouvelles méthodes à développer plus appropriées pour cette vision encore nouvelle dans le monde du test. Ceci met en avant la notion de temps en informatique, qui n'a pas l'habitude d'être considéré comme une entité de base mais comme une ressource ordinaire que l'on doit le moins utiliser. Un article CPS [Lee09] parle de ce sujet et critique cette façon de penser qui n'est pas la bonne pour les systèmes temps réel embarqués. A t-il raison de juger ainsi la pensée informatique et ce document est il une preuve supplémentaire pour montrer qu'elle doit être repenser pour le temps ?

hal-00868618, version 1 - 1 Oct 2013

Cinquième partie

Remerciements

Des remerciements particuliers à Florent Jacquemard pour avoir pris de son temps et m'avoir suivi et aidé durant les travaux de cette période de stage. Également au reste de l'équipe MUTANT : - Arshia Cont, Jean-Louis Giavitto, Jose Echeveste, Thomas Coffy et Philippe Cuvillier -, pour son accueil, sa participation et sa bonne entente. Enfin, merci aux personnes du bureau B43 et à tout l'IRCAM.

Références

[AHS08] Marius Mikucionis Brian Nielsen Paul Pettersson Anders Hessel, Kim G. Larsen and Arne Skou. *Testing Real-time systems using UPPAAL*, volume 4949 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008. Department of Information Technology, Uppsala University et Department of Computer Science, Aalborg.

[JE11] Jean-Louis Giavitto Florent Jacquemard José Echeveste, Arshia Cont. Operational semantics of a domain specific language for real time musician–computer interaction. *Discrete Event Dynamic Systems*, 23(4) :343–383, 2011.

[JS08] Jan Tretmans Julien Schmaltz. *On Conformance Testing for Timed Systems*, volume 5215 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2008.

[KGL09] Brian Nielsen Kim G. Larsen, Marius Mikučionis. Uppaal tron user manual. *CISS, BRICS, Aalborg University, Aalborg, Denmark*, 2009.

[Lee09] Edward A. Lee. Computing needs time. *COMMUNICATIONS OF THE ACM*, 52(5), 2009. University of California, Berkeley.

[LF13] Florent Jacquemard Léa Fanchon. Formal timing analysis of mixed music scores. *ICMC - International Computer Music Conference (2013)*, 2013. STMS Lab (IRCAM, CNRS, UPMC, INRIA) - RepMus/MuSync team.

[NB12] Amélie Stainer Moez Krichen Nathalie Bertrand, Thierry Jéron. *Off-line test selection with test purposes for non-deterministic timed automata*, volume 6605 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012.

[Rol11] Antoine Rollet. Model based testing : principes et applications dans le cadre temporisé. *Université de Bordeaux (LaBRI - CNRS UMR 5800)*, 2011.

hal-00868618, version 1 - 1 Oct 2013

Sixième partie
annexes

En annexe est présenté un exemple de test. Celui-ci suit l'état en fin de stage de la génération de test de conformité différée implantée. L'on montre ici les différentes versions des fichiers obtenus, ces fichiers sont transformés pour une meilleure lisibilité.

Voici image 28 la partition choisie pour cet exemple. La ligne de temps en bas du fichier décrivant le comportement que devrait avoir Antescofo sur cette partition.

```
BPM 90

NOTE D4 1.5 note1
->      GFWD 0.5 g0 global tight
        {
                0.25 Max_a 88
                ->      GFWD 0.25 g1 global tight
                        {
                                0.10 Max_a1 100
                                0.10 Max_a2 101
                        }
                0.25 Max_b 89
                0.5 Max_c 90
        }
NOTE 67 0.25 note2
NOTE 68 0.25 note3
NOTE 69 0.5  note4


E -> events (notes)
A -> actions

; 0.00   0.25 0.50   0.75   1.00 1.10  1.20 1.25 1.50 1.75 2.00    2.50   :: time
;
; e1     .      .      .      .    .     .    .    e2   e3   e4      end
; .      .     g0      .      .    .     .    .    .    .    .
; .      .      .      a     g1    .     .    b    .    c    .
; .      .      .      .      .   a1    a2    .    .    .    .
```

Figure 28: Partition

hal-00868618, version 1 - 1 Oct 2013

```
//Global Channels evts
broadcast chan e1, me2, group3, mgroup3, e5, me6, group7, mgroup7, me8, e10, e12,
me13, mgroup14, e16, e18, e20, e22;
//Global Channels actions
broadcast chan a0, a1, a2, a3, a4;

process Proxy_0() {
clock t;
state
        s00,
        s01,
        s02,
        s03,
        s04;
init s00;
trans
                // 1 [note1 NOTE 6200 3/2]
        s00 -> s01 { sync e1?; assign t :=0; },
                // 5 [note2 NOTE 6700 1/4]
        s01 -> s02 { sync e5?; assign t :=0; },
                // 10 [note3 NOTE 6800 1/4]
        s02 -> s03 { sync e10?; assign t :=0; },
                // 12 [note4 NOTE 6900 1/2]
        s03 -> s04 { sync e12?; assign t :=0; };
}

process Top_2() {
clock t;
state
        s00,
        s01,
        s02,
        s03,
        s04;
init s00;
trans
                // 2 [mnote1 NOTE 6200 3/2]
        s00 -> s03 { sync me2?; assign t :=0; },
                // 1 [note1 NOTE 6200 3/2]
        s00 -> s01 { sync e1?; assign t :=0; },
                // 3 [group g0 = note1.1]
        s01 -> s02 { guard t == 50 ; sync group3!; assign t :=0; },
                // 14 [mgroup g0 = note1.2]
        s03 -> s04 { sync mgroup14!; assign t :=0; };
}

[...]

system Proxy_0, Env_1, Top_2, TightGlobalGrp_3, TightGlobalGrp_4;
```

Figure 29: Automates d'UPPAAL .xta

Image 29 illustre le fichier décrivant une partie du réseau d'automates temporisés d'UPPAAL (fichier xta) découlant de la partition. L'on peut voir ses 3 parties : - les définitions des variables globales, les channels en mode broadcast, permettant de lier les synchronisations entre les automates temporisés du réseaux (I/O), broadcast veut dire qu'une transition de synchronisation en sortie (!) fait emprunter toutes les transitions d'entrées de même label (?), nous avons ensuite la déclaration des automates temporisés et la déclaration du système général -.

hal-00868618, version 1 - 1 Oct 2013

Nous obtenons image 30 la partition sous test. La version avec trace de la partition d'entrée (créée depuis l'AST du compilateur).

```
// Score under test generated from the original score
// Score with a simple form and automata's labels
traceout_0 "START of traces at " $NOW " tempo = " $RT_TEMPO "\n\n"
// -----------------------------------------------------------------
NOTE 6200 1.5 e1
 traceout_0 e1 " now=" $NOW " tempo=" $RT_TEMPO " rtime=" $RNOW " \n"
  0.5 group g0 @tight{
    0.25 traceout_0 "    " a0 " now=" $NOW " tempo=" $RT_TEMPO " rtime=" $RNOW " \n"
    0.25 group g1 @tight{
      0.1 traceout_0 "    " a1 " now=" $NOW " tempo=" $RT_TEMPO " rtime=" $RNOW "
\n"
      0.1 traceout_0 "    " a2 " now=" $NOW " tempo=" $RT_TEMPO " rtime=" $RNOW "
\n"
    }

    0.25 traceout_0 "    " a3 " now=" $NOW " tempo=" $RT_TEMPO " rtime=" $RNOW " \n"
    0.5 traceout_0 "    " a4 " now=" $NOW " tempo=" $RT_TEMPO " rtime=" $RNOW " \n"
  }


NOTE 6700 0.25 e5
 traceout_0 e5 " now=" $NOW " tempo=" $RT_TEMPO " rtime=" $RNOW " \n"

NOTE 6800 0.25 e10
 traceout_0 e10 " now=" $NOW " tempo=" $RT_TEMPO " rtime=" $RNOW " \n"

NOTE 6900 0.5 e12
 traceout_0 e12 " now=" $NOW " tempo=" $RT_TEMPO " rtime=" $RNOW " \n"

  traceout_0 "STOP ##### " $NOW " \n"
```

FIGURE 30: Partition sous test

hal-00868618, version 1 - 1 Oct 2013

L'image 31 suivante est une trace donnée par coVer. Nous allons prendre la trace 4 comme exemple d'illustration.

```
===== Trace #4====================================
 edgeN<edgeid Env_1.s00_to_s01>  edgeN<edgeid Env_1.s01_to_s02>  edgeN<edgeid
Env_1.s02_to_s03>  edgeN<edgeid Env_1.s02_to_s02>  edgeN<edgeid Env_1.s02_to_s02>
edgeN<edgeid Env_1.s03_to_s03>  edgeN<edgeid Env_1.s03_to_s03>  edgeN<edgeid
Env_1.s03_to_s03>
State:
( Proxy_0.s00 Env_1.s00 Top_2.s00 TightGlobalGrp_3.s00 TightGlobalGrp_4.s00 )
Proxy_0.t=0 Env_1.t=0 Top_2.t=0 TightGlobalGrp_3.t=0 TightGlobalGrp_4.t=0

Transitions:
  Env_1.s00->Env_1.s01 { t == 0, e1!, t := 0 }
  Proxy_0.s00->Proxy_0.s01 { 1, e1?, t := 0 }
  Top_2.s00->Top_2.s01 { 1, e1?, t := 0 }

State:
( Proxy_0.s01 Env_1.s01 Top_2.s01 TightGlobalGrp_3.s00 TightGlobalGrp_4.s00 )
Proxy_0.t=0 Env_1.t=0 Top_2.t=0 TightGlobalGrp_3.t=0 TightGlobalGrp_4.t=0

Delay: 50

State:
( Proxy_0.s01 Env_1.s01 Top_2.s01 TightGlobalGrp_3.s00 TightGlobalGrp_4.s00 )
Proxy_0.t=50 Env_1.t=50 Top_2.t=50 TightGlobalGrp_3.t=50 TightGlobalGrp_4.t=50

Transitions:
  Top_2.s01->Top_2.s02 { t == 50, group3!, t := 0 }
  TightGlobalGrp_3.s00->TightGlobalGrp_3.s01 { 1, group3?, t := 0 }

State:
( Proxy_0.s01 Env_1.s01 Top_2.s02 TightGlobalGrp_3.s01 TightGlobalGrp_4.s00 )
Proxy_0.t=50 Env_1.t=50 Top_2.t=0 TightGlobalGrp_3.t=0 TightGlobalGrp_4.t=50

Delay: 100

State:
( Proxy_0.s01 Env_1.s01 Top_2.s02 TightGlobalGrp_3.s01 TightGlobalGrp_4.s00 )
Proxy_0.t=150 Env_1.t=150 Top_2.t=100 TightGlobalGrp_3.t=100
TightGlobalGrp_4.t=150

Transitions:
  Env_1.s01->Env_1.s02 { t == 150, e5!, t := 0 }
  Proxy_0.s01->Proxy_0.s02 { 1, e5?, t := 0 }
  TightGlobalGrp_3.s01->TightGlobalGrp_3.s09 { 1, e5?, t := 0 }
```

Figure 31: Trace4 générée par coVer

hal-00868618, version 1 - 1 Oct 2013

Les partitions conductrice (figure 32) et patronne (figure 33) découlant de la trace précédente.

```
// Score-driver genered from a coVer trace 4
// ------------------------------------------------------------------
Group Driver
{
// Traces in a file (output file)
openoutfile traceout_4"/tmp/13-Output_4.txt"

//      START
0 antescofo::nextevent

//      NOTE e1 1.5
   1.5 antescofo::nextevent

//      NOTE e5 0.25
   0.25 antescofo::nextevent

//      NOTE e10 0
//      0 END of Part

   5 closefile traceout_4
}
```

FIGURE 32: Partition conductrice

```
// Score-pattern generated from the coVer trace
// the tempo will be found on the Antescofo output trace
// To facilitate the computation we mark the beat position of events and actions since
the beggining of the test

START of traces

NOTE e1 0 duration=1.5
NOTE e5 1.5 duration=0.25
  a0 1.55
  a1 1.65
NOTE e10 1.75 duration=0
  a2 1.75
  a3 1.75
  a4 1.75
```

FIGURE 33: Partition patronne

hal-00868618, version 1 - 1 Oct 2013

La partition de trace obtenue après exécution d'Antescofo. (Image 34)

```
START of traces at 84.0233 tempo = 60.0

e1 now=85.5233 tempo=60.0 rtime=0.0
e5 now=85.7733 tempo=63.0 rtime=1.5
e10 now=85.7733 tempo=67.0 rtime=1.75
  a0 now=85.7733 tempo=67.0 rtime=1.75
  a1 now=85.7733 tempo=67.0 rtime=1.75
  a2 now=85.7733 tempo=67.0 rtime=1.75
  a3 now=85.7733 tempo=67.0 rtime=1.75
  a4 now=85.7733 tempo=67.0 rtime=1.75
```

FIGURE 34: Trace d'Antescofo

Pour finir le verdict de la comparaison des traces d'Antescofo et patronnes. (35)

```
... Verification on air ... 13/13-Output_4.txt
           Aout             Pattern
         a0   1.75          a0   1.55     x
         a1   1.75          a1   1.65     x
         a2   1.75          a2   1.75
         a3   1.75          a3   1.75
         a4   1.75          a4   1.75
Error :: Test KO
              ... the end
```

FIGURE 35: Verdict