Université de Franche-Comté **UFR Sciences et Techniques**

Numéro d'ordre : 1189 **École doctorale SPIM**

Contributions à la vérification incrémentale des systèmes temporisés à composants

THÈSE

présentée et soutenue publiquement le 7 décembre 2006

pour l'obtention du grade de

Docteur de l'université de Franche-Comté

(Spécialité Informatique)

par

Emilie Oudot

Composition du jury

Directeurs :	Jacques Julliand, Professeur à l'Université de Franche-Comté, LIFC Hassan Mountassir, Professeur à l'Université de Franche-Comté, LIFC
Rapporteurs :	Olivier Roux, Professeur à l'Ecole Centrale de Nantes, IRCCyN Jeanine Souquières, Professeur à l'Université Nancy 2, LORIA Stavros Tripakis, Chargé de recherche CNRS, VERIMAG
Examinateurs :	Françoise Bellegarde, Professeur à l'Université de Franche-Comté, LIFC Radu Mateescu, Chargé de recherche, INRIA Rhône-Alpes

Laboratoire d'Informatique de l'Université de Franche-Comté — CNRS FRE 2661

Mis en page avec la classe thloria.

Remerciements

Je tiens en premier lieu à remercier Monsieur Hassan Mountassir, responsable de thèse, pour m'avoir permis d'effectuer cette thèse, et pour la confiance qu'il m'a accordée durant ces trois années. Cette thèse n'aurait jamais pu aboutir sans Monsieur Jacques Julliand, co-responsable de thèse, pour sa disponibilité, ses conseils et ses remarques pertinentes sur ce travail.

J'aimerais également remercier tous les membres du jury. Merci à Monsieur Olivier Roux, Madame Jeanine Souquières et Monsieur Stavros Tripakis pour l'intérêt porté à mes travaux en acceptant de rapporter cette thèse. Merci également à Monsieur Radu Mateescu pour avoir accepté de faire parti de ce jury.

Je tiens tout particulièrement à exprimer ma reconnaissance à Madame Françoise Bellegarde, présidente de ce jury, pour tout ce qu'elle m'a apporté durant cette thèse. Je ne la remercierai jamais assez pour le temps qu'elle m'a accordée, pour son soutien et ses conseils avisés, qui m'ont beaucoup appris.

Je remercie également Mesdames Sylvie Damy, Isabelle Jacques et Olga Kouchnarenko, ainsi que Messieurs Jacques Julliand et Hassan Mountassir, pour la confiance qu'il m'ont accordée en me permettant d'effectuer des activités d'enseignement.

J'aimerais remercier tous les membres du laboratoire pour leur accueil, et l'ambiance conviviale qui y règne. J'aimerais remercier en particulier Pierre-Cyrille pour avoir accepté de prendre du temps pour répondre à mes (nombreuses) questions !

J'ai une pensée particulière pour tous ceux qui ont été présents et m'ont aidé par leur amitié, dans les moments joyeux comme dans les moments plus difficiles : Fab, Yoh, Mathilde, Steph, Nicolas "Nickles", Francky, Jib, Ju, Fred, Eddie, Nicolas "Père Pain"...

Enfin, je tiens à remercier toute ma famille, en particulier ma maman et ma grand-mère, pour le soutien apporté durant ces trois années de thèse. Qu'ils trouvent ici le témoignage de ma plus profonde reconnaissance.

Table des matières

Partie I Contexte Scientifique

Table des figures

Liste des tableaux

1

Introduction

Sommaire

La composante logicielle est de plus en plus présente dans les systèmes qui nous entourent, et ceci dans de nombreux domaines : automobile, télécommunications, avionique, aérospatiale, médical, etc. On comprend alors pourquoi la nécessité de concevoir des logiciels sûrs est devenue un enjeu majeur. Une défaillance d'ordre logiciel peut avoir de lourdes conséquences, autant sur le plan financier que sur le plan humain, comme l'attestent de (trop) nombreux exemples survenus par le passé.

Dans le domaine médical, on peut citer le cas du Therac-25. Le Therac-25 était un appareil de traitement du cancer par radiations, ayant provoqué la mort de plusieurs personnes entre 1985 et 1987, du fait d'une surdose de radiations envoyées au patient. Lorsque les zones cancéreuses étaient superficielles, un faible rayonnement d'électrons était émis. Dans le cas de zones plus profondes, un traitement par rayons X était administré. Pour cela, un plus fort rayonnement d'électrons était émis et absorbé par un dispositif (appelé la *cible*) qui émettait ensuite les rayons X. La présence de la cible était primordiale afin d'éviter au patient de recevoir une surdose d'électrons. Les accidents se sont produits lorsque le rayonnement fort a été déclenché par le logiciel alors que la cible n'était pas en place. De plus, le moniteur de contrôle indiquait dans ce cas qu'une sous-dose de radiations avait été envoyée. Le technicien en charge de l'appareil relançait donc la procédure de radiations. Après enquête, il s'est avéré que le problème apparaissait quand tous les paramètres du traitement étaient entrés dans le terminal, et que l'opérateur modifiait ensuite trop vite le choix du type de traitement à effectuer (de rayons X à électrons). L'opérateur ne faisait ensuite que valider les autres paramètres, qui avaient déjà été saisis. Lorsque cette opération était trop rapide, l'appareil recevait des valeurs aléatoires, et des surdoses de radiations pouvaient être envoyées au patient, sans que la cible soit en place. Ces accidents étaient dûs à un dysfonctionnement du logiciel de contrôle du Therac-25.

Dans le domaine aérospatial, on peut citer la mission Mars Exploration Rovers de la NASA. Cette mission consistait à envoyer sur Mars deux robots explorateurs, Spirit et Opportunity, pour y collecter des informations sur la planète. En janvier 2004, Spirit est arrivé sur Mars. Trois semaines après son arrivée, le robot s'immobilise et coupe par intermittence ses communications avec la Terre. Son système se met alors à redémarrer en boucle. Le redémarrage était dû à un problème dans le logiciel de gestion de la mémoire flash de Spirit. Si, dans le cas de Spirit, le problème logiciel a pu être réglé et n'a pas eu de conséquences importantes, ce n'a pas été le cas lors du lancement de la première fusée Ariane 5 en 1996, qui s'est auto-détruite au bout de 37 secondes de vol. Le problème était dû au logiciel de contrôle de vol, et plus particulièrement dans le système de référence inertielle, qui a cessé de fonctionner.

Tous ces exemples montrent à quel point ces erreurs logicielles peuvent avoir des conséquences désastreuses : l'appareil Therac-25 a provoqué la mort de 5 personnes, tandis que d'autres ont gardé de sérieuses séquelles. L'échec d'Ariane 5 a provoqué 3 milliards de francs de pertes. Le coût de la mission Mars Exploration Rovers s'élevait à environ 800 millions de dollars. On imagine donc assez aisément la perte financière qu'un échec de la mission aurait pu engendrer.

Pourtant, le développement et la validation de ces logiciels sont soumis à de fortes exigences afin d'en assurer un bon fonctionnement. Un cahier des charges est d'abord rédigé afin de définir et de décrire le fonctionnement du logiciel et les besoins. Une analyse est alors réalisée à partir de ce cahier des charges, et le logiciel est ensuite développé, puis soumis à une intense phase de tests pour sa validation. Lors d'un tel processus de développement, on compte en particulier deux raisons pouvant potentiellement mener à

des erreurs. Tout d'abord, le cahier des charges peut être une source d'ambiguités. En effet, les personnes qui le rédigent ne sont généralement pas les mêmes que celles qui développent le logiciel. Il peut donc y avoir des erreurs d'interprétation du cahier des charges, se répercutant directement dans le logiciel. Concernant la phase de validation, même si elle est indispensable, elle peut se révéler insuffisante pour garantir totalement le bon fonctionnement du logiciel. En effet, on imagine assez aisément que, pour des logiciels complexes, il est impossible de tester de manière exhaustive tous leurs cas d'utilisation et de fonctionnement.

Afin de réduire ces risques d'erreurs, et ainsi de permettre d'accroître la confiance que l'on peut avoir dans le logiciel, d'autres techniques doivent être utilisées lors de son développement, en complément de la phase de tests, pour en assurer le bon fonctionnement : les méthodes formelles. On retrouve d'ailleurs cette recommandation, généralisée au développement des logiciels médicaux, dans le rapport relatant le problème logiciel du Therac-25 [LT93] :

"The software should be subjected to extensive testing and formal analysis at the module and software level; system testing alone is not adequate."

ainsi que dans le standard de développement de logiciels critiques de l'ESA (Agence Spatiale Européenne) [Age91], qui préconise également :

"Formal methods should be considered for the specification of safety-critical systems"

1.1 Les méthodes formelles

Les *méthodes formelles* rassemblent des techniques basées sur les mathématiques utilisées pour la spécification, le développement et la vérification de systèmes et logiciels informatiques. L'utilisation de méthodes formelles dans le développement de logiciels permet de disposer d'un formalisme de spécification avec une syntaxe et une sémantique précise, et d'une procédure, au moins partiellement automatisée, pour vérifier que la spécification est cohérente.

1.1.1 De la spécification du système...

Comme nous l'avons dit précédemment, une première source d'erreurs dans les logiciels réside dans une potentielle mauvaise compréhension du cahier des charges. L'utilisation de méthodes formelles requiert de donner une spécification formelle, basée sur les mathématiques, au système traité, permettant ainsi d'exprimer de manière non ambiguë les besoins et le système.

Le choix du formalisme à utiliser pour la spécification du système dépend en grande partie du type de système considéré (fini ou infini, séquentiel, distribué, concurrent, temps-réel, etc), mais également de la nature des vérifications qui devront être effectuées. Etant donné le grand nombre de formalismes existants, nous n'allons pas en donner ici une liste

exhaustive[1]. Toutefois, on peut dire que l'on retrouve classiquement les formalismes de spécification classés en deux catégories : ceux orientés propriétés et ceux orientés modèle.

Spécifications orientées propriétés.

Dans ce type de spécifications, le comportement du système est décrit de manière indirecte, en donnant un ensemble de propriétés que le système doit satisfaire. Ces propriétés sont en général représentées par un ensemble d'axiomes. On distingue deux approches dans ce type de spécifications : l'approche axiomatique ou l'approche algébrique. Les formalismes OBJ[Gog79] et Larch[GHG+93] sont des exemples d'approche algébrique.

Spécifications orientées modèle.

Dans les spécifications orientées modèle, le comportement du système est défini en construisant un modèle de ce système. La construction de ce modèle est fondée sur deux notions complémentaires : une modélisation statique, décrivant les entités qui constituent le système et les états qui leur sont associées, et une modélisation dynamique, représentant les changements d'états du système à l'aide d'actions.

Pour les systèmes séquentiels, on peut citer parmi les formalismes orientés modèle utilisés, B [Abr96a], VDM [Jon90] ou encore Z [Spi87]. Dans le cadre des systèmes concurrents, citons les réseaux de Petri [Esp97, JM95] ou encore les algèbres de processus[2] tels que CSP [Hoa85], CCS [Mil89] ou ACP [BK84]. Un formalisme de base et permettant de représenter tout système est celui des systèmes de transitions [Arn92]. Il est d'ailleurs en général utilisé comme modèle sémantique des formalismes orientés modèle, tels que ceux que nous avons cités précédemment. Un système de transitions est composé d'états, donnant à chaque instant la valeur des variables du système, et de transitions, modélisant les changements d'états et représentant les évolutions atomiques du système.

1.1.2 ... A la vérification de cette spécification

La vérification consiste à garantir qu'une propriété devant être respectée par le système l'est effectivement sur la spécification du système. On distingue deux techniques permettant d'effectuer cette vérification : la preuve et le model-checking.

Preuve

Avec les techniques de preuve, le système et la propriété à vérifier sont exprimés dans le même langage mathématique de spécification. Cette méthode de vérification consiste alors à démontrer que les propriétés sont satisfaites par le système, en utilisant des règles de déduction. On compte de très nombreux outils de preuve, qu'ils soient automatisés ou

[1] Voir cependant les sites `http ://vl.fmnet.info/` et `http ://www.fmeurope.org/`, qui fournissent un bon aperçu des formalismes existants.

[2] Les algèbres de processus sont parfois considérés comme une troisième famille de formalismes de spécification.

interactifs. On peut citer notamment Simplify [DNS05], haRVey [DR03], Mona [HJJ+96], l'Atelier B [Abr96a], PVS [ORS92], HOL [GM93] ou encore Coq [Tea01].

Si elle permet de traiter un large éventail de systèmes, en particulier les systèmes infinis, cette technique possède l'inconvénient principal de ne pas être toujours automatique, et de nécessiter dans certains cas l'intervention d'un utilisateur expert pour guider la preuve.

Model-checking

Contrairement à la preuve, le model-checking est une technique de vérification algorithmique totalement automatique, en général appliquée sur des modèles du type système de transitions (finis). Les premiers algorithmes de model-checking ont été introduits au début des années 80 indépendamment par Clarke et Emerson d'un côté [CE81] et Queille et Sifakis [QS82] de l'autre. Les propriétés du système sont souvent exprimées par des formules d'une logique temporelle. Parmi les plus connues, on peut citer la logique linéaire LTL (Linear Temporal Logic) [Pnu81], ou encore la logique arborescente CTL (Computational Tree Logic) [CE81, EH82, QS82].

La vérification par model-checking est basée sur une énumération exhaustive de l'espace d'états du modèle, permettant ainsi de s'assurer que la propriété est vérifiée par tous les enchaînements d'états que le système peut effectuer. Cette méthode de vérification offre ainsi des garanties totales quant à la conclusion qu'elle donne, à savoir si la propriété est vérifiée ou non. Un autre avantage du model-checking est sa capacité à fournir des contre-exemples lorsque la vérification de la propriété échoue. Ce contre-exemple est constitué de la séquence d'états du modèle qui ne respecte pas la propriété et permet donc d'indiquer quel comportement du modèle ne la vérifie pas.

Malgré tous ces avantages, le principe même de la méthode (l'énumération exhaustive de l'espace d'états) en constitue un inconvénient majeur. Tout d'abord, le model-checking est uniquement applicable aux modèles à nombre d'états fini. Le deuxième problème concerne l'applicabilité de la méthode en pratique. En effet, le model-checking s'avère être difficile à utiliser lorsque les systèmes à traiter possèdent un grand nombre d'états. Ce problème est plus connu sous le nom d'explosion combinatoire. Afin d'améliorer l'applicabilité de la méthode en pratique, différentes techniques ont été étudiées. On peut citer notamment

- les techniques de compression d'états, comme le *bitstate hashing* [Hol97b, Hol98],
- l'utilisation de BDDs comme représentation symbolique d'ensembles d'états [Bry86],
- l'utilisation de techniques d'abstraction pour réduire la taille du modèle du système [CGL94],
- ou encore, les techniques de réduction d'ordre partiel visant à réduire l'espace d'états exploré pour la vérification [Pel96].

Cette méthode, ainsi que les techniques citées précédemment et permettant de l'amé-

liorer, sont implantées dans de nombreux outils, appelés *model-checkers*. On peut citer notamment SPIN [Hol97a] ou encore SMV [McM93]. HYTECH [HHWT97] est un model-checker pour les systèmes hybrides. KRONOS [Yov97] et UPPAAL [LPY97a] sont des model-checkers pour les systèmes temporisés.

1.2 Contexte et problématique

Comme nous l'avons dit précédemment, les méthodes de modélisation et de vérification diffèrent suivant le type de système considéré. Ces travaux de thèse se placent dans le contexte de la vérification par model-checking de systèmes temporisés modélisés à base de composants, en utilisant des techniques de développement incrémental. Présentons donc ces trois notions essentielles autour desquelles s'articule notre travail : les systèmes temporisés, la modélisation à base de composants, et les techniques de développement incrémental.

1.2.1 Les systèmes temporisés

Un système temporisé est un système dont le comportement est soumis à de fortes contraintes de temps, comme le fait qu'une certaine action doit arriver à un certain moment ou encore que deux actions doivent impérativement être séparées par un certain délai. La modélisation de tels systèmes nécessite de pouvoir représenter de manière quantitative l'écoulement du temps et il faut donc des modèles qui intègrent ces aspects.
Il existe deux manières possibles de considérer le temps dans les modèles : en le discrétisant ou en le considérant comme étant continu. Dans le cadre du temps continu, de nombreux modèles définis dans le cadre non temporisé ont été étendus pour prendre en compte ces contraintes de temps supplémentaires : réseaux de Petri temporisés, algèbres de processus temporisées, etc. Parmi tous les modèles temporisés introduits, l'un des plus étudiés depuis son introduction au début des années 90 est celui des automates temporisés d'Alur et Dill [AD94].

De la même façon, des formalismes permettant d'exprimer les propriétés temporisées de tels systèmes ont vu le jour. A titre d'exemple, les logiques LTL et CTL, évoquées précédemment, ont été étendues en prenant en compte des aspects quantitatifs du temps, et ont donné naissance respectivement aux deux logiques temporisées MITL (Metric Interval Temporal Logic) [AFH96] et TCTL (Timed Computational Temporal Logic) [ACD93].

Dans ce contexte temporisé, le problème du model-checking pour ces deux logiques a été montré décidable. Cependant, sa complexité reste élevée. En effet, elle est accentuée par la présence des contraintes de temps, qui font grossir l'espace d'états à explorer pour la vérification. A titre de comparaison, le tableau 1.1 donne la complexité du model-checking pour LTL et CTL, et leurs versions temporisées MITL et TCTL.

Si la complexité de la vérification est plus élevée pour les systèmes temporisés, leur

	Non temporisée		Temporisée	
Logique arborescente	(CTL)	PTIME-complet	(TCTL)	PSPACE-complet
Logique linéaire	(LTL)	PSPACE-complet	(MITL)	EXPSPACE-complet

TAB. 1.1 – Comparaison de la complexité du model-checking dans les cas non temporisé et temporisé

modélisation peut également s'avérer être une tâche difficile. En effet, ce sont souvent des systèmes au comportement complexe où la composante temps est primordiale. Pour faciliter la phase de modélisation des systèmes complexes en général, mais également, nous le verrons, leur vérification, une méthode qui reçoit de plus en plus d'attention repose sur l'utilisation de composants.

1.2.2 Modélisation à base de composants

Plutôt que de considérer un système complexe directement dans son ensemble et de manière monolithique, une manière de le traiter peut être en utilisant un processus de décomposition. La décomposition consiste à découper le système en plusieurs blocs, appelés composants, qui vont ensuite interagir et communiquer. De nombreux systèmes réels sont composés par nature de plusieurs entités ou sous-systèmes fonctionnant de manière parallèle : protocoles de réseaux, gestionnaires de processus, etc. La décomposition peut donc guidée par la nature même de ces systèmes.

Dans la même optique, un autre point de vue est de construire un système complexe à partir de composants existants. L'idée sous-jacente à cette démarche est que les composants doivent être réutilisables. Un composant construit initialement pour être intégré dans un certain système doit pouvoir être intégré dans d'autres systèmes.

Modéliser des systèmes à base de composants consiste alors à modéliser chacun des composants, puis à les assembler en utilisant un certain opérateur de composition : on obtient ainsi le modèle complet du système. On trouve de nombreux opérateurs de composition dans la littérature, notamment dans la littérature des algèbres de processus, chacun utilisant un paradigme de communication et de synchronisation différent. Dans le cadre temporisé, les opérateurs de composition doivent également définir la manière dont le temps doit se synchroniser.

Les aspects concernant l'inter-opérabilité des composants constituent un large secteur de recherche dans ce domaine, afin de garantir la compatibilité entre les composants, dans le cadre de leur réutilisation. Néammoins, l'aspect "comportement" et fonctionnel du composant reste également important. Le fait de modéliser à base de composants introduit de nouvelles exigences en termes de vérification. En effet, il faut non seulement s'assurer que le système global est correct, mais également que le comportement de chaque composant est bien celui attendu. En effet, l'objectif dans l'utilisation de composants est tout d'abord

de simplifier la phase de modélisation d'un système donné, mais également d'assurer qu'ils s'intègrent correctement dans divers systèmes lorsqu'ils sont réutilisés.

Les modèles à base de composants possèdent deux grandes classes de propriétés à vérifier : des propriétés dites globales, portant sur le fonctionnement du modèle complet du système, et des propriétés locales aux composants, ne concernant que le comportement des composants ou de groupes de composants. Dans les deux cas, la méthode classique de model-checking consiste à vérifier ces propriétés sur un modèle global du système. Or, nous avons vu précédemment que cette méthode est difficilement applicable en pratique, et que ces difficultés sont encore accentuées dans le cadre des systèmes temporisés.
Pour palier à cela, des méthodes alternatives doivent être utilisées pour faciliter la vérification. C'est dans cette optique que les méthodes de développement incrémental ont été introduites.

1.2.3 Le développement incrémental

L'idée de base des méthodes de développement incrémental est qu'il pourrait être judicieux de tirer profit du processus de modélisation pour vérifier des systèmes complexes. La modélisation incrémentale permet de modéliser un système pas à pas, en considérant initialement une vue astraite du système à modéliser, et en détaillant progressivement ce modèle. Du point de vue de la vérification, cette démarche a pour but de vérifier à chaque étape du processus les propriétés du système portant sur le niveau d'abstraction considéré. Ainsi, les modèles traités à chaque étape restent en général de taille raisonnable pour pouvoir effectuer une vérification par model-checking, sans atteindre les limites pratiques dûes au problème de l'explosion combinatoire.

Un premier type de développement incrémental, utilisé dans des méthodes de développement telles que la méthode B, est le raffinement. Dans le cadre des systèmes à composants, que nous considérons, la méthode par raffinement peut également être utilisée, ainsi qu'une méthode plus spécifique à ce type de système : l'intégration de composants. Toutefois, ces deux méthodes ne s'adressent pas à la vérification du même type de propriétés. L'intégration de composants est plus adaptée aux propriétés locales, tandis que le raffinement est plus approprié aux propriétés globales, comme nous allons le voir dans ce qui suit.

Intégration de composants.

Considérons un ensemble de composants $C_1, C_2, \cdots, C_n$, assemblés en utilisant un opérateur de composition noté $||$. Le modèle complet du système est donc $C_1||C_2|| \cdots ||C_n$. L'intégration de composants consiste à considérer un composant (ou groupe de composants) du système, par exemple C_1, et à lui ajouter progressivement les autres composants $C_2, \cdots, C_n$. Le système complet est donc obtenu progressivement par intégrations successives des autres composants.

Dans ce type de développement incrémental, une propriété essentielle est la *composabilité*, c'est-à-dire le fait que les propriétés locales, établies à une certaine étape d'intégration, soient préservées par les intégrations suivantes. Par exemple, les propriétés de C_1, vérifiées sur C_1, doivent être préservées par l'ajout successif des $C_2, \cdots, C_n$.
Cette propriété de composabilité permet en effet de ne vérifier les propriétés locales des composants que sur les composants concernés, plutôt que sur le système complet une fois construit. Les composants étant en général de taille plus petite que le modèle complet, le model-checking est plus facilement applicable à ce niveau.

Raffinement.

Le raffinement représente une autre démarche de développement incrémental. Lorsque l'on considère un système complexe, le besoin peut se faire sentir de donner tout d'abord une spécification abstraite du système, et de la détailler petit à petit. Bien entendu, cette introduction de détails ne doit pas apporter d'incohérences par rapport à la spécification abstraite initiale. En particulier les propriétés garanties par la spécification abstraite doivent toujours être vraies après l'ajout de détails. Pour assurer cela, il faut donner un cadre formel à cette démarche d'introduction de détails : c'est le but du raffinement.

Dans le cadre des systèmes à composants, le raffinement consiste, non pas à modéliser incrémentalement le système complet, mais chacun de ses composants. Bien évidemment, garantir le raffinement entre chaque composant abstrait et sa version raffinée n'implique pas nécessairement le raffinement entre le système complet formé des composants abstraits et le système complet formé des composants raffinés. Une propriété essentielle dans cette démarche de raffinement pour les systèmes à composants est la propriété de *compositionnalité*, exprimant justement le fait que le raffinement est préservé par la composition. En d'autres termes, si les composants $C'_1, C'_2, \cdots, C'_n$ raffinent respectivement les composants $C_1, C_2, \cdots C_n$, alors le système complet $C_1||C_2|| \cdots ||C_n$ est raffiné par la composition $C'_1||C'_2|| \cdots ||C'_n$.
Cette propriété de compositionnalité permet de vérifier les propriétés globales du système sur le modèle abstrait du système, et d'assurer qu'elles sont préservées sur le système complet raffiné en vérifiant uniquement le raffinement des composants, plutôt que celui du système complet, plus coûteux à vérifier du fait de la plus grande taille des modèles complets.

Ainsi, les méthodes de développement incrémental sont une approche viable pour contourner le problème d'explosion combinatoire de la vérification par model-checking. Que ce soit par intégration de composants pour les propriétés locales aux composants, ou par raffinement pour les propriétés globales du système, le model-checking pourra être applicable dans un plus grand nombre de cas, car mis en pratique sur des modèles de plus petite taille.

1.3 Contributions et résultats principaux

L'objectif de cette thèse est de pouvoir appliquer ces méthodes de développement incrémental aux systèmes temporisés modélisés à base de composants, ainsi que d'étudier l'impact de l'utilisation de telles méthodes en pratique. Nous considérons pour cela des systèmes temporisés modélisés par des automates temporisés. Le formalisme que nous considérons pour spécifier les propriétés de ces systèmes est la logique MITL.

1.3.1 De la définition de relations de simulation temporisées...

La première contribution consiste à définir formellement le concept de développement incrémental pour les automates temporisés. Pour cela, nous utilisons des relations de τ-simulation. Nous avons donc défini deux telles relations pour les automates temporisés. La première, appelée τ-simulation temporisée, préserve les propriétés dites de *sûreté*[3]. Pour préserver un plus large spectre de propriétés, et en particulier les propriétés de *vivacité*[4], nous avons défini une seconde relation, appelée τ-simulation temporisée sensible à la divergence et respectant la stabilité (DS). Cette seconde relation préserve toutes les propriétés temporisées pouvant être exprimées à l'aide de la logique MITL. Ces relations de simulation ont été présentées dans [BJMO05].

Dans le but d'être complets, même si nous nous focalisons sur la préservation de propriétés linéaires MITL, nous avons également étudié la préservation d'autres types de propriétés, comme l'atteignabilité, l'absence de blocages et les propriétés propres aux automates temporisés telles que le non-zénonisme. Nous avons également étudié ce qu'il en est de la préservation du formalisme des automates de Büchi temporisés.

1.3.2 ... à leur utilisation lors de la modélisation incrémentale des systèmes temporisés

Rappelons que nous nous situons dans un cadre de modélisation des systèmes temporisés basée sur les composants. Nous avons présenté deux types de développement incrémental pouvant être utilisé pour ce type de modélisation : l'intégration de composants et le raffinement. Les propriétés essentielles à garantir dans ce type de développement sont celles de composabilité, de compatibilité et de compositionnalité. Nous avons donc étudié si les relations de simulations permettent de bénéficier de ces propriétés avec différents opérateurs de composition, utilisant chacun un paradigme de composition différent : le premier étant un opérateur *à la CSP*, tandis que le second est un opérateur utilisant un paradigme de composition *à la CCS*, avec des priorités entre actions.
Nous montrons que la τ-simulation temporisée DS possède les propriétés requises vis-à-vis de l'opérateur *à la CCS*, sous certaines conditions simples. Ainsi, elle est bien adaptée au développement incrémental de modèles à composants utilisant cet opérateur. En revanche, seule la τ-simulation temporisée possède ces propriétés vis-à-vis de l'opérateur *à la CSP*.

[3] Les propriétés de sûreté expriment le fait que *quelque chose de mauvais n'arrivera jamais.*
[4] Les propriétés de vivacité expriment le fait que *quelque chose finira obligatoirement par arriver.*

Ceci implique que, avec cet opérateur, seule la préservation de propriétés de sûreté peut être assurée gratuitement[5]. Les propriétés de ces relations vis-à-vis de l'opérateur *à la CSP* ont été présentées dans [BJMO05].

1.3.3 Le prototype VeSTA pour vérifier la τ-simulation temporisée DS

La τ-simulation temporisée DS ne possédant pas les propriétés requises vis-à-vis de l'opérateur *à la CSP*, les propriétés de vivacité ne sont donc pas automatiquement préservées lors d'une modélisation incrémentale. Pour assurer leur préservation, il est donc nécessaire de vérifier la τ-simulation temporisée DS. Ceci nous amène à la contribution technique de cette thèse, qui est le développement du prototype VeSTA (`Ve`rification of `S`imulations for `T`imed `A`utomata). Il permet de vérifier cette relation dans le cadre d'une modélisation incrémentale par intégration de composants. L'opérateur de composition considéré est donc l'opérateur *à la CSP*. Pour bénéficier d'une représentation mémoire la plus performante possible, VeSTA est basé sur des bibliothèques efficaces, telles que SMI[6] (Symbolic Model Interface), et utilise le module PROFOUNDER [TYB05] de l'outil OPEN-KRONOS [Tri98]. Il possède également la possibilité de connecter les modèles ainsi étudiés à la plate-forme de vérification de OPEN-CAESAR [Gar98]. Le prototype VeSTA fait l'objet du rapport [BJMO06b].

1.3.4 Des études de cas

Afin d'étudier l'apport en pratique des techniques de développement incrémental que nous avons formalisées par les simulations (nous nous focalisons en particulier sur l'intégration de composants), nous avons utilisé VeSTA pour modéliser et vérifier de manière incrémentale quelques études de cas. Nous avons comparé les résultats obtenus avec la méthode classique de model-checking. Les résultats obtenus sont encourageants en termes de temps de calcul, qui sont plus courts avec la méthode proposée qu'avec la méthode classique de vérification. De plus, ces expérimentations nous ont permis de mettre en évidence un intérêt particulier de la méthode dans le cas d'un certain type de systèmes paramétrés. Ces expérimentations sont présentées dans [BJMO06a].

1.4 Organisation du document

Ce mémoire de thèse est organisé entre trois parties.

La première partie décrit le contexte scientifique de cette thèse, en trois chapitres :

- Le chapitre 2 est consacré à la présentation des différents formalismes définis pour la modélisation des systèmes temporisés. En particulier, nous présentons le modèle des automates temporisés que nous considérons dans ce document pour modéliser

[5] ici, *gratuitement* signifie *sans vérification supplémentaire.*

[6] `http://www-verimag.imag.fr/~async/SMI/`

les systèmes temporisés. Nous en présentons les notions de base, ainsi que certaines extensions.

- Suite au chapitre précédent, le chapitre 3 est consacré à présenter les notions essentielles concernant la vérification par model-checking des systèmes temporisés. Nous présentons en particulier deux formalismes logiques permettant de spécifier leurs propriétés : MITL, qui est la logique que nous considérons principalement dans ce document, et TCTL. Nous présentons également la technique de model-checking pour chacune de ces logiques.

- Dans tout ce document, nous considérons des systèmes temporisés modélisés à base de composants. Nous introduisons donc dans le chapitre 4 les opérateurs de composition pour les automates temporisés que nous utiliserons au long de ce document. Nous présentons également en détail les deux types de modélisation incrémentale : intégration de composants et raffinement.

La deuxième partie présente les deux premières contributions de cette thèse :

- Le chapitre 5 présente les deux relations de simulation que nous avons définies sur les automates temporisés : la τ-simulation temporisée, et la τ-simulation temporisée DS. Pour chacune d'elles, nous étudions également dans ce chapitre les propriétés qu'elles préservent.

- L'utilisation de ces simulations lors de la modélisation incrémentale est étudiée dans le chapitre 6. En particulier, nous étudions les propriétés de ces relations par rapport aux deux opérateurs de composition que nous considérons, et indiquons les résultats en termes de préservation de propriétés dont peut bénéficier la modélisation incrémentale.

Enfin, la troisième partie regroupe la partie technique des contributions, c'est-à-dire l'outillage réalisé et les études de cas menées :

- Dans le chapitre 5, nous avons donné une définition sémantique des simulations. Le chapitre 7 en donne une définition symbolique, pouvant être implantée, ainsi que les algorithmes correspondant à leur vérification.

- Le chapitre 8 présente le prototype VESTA qui implante les algorithmes précédents. Nous présentons en particulier l'architecture de VESTA, ainsi que son mode d'emploi.

- Les études de cas menées pour étudier l'intérêt de la modélisation incrémentale dans le cadre des systèmes temporisés sont présentées dans le chapitre 9. Nous comparons les résultats obtenus avec ceux de la méthode de vérification classique.

Enfin, nous dressons dans la quatrième partie les conclusions et le bilan de cette thèse, et exposons les perspectives aux travaux présentés dans ce document.

Première partie

Contexte Scientifique

2

Modélisation des systèmes temporisés

Sommaire

De nombreux formalismes ont été définis dans la littérature pour modéliser des systèmes temporisés. L'utilisation d'un modèle ou d'un autre dépend du type de système à modéliser, mais également de la nature du temps considérée : temps discret ou temps continu.

Nous présentons dans ce chapitre les notions fondamentales sur la modélisation des systèmes temporisés sur lesquelles se base ce travail de thèse. Nous commençons dans la section 2.1 par expliquer cette distinction temps discret / temps continu. Dans le cadre des modèles à temps continu, que nous considérons, le formalisme le plus populaire et le plus étudié est celui des automates temporisés [AD94]. Dans la section 2.2, nous en présentons donc la définition de base sur laquelle nous nous appuyons, ainsi que certaines extensions et variantes (section 2.3). En particulier, nous présentons une variante des automates temporisés, appelée automates temporisés avec deadlines, que nous utiliserons également dans ce document. Nous terminerons par un panorama de quelques autres formalismes pouvant être utilisés pour modéliser des systèmes temporisés (section 2.4).

2.1 La nature du temps dans les modèles

Dans les systèmes temporisés, la modélisation d'un système tient compte non seulement de l'enchaînement des différentes actions lors de l'exécution du système, mais également des dates auxquelles surviennent ces actions. Dès lors, il est primordial de se demander quelle sera la nature de ces dates lors de la modélisation : seront-elles discrétisées ou non ? En d'autres termes, cela revient à se demander quelle est la nature du temps à considérer : temps discret ou temps continu. Dans le premier cas, les dates prendront leur valeur dans l'ensemble des entiers naturels $\mathbb{N}$, alors que dans le second cas, le domaine considéré sera celui des réels positifs $\mathbb{R}^+$.
Traditionnellement, nous avons une perception continue du temps à travers les moyens de mesure imaginés par l'homme (écoulement de liquide, ...). Le temps continu semble donc plus adapté à la modélisation de systèmes réels. Discrétiser le temps pour modéliser des systèmes temps-réel consiste alors à trouver une quantité de temps telle que la durée séparant deux actions du système puisse être exprimée par un multiple de cette quantité. Il semble alors difficile de trouver une telle quantité de temps, assez petite pour que le modèle soit le plus précis possible, mais également assez grande pour que la taille du modèle n'explose pas. [BS91] a d'ailleurs montré que, pour une certaine classe de circuits asynchrones, il n'est pas possible de discrétiser le temps de manière correcte.
Nous considérons dans ce document une nature du temps continue.

2.2 Les automates temporisés

Les automates temporisés sont l'un des modèles les plus étudiés et les plus utilisés pour modéliser des systèmes temps-réel quand le temps est considéré comme continu. Ils ont été introduits dans [AD94]. Ce sont des automates finis, qui utilisent des variables de type réel, appelées horloges, pour modéliser le temps.

2.2.1 Horloges et contraintes d'horloges

Commençons tout d'abord par présenter les notions de base concernant les horloges.

Valuations d'horloges

Soit $X = \{x_1, \cdots, x_n\}$ un ensemble d'horloges. Une valuation d'horloges $\mathbf{v}$ sur X, $\mathbf{v} : X \to \mathbb{R}^+$, associe une valeur réelle à chaque horloge $x_1, \cdots, x_n$ de X. On notera $\mathbf{0}$ la valuation pour laquelle toutes les horloges de X sont égales à 0. La notation $\texttt{ent}(\mathbf{v})$ désigne la partie entière d'une valuation $\mathbf{v}$ et $\texttt{fract}(\mathbf{v})$ sa partie fractionnaire.

Opérations sur les valuations d'horloges. On définit les opérations suivantes sur les valuations d'horloges. Soient une durée $t \in \mathbb{R}^+$ et $\mathbf{v}$ une valuation d'horloges sur X. La valuation $\mathbf{v}+t$ est une nouvelle valuation obtenue en ajoutant t à la valeur de chaque horloge de X dans $\mathbf{v}$. Les valuations $\mathbf{v}-t$ et $\mathbf{v}\cdot t$ sont de nouvelles valuations dans lesquelles t est respectivement soustrait ou multiplié à la valeur de chaque horloge de X dans $\mathbf{v}$. Pour un sous-ensemble $Y \subseteq X$, l'opération de remise à zéro de Y dans $\mathbf{v}$, notée $[Y := 0]\mathbf{v}$, définit une nouvelle valuation $\mathbf{v}'$ obtenue à partir de $\mathbf{v}$ en assignant la valeur 0 à toutes les horloges de Y. La valeur des autres horloges reste inchangée :

$$\forall x \cdot (x \in Y \Rightarrow \mathbf{v}'(x) = 0) \text{ et } \forall x \cdot (x \in X \backslash Y \Rightarrow \mathbf{v}'(x) = \mathbf{v}(x))$$

La projection avec restriction de dimension de $\mathbf{v}$ sur Y, notée $\mathbf{v}\rfloor_Y$, crée une nouvelle valuation $\mathbf{v}'$ sur Y ne contenant que les valeurs dans $\mathbf{v}$ des horloges de Y, $\mathbf{v}'(x) = \mathbf{v}(x)$, pour tout $x \in Y$, et les valeurs des horloges de $X \backslash Y$ n'apparaissent plus dans $\mathbf{v}'$.

Contraintes d'horloges

L'ensemble $\mathcal{C}(X)$ des contraintes d'horloges est défini par la grammaire suivante :

$$g ::= x \sim c \mid x - y \sim c \mid g \wedge g \mid \texttt{true} \text{ où } x, y \in X,\ c \in \mathbb{N}, \text{ et } \sim \in \{<, \leq, =, \geq, >\}.$$

Les contraintes de ce type sont appelées *contraintes générales*. On définit également un sous-ensemble $\mathcal{C}_{df}(X)$ de $\mathcal{C}(X)$. L'ensemble $\mathcal{C}_{df}(X)$ contient les contraintes d'horloges dites *non diagonales*. Cet ensemble n'autorise pas les contraintes appelées *diagonales*, c'est-à-dire contenant des comparaisons entre horloges, du type $x - y \sim c$. Il est défini par la grammaire suivante :

$$g' ::= x \sim c \mid g' \wedge g' \mid \texttt{true} \text{ où } x, y \in X,\ c \in \mathbb{N}, \text{ et } \sim \in \{<, \leq, =, \geq, >\}.$$

On définit la satisfaction d'une contrainte d'horloges ζ par une valuation $\mathbf{v}$, notée $\mathbf{v} \in \zeta$, de la manière suivante :

- $\mathbf{v} \in x \sim c$ si $\mathbf{v}(x) \sim c$,
- $\mathbf{v} \in x - y \sim c$ si $\mathbf{v}(x) - \mathbf{v}(y) \sim c$,
- $\mathbf{v} \in \zeta_1 \wedge \zeta_2$ si $\mathbf{v} \in \zeta_1$ et $\mathbf{v} \in \zeta_2$,
- $\mathbf{v} \in \texttt{true}$.

Notons qu'une contrainte d'horloges ζ sur X définit un ensemble de valuations sur X qui forme un polyhèdre convexe[7] sur $\mathbb{R}^X$. Nous appelons par la suite X-polyhèdre un polyhèdre sur $\mathbb{R}^X$. De la même façon que pour les valuations, on notera `zero` le X-polyhèdre défini par $\bigwedge_{x \in X} x = 0$.
Dans la suite du document, nous utilisons également les prédicats suivants sur les polyhèdres. Etant donné un X-polyhèdre ζ, le prédicat $\texttt{convex}(\zeta)$ permet de tester si ζ est convexe :

$$\texttt{convex}(\zeta) \stackrel{def}{=} \forall v, v', t \cdot (v \in \zeta \wedge v' \in \zeta \wedge 0 < t < 1 \Rightarrow t \cdot v + (1-t)v' \in \zeta).$$

Le prédicat $\texttt{unbounded}(x, \zeta)$ permet de tester si, pour une horloge $x \in X$, x est non bornée dans le X-polyhèdre ζ :

$$\texttt{unbounded}(x, \zeta) \stackrel{def}{=} \forall t \cdot (t \in \mathbb{R}^+ \Rightarrow \exists v \cdot (v \in \zeta \wedge v(x) > t)).$$

Opérations sur les polyhèdres. Les opérations de remise à zéro et de projection avec restriction de dimension définies sur les valuations peuvent être directement étendues aux polyhèdres. Considérons un ensemble d'horloges X, un sous-ensemble $Y \subseteq X$ et un X-polyhèdre ζ. L'opération de remise à zéro $[Y := 0]\zeta$ crée un X-polyhèdre ζ' tel que :

$$\zeta' = \{[Y := 0]\mathbf{v} \mid \mathbf{v} \in \zeta\}$$

La projection avec restriction de dimension du X-polyhèdre ζ sur Y, $\zeta\rfloor_Y$, est un Y-polyhèdre ζ' défini par :

$$\mathbf{v}' \in \zeta' \text{ si } \exists \mathbf{v} \cdot (\mathbf{v} \in \zeta \wedge \forall x \cdot (x \in Y \Rightarrow \mathbf{v}(x) = \mathbf{v}'(x)))$$

On définit également sur les polyhèdres les opérations de projection en avant et en arrière, respectivement notées $\nearrow \zeta$ et $\swarrow \zeta$. L'opération de projection en avant $\nearrow \zeta$ crée un nouveau X-polyhèdre regroupant toutes les valuations d'horloges qui peuvent être atteintes en laissant s'écouler le temps depuis des valuations de ζ, tandis que l'opération de projection en arrière regroupe toutes les valuations qui, en laissant s'écouler le temps, permettent d'atteindre une valuation de ζ :

$$\mathbf{v}' \in \nearrow \zeta \text{ si } \exists t \cdot (t \in \mathbb{R}^+ \wedge \mathbf{v}' - t \in \zeta)$$
$$\mathbf{v}' \in \swarrow \zeta \text{ si } \exists t \cdot (t \in \mathbb{R}^+ \wedge \mathbf{v}' + t \in \zeta)$$

Etant donnée une constante entière c, l'opération d'extrapolation $\texttt{Approx}_c(\zeta)$ crée le plus petit polyhèdre (en termes d'inclusion) ζ' incluant ζ, dont la contrainte est définie intuitivement à partir de celle de ζ de la manière suivante : toutes les bornes inférieures de ζ plus grandes que c sont remplacées par c, et toutes les bornes supérieures plus grandes que c sont ignorées.

Il est important de noter que toutes ces opérations préservent la convexité. La figure 2.1 illustre les polyhèdres calculés par ces opérations, à partir d'un polyhèdre ζ défini par la contrainte : $1 \leq x \leq 5 \wedge 2 \leq y \leq 5 \wedge x - y \leq 2 \wedge y - x \leq 3$.

[7] d'où la notation $\in$ pour la relation de satisfaction d'une contrainte par une valuation.

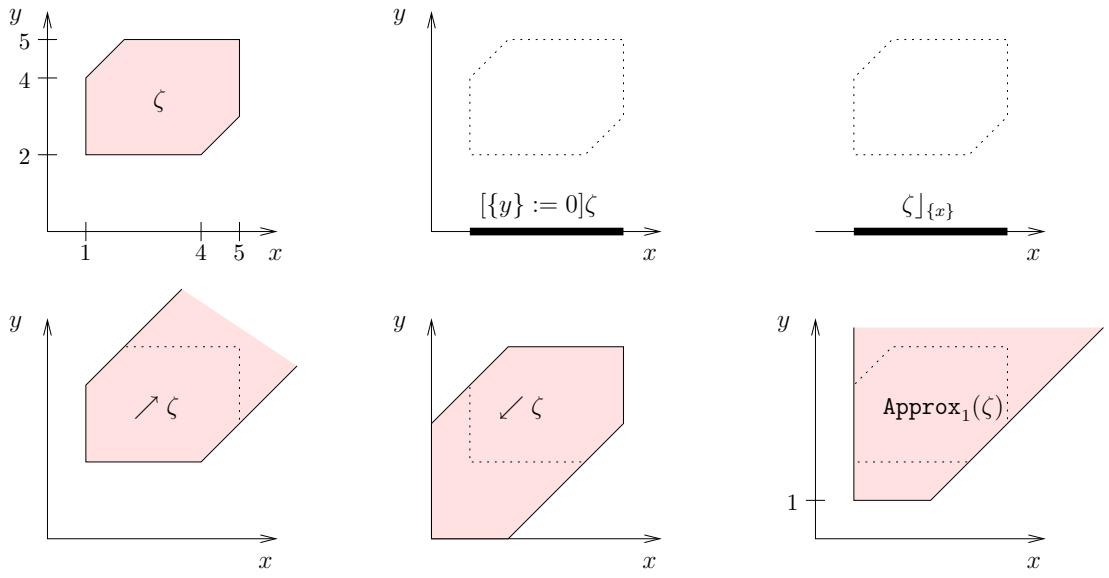


FIG. 2.1 – Opérations sur les polyhèdres

2.2.2 Syntaxe et sémantique

Un automate temporisé est un automate à nombre d'états fini, muni d'horloges qui sont utilisées pour modéliser le temps. Intuitivement, le temps s'écoule dans les états de l'automate, tandis que les transitions sont immédiates et ne prennent pas de temps.
Pour définir de quelle manière le temps peut progresser dans les états, à chaque état est associée une condition de progression du temps, appelée *invariant* et donnée sous la forme d'une contrainte d'horloges. Chaque transition de l'automate est munie d'une garde, qui est également une contrainte d'horloges, modélisant la condition d'activation des transitions. Ces transitions peuvent également remettre les horloges à zéro.

Avant de définir formellement les automates temporisés, étudions un exemple de système temporisé modélisé à l'aide d'automates temporisés. Cet exemple sera utilisé par la suite pour illustrer les résultats présentés dans ce document.

EXEMPLE 2.1. (PASSAGE À NIVEAU). Le passage à niveau est un exemple de système temporisé bien connu [Alu91]. Il prend en compte différents éléments, soumis à des contraintes de temps : un ou plusieurs trains, une barrière et un contrôleur en charge de l'ouverture et de la fermeture de la barrière. Le fonctionnement du passage à niveau avec un seul train est le suivant : au moment d'arriver au passage à niveau, le train envoie un signal au contrôleur pour l'avertir de son approche. Une unité de temps (u.t.) plus tard, le contrôleur commande la fermeture de la barrière qui doit alors se baisser dans l'u.t. qui suit, avant que le train n'entre sur le passage à niveau. Le train ne reste sur le passage à niveau qu'au plus trois u.t. Dès qu'il sort, il envoie de nouveau un signal au contrôleur qui, l'u.t. suivante, peut commander l'ouverture de la barrière. Celle-ci doit alors être levée au plus une u.t. plus tard.
La figure 2.2 représente les automates temporisés modélisant chaque élément de ce

système : le train, la barrière et le contrôleur. Chaque automate temporisé possède sa propre horloge, x pour le train, y pour la barrière et z pour le contrôleur. Chaque état est désigné par un nom (noté au dessus de l'état), que l'on appellera par la suite son décor. Les contraintes d'horloges à l'intérieur des états représentent leurs invariants. Sur chaque transition, sont notés dans l'ordre suivant : sa garde, son étiquette, et l'ensemble des horloges remises à zéro. Pour plus de lisibilité, les invariants *true* dans les états n'ont pas été inscrits, ainsi que les gardes *true* et les ensembles vides d'horloges remises à zéro sur les transitions.
Détaillons deux contraintes de temps de ce système, celles du train : le train ne doit entrer sur le passage à niveau qu'au moins deux u.t. après qu'il ait envoyé le signal d'approche, et n'y reste qu'au plus trois u.t. Autrement dit, le train doit sortir du passage à niveau au plus cinq u.t. après l'envoi du signal d'approche. L'horloge x du train est donc remise à zéro lors de l'envoi de ce signal d'approche. La garde $x > 2$ sur la transition *enter* suivante permet d'assurer la première contrainte de temps. La seconde est modélisée par l'ajout de l'invariant $x \leq 5$ dans les états *near* et *in*. Ainsi, la sortie du train du passage à niveau (la transition *exit*) sera effectuée au maximum cinq u.t. après l'envoi du signal *approach*.

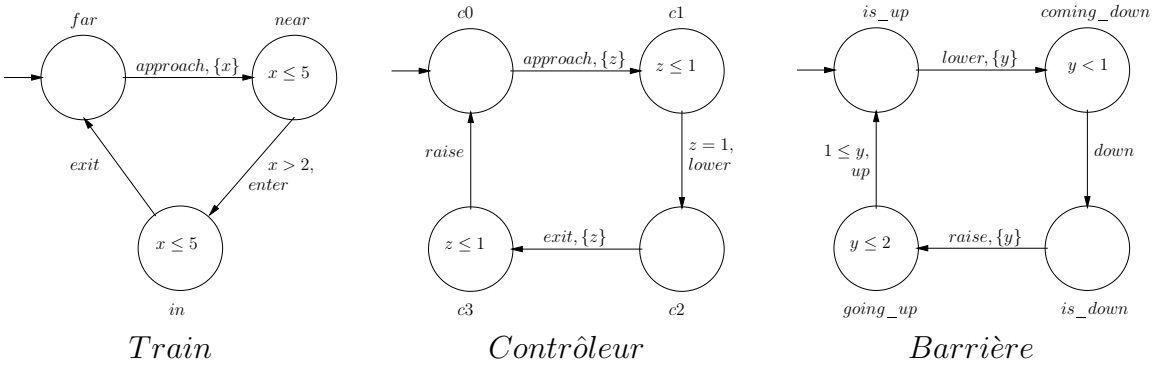


FIG. 2.2 – Automates temporisés modélisant le train, le contrôleur et la barrière

Syntaxe

Cet exemple nous a permis de présenter intuitivement les automates temporisés. Définissons les à présent de manière formelle.

DÉFINITION 1 (AUTOMATE TEMPORISÉ) *Soit* Props *un ensemble de propositions atomiques. Un automate temporisé est un t-uplet* $A = \langle \mathrm{Q}, \mathrm{q}_0, \mathrm{Labels}, \mathrm{X}, \mathrm{T}, \mathtt{Invar}, \mathrm{L} \rangle$ *où :*

- Q *est un ensemble fini d'états,*
- $\mathrm{q}_0 \in \mathrm{Q}$ *est l'état initial de l'automate,*
- Labels *est un alphabet fini de noms d'actions,*
- X *est un ensemble fini d'horloges,*
- $\mathrm{T} \subseteq \mathrm{Q} \times \mathcal{C}_{df}(X) \times \mathrm{Labels} \times 2^X \times \mathrm{Q}$ *est l'ensemble fini des transitions,*
- $\mathtt{Invar}$ *est une fonction associant une contrainte d'horloges de* $\mathcal{C}_{df}(X)$ *à chaque état, appelée invariant,*

– $\mathrm{L} : Q \to 2^{\mathrm{Props}}$ *est la fonction de décor des états, associant à chaque état un ensemble de propositions atomiques de* Props.

Chaque transition est représentée par un quintuplet $e = (q, g, a, \gamma, q')$ où q et q' sont les états source et cible de la transition, g est une contrainte d'horloges définissant la garde de la transition, a est l'étiquette de la transition et γ est l'ensemble d'horloges remises à zéro par la transition. Pour plus de lisibilité, une transition e pourra être simplement notée dans le texte par $q \xrightarrow{e} q'$. Les notations suivantes seront également utilisées : $\texttt{source}(e)$, $\texttt{guard}(e)$, $\texttt{label}(e)$, $\texttt{reset}(e)$ et $\texttt{target}(e)$ pour dénoter respectivement q, g, a, γ et q'. Etant donné un état q, les notations $\texttt{in}(q)$ et $\texttt{out}(q)$ représenteront respectivement l'ensemble des transitions ayant pour état cible q et l'ensemble des transitions ayant pour état source q. On appellera $c_{max}(A)$ la plus grande constante apparaissant dans une contrainte d'horloges (garde ou invariant) de A.

Sémantique

Du fait de la nature continue du temps, la sémantique d'un automate temporisé A est donnée par un graphe infini $\mathcal{G}(A)$. Les états de $\mathcal{G}(A)$ sont des couples (q, v), appelés *configurations* de A, où q est un état de A et v est une valuation d'horloges sur X. On notera par la suite $\texttt{disc}((q, v))$ la partie discrète q de la configuration (q, v), et $\texttt{val}((q, v))$ sa valuation v. Deux types de transitions apparaissent dans $\mathcal{G}(A)$: des transitions discrètes, représentant le déclenchement d'une action et des transitions de temps, représentant l'écoulement du temps.

Définition 2 (Graphe sémantique d'un automate temporisé) *Soit* $A = \langle \mathrm{Q}, \mathrm{q}_0, \mathrm{Labels}, \mathrm{X}, \mathrm{T}, \texttt{Invar}, \mathrm{L} \rangle$ *un automate temporisé, avec* $X = \{x_1, \cdots, x_n\}$. *Le graphe sémantique de* A *est un t-uplet* $\mathcal{G}(A) = \langle \mathrm{S}, \mathrm{s}_0, \mathrm{Labels}, \longrightarrow \rangle$ *où :*

– $S \subseteq Q \times \mathbb{R}^{+^n}$ *est l'ensemble des états du graphe. Une configuration* (q, v) *appartient à* S *si* $v \in \texttt{Invar}(q)$.
– $s_0 = (q_0, \mathbf{0})$ *est l'état initial du graphe,*
– $\longrightarrow \subseteq S \times T \cup \mathbb{R}^+ \times S$ *est l'ensemble des transitions du graphe :*
  (i) Transitions discrètes : étant donnée une transition $e = (q, g, a, \gamma, q')$ *de* A, $(q, v) \xrightarrow{e} (q', v')$ *est une transition discrète de* $\mathcal{G}(A)$, *où* $v' = [\gamma := 0]v$, *si* $v \in g$ *et* $v' \in \texttt{Invar}(q')$.
  (ii) Transitions de temps : $(q, v) \xrightarrow{t} (q, v + t)$ *est une transition de temps de* $\mathcal{G}(A)$, *pour* $t \in \mathbb{R}^+$, *si* $v + t \in \texttt{Invar}(q)$.

Une exécution d'un automate temporisé est un chemin (fini ou infini) dans son graphe sémantique. Formellement, une exécution est donc une séquence (finie ou infinie) alternant configurations et transitions de temps ou discrètes :

$$\rho = (q_0, \mathbf{0}) \xrightarrow{t_0} (q_0, v_1) \xrightarrow{e_0} (q_1, v_2) \xrightarrow{t_1} (q_1, v_3) \xrightarrow{t_2} (q_1, v_4) \xrightarrow{e_1} (q_2, v_5) \cdots$$

où $v_1 = v_0 + t_0$, $v_2 = [\texttt{reset}(e_0) := 0]v_1$, $v_3 = v_2 + t_1$, $v_4 = v_3 + t_2$ et $v_5 = [\texttt{reset}(e_1) := 0]v_4$. Notons que nous ne concaténons pas les transitions de temps successives dans les exécutions. Dans le reste du document, nous noterons (ρ, k) le k^{e} état de l'exécution ρ. La notation $\texttt{time}(\rho, (q, v))$ désigne le temps écoulé dans l'exécution ρ depuis l'état initial jusqu'à

la configuration (q, v). Par exemple, pour l'exécution ρ ci-dessus, $\texttt{time}(\rho, (q_1, v_3)) = t_0 + t_1$. La notation $\texttt{time}(\rho)$ représente le temps total écoulé dans l'exécution. L'ensemble des exécutions d'un automate temporisé A est noté $\Gamma(A)$.

EXEMPLE 2.2. Reprenons l'exemple du passage à niveau, et plus particulièrement l'automate temporisé modélisant le train. Une exécution de cet automate est la suivante :

$$(far, 0) \overset{3.5}{\rightarrow} (far, 3.5) \overset{approach}{\rightarrow} (near, 0) \overset{2}{\rightarrow} (near, 2) \overset{1}{\rightarrow} (near, 3) \overset{enter}{\rightarrow} (in, 3) \overset{0}{\rightarrow} (in, 3) \overset{exit}{\rightarrow} (far, 3) \cdots$$

Pour plus de lisibilité, nous utilisons dans cet exemple le nom des états de contrôle pour la partie discrète de chaque configuration, et nous n'avons noté sur les transitions que leur étiquette. De plus, l'automate du train ne contenant qu'une seule horloge, la valuation de chaque configuration est notée comme une valeur (celle de l'horloge x), plutôt que comme un ensemble contenant uniquement cette valeur.

Nous pouvons à présent caractériser l'ensemble des configurations atteignables d'un automate temporisé A. Une configuration (q_i, v_i) de A est atteignable s'il existe une exécution contenant cette configuration, i.e.,

$$\exists \rho \cdot (\rho \in \Gamma(A) \wedge \rho = (q_0, \mathbf{0}) \overset{t_0}{\rightarrow} (q_1, v_1) \overset{e_0}{\rightarrow} (q_2, v_2) \overset{t_1}{\rightarrow} \cdots (q_i, v_i) \cdots).$$

L'ensemble des configurations atteignables de A sera noté $\texttt{Atteign}(A)$.

Mots et langages temporisés

Un mot temporisé est une séquence finie ou infinie $\omega = (a_0, t_0)(a_1, t_1)(a_2, t_2) \cdots$ où $a_0, a_1, a_2 \cdots$ sont des étiquettes de l'ensemble `Labels` et $t_0, t_1, t_2 \cdots$ des valeurs de $\mathbb{R}^+$. Intuitivement, un mot temporisé est donc une séquence d'actions, chacune étant associée au temps auquel elle est exécutée. Le mot temporisé accepté par l'exécution de l'exemple 2.2 est donc le suivant :

$$(approach, 3.5)\ (enter, 6.5)\ (exit, 6.5) \cdots$$

Le langage reconnu par un automate temporisé est l'ensemble des mots reconnus par les exécutions de l'automate. On notera $\mathcal{L}(A)$ le langage reconnu par un automate temporisé A.
On peut également associer un langage non temporisé à un automate temporisé A. Les mots de ce langage sont donc des séquences d'actions, sans information sur l'instant où est effectuée l'action. Ainsi, le langage $\texttt{Untime}(\mathcal{L}(A))$ contiendra tous les mots pouvant être obtenus à partir des mots de $\mathcal{L}(A)$ en ne gardant de ces mots que les actions. Par exemple, le mot temporisé ci-dessus deviendra "*approach enter exit* $\cdots$". Formellement,

$$\texttt{Untime}(\mathcal{L}(A)) = \{a_1 a_2 \cdots a_n \cdots \mid \exists t_1, t_2, \cdots, t_n, \cdots \in \mathbb{R}^+ \text{ tels que } ((a_1, t_1)(a_2, t_2) \cdots (a_n, t_n) \cdots \in \mathcal{L}(A))\}$$

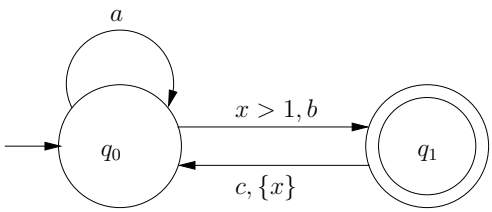


FIG. 2.3 – Automate temporisé avec condition de Büchi

De plus, tout comme pour les automates finis classiques, on peut ajouter aux automates des conditions d'acceptation, restreignant l'ensemble des mots pouvant être acceptés par l'automate. La condition la plus généralement citée est la condition de Büchi, consistant à ajouter à l'automate un ensemble d'états d'acceptation, qui sont certains états de l'automate. Le langage accepté par l'automate sera alors restreint aux mots reconnus par des exécutions passant infiniment souvent par au moins l'un de ces états. La figure 2.3 présente un automate de Büchi temporisé, dont l'ensemble d'états d'acceptation est $\{q_1\}$. L'ensemble des exécutions de cet automate est donc restreint aux exécutions passant infiniment souvent par q_1. Par exemple, l'exécution restant dans q_0 en prenant uniquement des transitions étiquetées par a n'est pas considérée comme une exécution de l'automate. Le mot $(a,t_1)(a,t_2)(a,t_3)\cdots$, avec $t_1,t_2,t_3 \in \mathbb{R}^+$ n'est donc pas un mot du langage reconnu par cet automate.

Restriction du modèle

Suppression des contraintes diagonales. Dans la définition que nous donnons des automates temporisés, nous considérons uniquement des automates utilisant des contraintes non diagonales. Ceci n'est pas réellement une restriction. En effet, le lemme suivant montre que les automates temporisés sans contraintes diagonales ont le même pouvoir d'expression que les automates temporisés utilisant des contraintes d'horloges générales. Une preuve de ce résultat peut être trouvée dans [BDGP98].

LEMME 1 *Soit A un automate temporisé avec des contraintes d'horloges générales. Il existe un automate temporisé A' ne comportant que des contraintes non diagonales qui reconnait le même langage que A, $\mathcal{L}(A) = \mathcal{L}(A')$.*

Notons toutefois qu'une telle construction entraîne une explosion (exponentielle dans le nombre de contraintes diagonales) du nombre d'états pour l'automate obtenu.

Restriction à des constantes entières. Il est également possible d'utiliser des constantes rationnelles dans l'expression des contraintes d'horloges. Le lemme suivant [Bou02] montre que l'on peut toujours se ramener à des constantes entières à partir de ces constantes rationnelles.

LEMME 2 *Soit un automate temporisé A utilisant des constantes rationnelles et $\lambda \in \mathbb{Q}^+$. L'automate λA est obtenu à partir de A en multipliant toutes les constantes de A par λ. Soit $u = (a_1,t_1)\cdots(a_n,t_n)\cdots$ un mot accepté par A, on a alors*

$$u \in \mathcal{L}(A) \Leftrightarrow \lambda u = (a_1, \lambda \cdot t_1) \cdots (a_n, \lambda \cdot t_n) \cdots \in \mathcal{L}(\lambda A)$$

où λu représente le mot u où toutes les dates sont multipliées par λ.

Non blocage et non zénonisme

La propriété de non blocage d'un système est souvent vue comme une propriété essentielle à garantir. Dans le cadre des systèmes temporisés, un blocage (*deadlock*) représente le fait qu'à partir d'une configuration, il n'est pas possible d'effectuer une action discrète. Une configuration (q, v) est donc un blocage s'il n'existe pas de délai t et de transition discrète e tels que e puisse être prise à partir de (q, v) après le délai t.

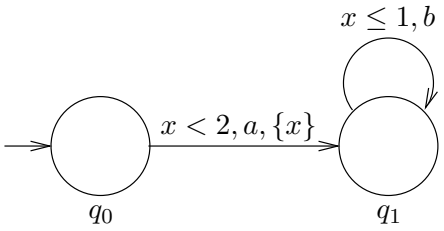


FIG. 2.4 – Un automate temporisé zénon et avec blocages

Considérons par exemple l'automate temporisé de la figure 2.4. Les configurations (q_0, v) pour $v \in x \geq 2$ sont des blocages car la seule action possible a ne peut pas être prise. Les configurations (q_1, v') pour $v' \in x > 1$ sont également des blocages, car l'action b ne peut plus être effectuée.

Un automate temporisé est dit sans blocage si aucune de ses configurations atteignables n'est un blocage. Dans la suite du document, nous utiliserons le prédicat `free` défini par Stavros Tripakis dans [Tri98] pour caractériser l'ensemble des configurations qui ne sont pas des blocages. De manière informelle, étant donné un état de contrôle q d'un automate temporisé A, $\texttt{free}(q)$ calcule l'ensemble des valuations des configurations de A, ayant pour partie discrète q, à partir desquelles le temps peut s'écouler puis effectuer une transition discrète. Ce prédicat est défini par :

$$\texttt{free}(q) = \bigcup_{e \in \texttt{out}(q)} \swarrow (\texttt{guard}(e) \cap ([\texttt{reset}(e) := 0]\texttt{Invar}(\texttt{target}(e))))$$

Un automate temporisé A est donc sans blocage si et seulement si $\forall (q, v) \cdot ((q, v) \in \texttt{Atteign}(A) \Rightarrow v \in \texttt{free}(q))$ [Tri98].

Une autre propriété essentielle, spécifique aux systèmes temporisés, est la propriété de *non zénonisme*. Elle représente le fait que seul un nombre fini d'actions peut être effectué durant un intervalle de temps fini. Cette propriété est énoncée sur les exécutions infinies d'un automate temporisé de la manière suivante : une exécution infinie ρ est dite non zénon si le temps peut diverger au long de l'exécution, en d'autres termes, si le temps peut progresser à l'infini, sans aucune borne supérieure. Formellement, ρ est

non zénon si $\texttt{time}(\rho) = \infty$, i.e., $\forall t \in \mathbb{R}^+$, il existe une configuration (q, v) de ρ telle que $\texttt{time}(\rho, (q, v)) \geq t$. Par extension, un automate temporisé est donc dit non zénon si toutes ses exécutions sont non zénon.

Reprenons par exemple l'automate temporisé de la figure 2.4 et considérons l'exécution

$$(q_0, \mathbf{0}) \xrightarrow{\frac{1}{2}} (q_0, \{\tfrac{1}{2}\}) \xrightarrow{a} (q_1, \mathbf{0}) \xrightarrow{\frac{1}{2}} (q_1, \{\tfrac{1}{2}\}) \xrightarrow{\frac{1}{4}} (q_1, \{\tfrac{3}{4}\}) \xrightarrow{\frac{1}{8}} (q_1, \{\tfrac{7}{8}\}) \cdots$$

de cet automate. Cette exécution est zénon car, à partir de la configuration $(q_1, \mathbf{0})$, le temps converge vers 1. Une infinité d'actions b peut donc être prise dans un intervalle borné de temps.

Dans cette section, nous avons présenté un formalisme – les automates temporisés – permettant de modéliser des systèmes temporisés. Nous avons également vu que le graphe sémantique d'un automate temporisé est infini. Alur et Dill ont présenté dans [AD94] une représentation finie de cet espace d'états, l'automate des régions, tel que cet automate reconnaisse le langage non temporisé `Untime` d'un automate temporisé. Ainsi, cette construction a notamment permis de réduire certains problèmes de décidabilité des automates temporisés aux problèmes de décidabilité pour les automates finis classiques.

2.2.3 L'automate des régions

L'automate des régions est une abstraction du graphe sémantique d'un automate temporisé. Cette abstraction vise à regrouper les configurations dont les valuations sont équivalentes relativement à une relation d'équivalence $\cong_r$ d'indice fini, appelée l'équivalence des régions.

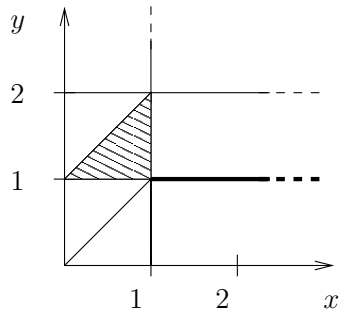
L'équivalence des régions

Considérons un automate temporisé A, muni d'un ensemble d'horloges X. Soient $x \in X$ et $y \in X$ deux horloges de cet automate. On appelle c_x (respectivement c_y) la plus grande constante à laquelle est comparée l'horloge x (respectivement y) dans une contrainte impliquant cette horloge. Deux valuations v et v' sur X sont dites équivalentes relativement à $\cong_r$ si :

1. $\forall x \in X$, soit $\texttt{ent}(v) = \texttt{ent}(v')$, soit $v(x) > c_x$ et $v'(x) > c_x$,
2. $\forall x, y \in X$, si $v(x) \leq c_x$ et $v(y) \leq c_y$ alors

$$\texttt{fract}(v(x)) \leq \texttt{fract}(v(y)) \text{ ssi } \texttt{fract}(v'(x)) \leq \texttt{fract}(v'(y)).$$

Cette relation d'équivalence induit un nombre fini de classes d'équivalence, borné par $|X|! \cdot 2^{|X|} \cdot \prod_{x \in X} (2c_x + 2)$ [AD94]. Ces classes sont appelées *régions d'horloges* ou plus simplement *régions*. Nous notons $[v]$ la région à laquelle appartient la valuation v. Chaque région peut être représentée par une contrainte d'horloges, comme le montre l'exemple suivant.



On compte ici 28 *régions d'horloges* :
- 8 *régions ouvertes, e.g. la région hachurée* $x < 1 \wedge 1 < y < 2 \wedge y < x$,
- 14 *régions de type segment, e.g. la région en gras* $x > 1 \wedge y = 1$,
- 6 *régions de type point, e.g. la région* $x = y = 1$.

FIG. 2.5 – Régions pour deux horloges x et y avec $c_x = 1$ et $c_y = 2$

EXEMPLE 2.3. (RÉGIONS) Considérons un automate temporisé à deux horloges x et y telles que $c_x = 1$ et $c_y = 2$. L'équivalence des régions sur l'ensemble des valuations sur X crée 28 régions d'horloges représentées par la figure 2.5.

On dit qu'une région r' est un successeur d'une région r, noté $r' = \texttt{succ}(r)$, si en laissant s'écouler le temps, on peut atteindre une valuation de la région r' depuis une valuation de la région r. Formellement,

$$r' \in \texttt{succ}(r) \text{ ssi } \exists v, t \cdot (v \in r \wedge t \in \mathbb{R}^+ \wedge v + t \in r').$$

Sur l'exemple de la figure 2.5, la région hachurée a quatre successeurs, définis par les contraintes $x = 1 \wedge 1 < y < 2$, $x > 1 \wedge 1 < y < 2$, $x > 1 \wedge y = 2$ et $x > 1 \wedge y > 2$.

Automate des régions

Cette relation d'équivalence $\cong_r$ entre valuations permet de définir un graphe fini – l'automate des régions – à partir du graphe sémantique d'un automate temporisé A. On notera $\mathcal{R}(A)$ l'automate des régions de A. Chaque état de $\mathcal{R}(A)$ sera donc un ensemble de configurations tel que ces configurations ont la même partie discrète, et leurs valuations forment une région d'horloges. Une configuration (q, v) de A appartiendra donc à l'état $(q, [v])$ de $\mathcal{R}(A)$. La relation de transition est définie de manière à ce que $\mathcal{R}(A)$ puisse effectuer les mêmes séquences d'actions que A. Ainsi, depuis un état (q, r) de $\mathcal{R}(A)$, où r est une région, on peut effectuer une action étiquetée par a jusqu'à un état (q, r') si et seulement si, depuis toutes les configurations (q, v) telles que $v \in r$, cette action peut être également effectuée et mène à une configuration (q, v') telle que $v' \in r'$. Formellement, cet automate est défini de la manière suivante.

DÉFINITION 3 (AUTOMATE DES RÉGIONS) *Soit* $A = \langle \mathrm{Q}, \mathrm{q}_0, \mathrm{Labels}, \mathrm{X}, \mathrm{T}, \texttt{Invar}, \mathrm{L} \rangle$ *un automate temporisé. On note* R *l'ensemble des régions induites par* $\cong_r$ *sur l'ensemble des valuations de* X. *L'automate des régions correspondant à* A *est un t-uplet* $\mathcal{R}(A) = \langle \mathrm{QR}, (\mathrm{q}_0, \texttt{zero}), \mathrm{Labels}, \mathrm{TR} \rangle$ *où :*
- $QR \subseteq Q \times R$ *est l'ensemble fini des états. Chaque état est un couple* (q, r) *tel que* $r \subseteq \texttt{Invar}(q)$,
- $(q_0, \texttt{zero})$ *est l'état initial,*
- $TR \subseteq QR \times \texttt{Labels} \times QR$ *est l'ensemble fini des transitions. La transition* $(q, r) \xrightarrow{a} (q', r')$ *est une transition de* $\mathcal{R}(A)$ *ssi*

$$q \overset{g,a,\gamma}{\rightarrow} q' \in T \text{ et } \exists r'' \in \mathtt{succ}(r) \text{ t.q. } r'' \subseteq g,\ r'' \subseteq \mathtt{Invar}(q) \text{ et } r' = [\gamma := 0]r''.$$

EXEMPLE 2.4. Reprenons l'exemple du passage à niveau. L'automate des régions associé à l'automate temporisé de la barrière est donné par la Fig. 2.6.

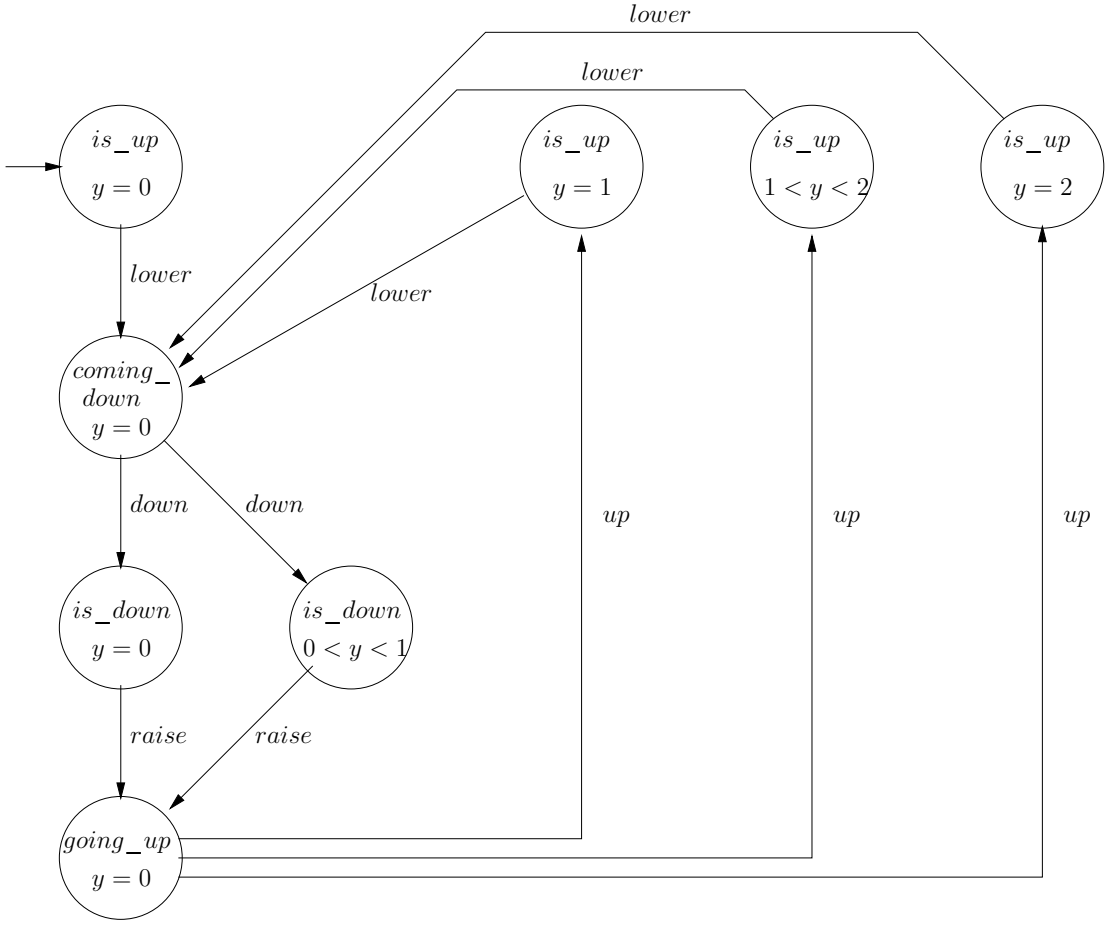


FIG. 2.6 – Automate des régions associé à l'automate temporisé de la barrière

Etant donné un automate temporisé A, l'automate des régions $\mathcal{R}(A)$ construit à partir de A reconnait exactement le langage $\mathtt{Untime}(\mathcal{L}(A))$ [AD94]. Cette construction a permis de montrer la décidabilité du problème du vide pour la classe des automates temporisés. Le problème du vide est le fait de pouvoir tester si le langage reconnu par un automate est vide ou non. C'est un problème essentiel et le résultat suivant montre qu'il est décidable.

THÉORÈME 1 (PROBLÈME DU VIDE [AD94]) *Le problème du vide est décidable pour les automates temporisés. C'est un problème PSPACE-complet.*

L'automate des régions permet donc d'abstraire le graphe sémantique infini d'un automate temporisé A de façon à obtenir un graphe à nombre d'états fini, reconnaissant exactement le langage non temporisé de A. Cependant, cette construction reste inutili-

sable en pratique, du fait d'un nombre d'états encore trop important. Pour cette raison, d'autres abstractions ont été étudiées, comme le graphe de simulation.

2.2.4 Le graphe de simulation

Le graphe de simulation est une abstraction du graphe sémantique d'un automate temporisé couramment utilisée en pratique et basée sur la notion de zones.

Zones

Considérons un automate temporisé A. Une zone est un état symbolique regroupant un ensemble de configurations de A ayant la même partie discrète. Les valuations de ces configurations doivent former un polyhèdre convexe. Une zone z est donc un couple (q, ζ) où q est un état de A et ζ un polyhèdre convexe. On utilisera par la suite $\texttt{disc}(z)$ pour la partie discrète q de la zone z, et $\texttt{poly}(z)$ pour son polyhèdre ζ.

Opérations sur les zones. Les opérations suivantes permettent de calculer les zones successeur et prédécesseur d'un zone donnée z. L'opération $\texttt{time} - \texttt{succ}(z)$ (respectivement $\texttt{time} - \texttt{pred}(z)$) définit l'ensemble des successeurs (resp. prédécesseurs) *temporels* de z, i.e., l'ensemble des configurations pouvant être atteintes depuis une configuration de z (resp. l'ensemble des configurations permettant d'atteindre une configuration de z) en laissant s'écouler le temps. Les opérations $\texttt{disc} - \texttt{succ}(e, z)$ et $\texttt{disc} - \texttt{pred}(e, z)$ définissent respectivement l'ensemble des successeurs et prédécesseurs d'une configuration de z par la transition e.

$$\begin{aligned}
\texttt{time-succ}(z) &\stackrel{def}{=} \{s' \mid \exists s \in z, t \in \mathbb{R}^+ \; s \xrightarrow{t} s'\} \\
\texttt{time-pred}(z) &\stackrel{def}{=} \{s \mid \exists s' \in z, t \in \mathbb{R}^+ \cdot s \xrightarrow{t} s'\} \\
\texttt{disc-succ}(e, z) &\stackrel{def}{=} \{s' \mid \exists s \in z \cdot s \xrightarrow{e} s'\} \\
\texttt{disc-pred}(e, z) &\stackrel{def}{=} \{s \mid \exists s' \in z \cdot s \xrightarrow{e} s'\}
\end{aligned}$$

Définissons à présent les opérations successeur et prédécesseur, notées `post` et `pre`, d'une zone z par une transition e. Le successeur d'une zone z par une transition e (et par rapport à une constante c de $\mathbb{N}$), noté $\texttt{post}(e, z, c)$, est l'ensemble des configurations pouvant être atteintes depuis les configurations de z en prenant la transition e, puis en laissant le temps s'écouler. La zone ainsi obtenue est ensuite modifiée en utilisant l'opérateur d'extrapolation défini précédemment sur les polyhèdres. Nous expliquerons la raison de cette modification lors de la définition du graphe de simulation. Notons également que, pour plus de lisibilité, l'opération `Approx` est appliquée sur une zone, plutôt que sur le polyhèdre de la zone. De manière duale, le prédécesseur d'une zone z par la transition e, noté $\texttt{pre}(e, z)$, est l'ensemble des configurations permettant d'atteindre par e une configuration, qui en laissant le temps s'écouler, mène à une configuration de z.

$$\begin{aligned}
\texttt{post}(e, z, c) &\stackrel{def}{=} \texttt{Approx}_c(\texttt{time} - \texttt{succ}(\texttt{disc} - \texttt{succ}(e, z))) \\
\texttt{pre}(e, z) &\stackrel{def}{=} \texttt{disc} - \texttt{pred}(e, \texttt{time} - \texttt{pred}(z))
\end{aligned}$$

Le graphe de simulation

Le graphe de simulation est une abstraction finie du graphe sémantique d'un automate temporisé, basée sur l'utilisation des zones. En effet, chaque état du graphe est une zone, et les transitions sont calculées à l'aide de l'opération `post`. Seules des transitions discrètes apparaissent dans le graphe, le temps s'écoulant implicitement à l'intérieur des états.

DÉFINITION 4 (GRAPHE DE SIMULATION) *Soit* $A = \langle \mathrm{Q}, \mathrm{q}_0, \mathrm{Labels}, \mathrm{X}, \mathrm{T}, \mathtt{Invar}, \mathrm{L} \rangle$ *un automate temporisé et* $c \in \mathbb{N}$ *une constante supérieure ou égale à* $c_{max}(A)$. *Le graphe de simulation de* A *par rapport à* c, *noté* $SG(A, c)$, *est un quadruplet* $\langle \mathrm{Z}, \mathrm{z}_0, \mathrm{Labels}, \mathcal{E} \rangle$ *où :*

- Z *est l'ensemble d'états du graphe, qui est donc un ensemble de zones,*
- $z_0 = (q_0, \nearrow \mathtt{zero} \cap \mathtt{Invar}(q_0))$ *est la zone initiale,*
- $\mathcal{E} \subseteq Z \times T \times Z$ *est l'ensemble fini des transitions du graphe. Etant donnée une zone* z *et une transition* $e \in T$, *si* $z' = \mathtt{post}(e, z, c) \neq \emptyset$, *alors* z' *est une zone du graphe et* $z \xrightarrow{e} z'$ *est une transition du graphe.*

REMARQUE 2.1. (UTILISATION DE `Approx` POUR LE CALCUL DES SUCCESSEURS) L'opération d'extrapolation `Approx` dans la définition de `post` permet d'assurer la terminaison de la construction du graphe de simulation. Cette abstraction supplémentaire se doit d'être correcte par rapport à l'atteignabilité. En d'autres termes, elle ne doit pas rendre atteignables dans le graphe de simulation des configurations qui ne l'étaient pas dans le graphe sémantique. Or, l'utilisation de contraintes d'horloges générales dans A, en particulier les contraintes diagonales, ne permet pas d'assurer cette propriété [Bou03, Bou04] pour des automates temporisés contenant plus de trois horloges. C'est pour cette raison que nous considérons directement des automates temporisés ne possédant que des contraintes d'horloges non diagonales.

Un chemin du graphe de simulation est une séquence finie ou infinie $\pi = z_0 \xrightarrow{e_0} z_1 \xrightarrow{e_1} z_2 \cdots$. L'ensemble des chemins d'un graphe de simulation SG est noté $\Pi(SG)$.
La notion de non zénonisme définie sur les exécutions d'un automate temporisé est étendue aux chemins du graphe de simulation de la manière suivante. Un chemin est dit non zénon si, pour chaque horloge $x \in X$, soit x est remise à zéro infiniment souvent dans le chemin, soit x reste toujours non bornée à partir d'une zone dans le chemin [Tri98]. Nous appelons $\mathtt{non-zenon}$ le prédicat permettant de tester si un chemin $\pi = z_0 \xrightarrow{e_0} z_1 \xrightarrow{e_1} z_2 \cdots$ est non zénon :

$$\mathtt{non-zenon}(\pi) \stackrel{def}{=} \forall x \cdot (x \in X \Rightarrow$$

$$\forall i \cdot (i \geq 0 \Rightarrow \exists j \cdot (j > i \wedge x \in \mathtt{reset}(e_j))) \vee$$
$$\exists i \cdot (i \geq 0 \wedge \forall j \cdot (j > i \Rightarrow \mathtt{unbounded}(x, \mathtt{poly}(z_j)) \wedge \mathtt{unbounded}(x, \mathtt{guard}(e_j)))))$$

Le lemme 3, énoncé et prouvé dans [Tri98], permet de relier les exécutions d'un automate temporisé A aux chemins du graphe de simulation $SG(A, c)$. Notons tout d'abord qu'on dit qu'une exécution

$$\rho = (q_1, v_1) \xrightarrow{t_1} (q_1, v_1') \xrightarrow{t_1'} (q_1, v_1'') \xrightarrow{e_1} (q_2, v_2) \xrightarrow{t_2} (q_2, v_2') \xrightarrow{e_2} \cdots$$

de A est inscrite dans un chemin $\pi = (q_1, \zeta_1) \xrightarrow{e_1} (q_2, \zeta_2) \xrightarrow{e_2} \cdots$ de $SG(A, c)$ si pour tout $i = 1, 2, \cdots$, $v_i^\dagger \in \zeta_i$, où $v_i^\dagger \in \{v_i, v_i', v_i'', \cdots\}$.

LEMME 3 *Chaque exécution (respectivement non-zénon) de A est inscrite dans un chemin unique (resp. non-zénon) π de $SG(A, c)$, et pour chaque chemin (resp. non-zénon) π de $SG(A, c)$, il existe une exécution (resp. non-zénon) de A inscrite dans π.*

EXEMPLE 2.5. Reprenons l'exemple du passage à niveau modélisé par les automates temporisés de la Fig. 2.2. Les graphes de simulation construits à partir de chaque automate temporisé sont présentés Fig. 2.7. Le polyhèdre de chaque zone est représenté par une contrainte placée à l'intérieur de la zone. De plus, pour chaque zone du graphe, nous avons indiqué entre parenthèses l'état de contrôle correspondant à la partie discrète de la zone.

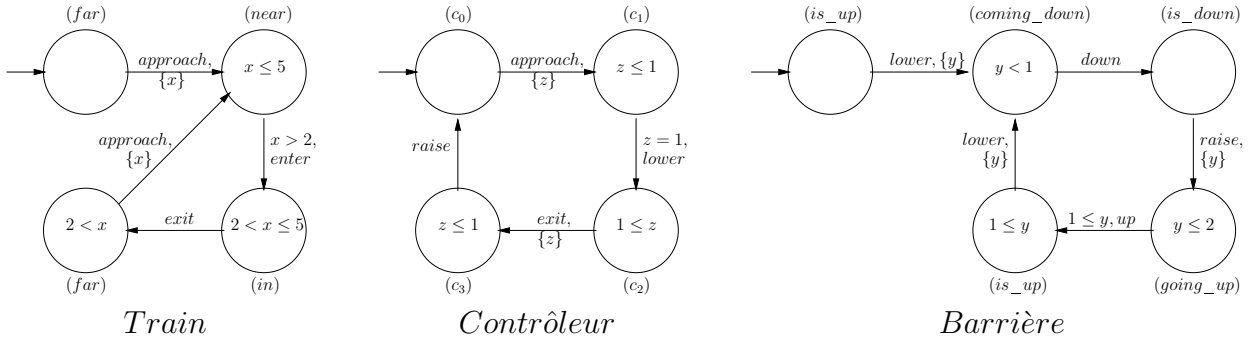


FIG. 2.7 – Graphes de simulation du train, du contrôleur et de la barrière

2.3 Extensions et variantes du modèle original

Dans la section précédente, nous avons présenté le modèle de base des automates temporisés, tel qu'il a été introduit dans [AD94]. Depuis, de nombreux travaux se sont focalisés sur des extensions ou variantes de ce modèle, pour étudier notamment le pouvoir d'expression de ces automates *enrichis* par rapport au modèle initial. L'objet de cette section est de présenter quelques-uns de ces travaux.

2.3.1 Extensions

Parmi les extensions du modèle initial, nous pouvons citer l'ajout d'actions silencieuses dans les automates, l'ajout de nouveaux types de contraintes, ou encore l'utilisation d'opérations supplémentaires sur les horloges. Bien que quelque peu différents, les automates hybrides peuvent également être considérés comme une extension des automates temporisés. Pour cette raison, nous les introduisons également dans cette section.
Chacune de ces extensions est comparée au modèle original en termes de décidabilité (du

problème du vide) et de pouvoir d'expression. Le pouvoir d'expression d'une classe d'automates est donné en termes de langages reconnus par les automates de cette classe. Une classe d'automates $\mathcal{C}_1$ est dite aussi expressive qu'une classe $\mathcal{C}_2$ si pour chaque automate de $\mathcal{C}_1$, il existe un automate de $\mathcal{C}_2$ qui reconnait le même langage. $\mathcal{C}_1$ est dite strictement plus expressive que $\mathcal{C}_2$ s'il existe un automate de $\mathcal{C}_1$ dont le langage n'est reconnu par aucun automate de $\mathcal{C}_2$, et si pour chaque automate de $\mathcal{C}_2$, il existe un automate de $\mathcal{C}_1$ reconnaissant le même langage.

Ajout d'actions silencieuses

Dans le cadre non temporisé des automates finis, l'ajout d'actions silencieuses (ou ε-transitions) n'ajoute pas de pouvoir d'expression à ces automates. Il en est tout autrement pour les automates temporisés. En effet, il a été montré dans [BDGP98] que la classe des automates temporisés avec transitions silencieuses a un pouvoir d'expression strictement supérieur à la classe initiale d'automates temporisés[8] (mais reste décidable). Par exemple, le langage de l'automate (de Büchi) temporisé de la Fig. 2.8 ne peut être accepté par aucun automate (de Büchi) temporisé classique. Cet exemple est tiré de [BDGP98].

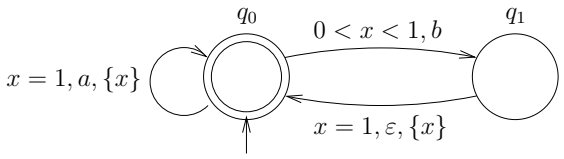


FIG. 2.8 – Automate temporisé avec actions silencieuses

Ajout d'autres types de contraintes

Une autre extension également étudiée dans la littérature concerne les contraintes d'horloges. Les contraintes utilisées dans le modèle classique des automates temporisés n'autorisent que des comparaisons entre horloges et constantes, ou des comparaisons entre différences d'horloges et constantes.
Dans [AD94, BD00], l'ajout de contraintes incluant des additions entre horloges comparées à une constante, du type $x+y \sim c$ a été étudiée. En termes de décidabilité, ce modèle reste décidable pour des modèles contenant deux horloges, avec un pouvoir d'expression strictement supérieur à celui du modèle initial, mais devient indécidable avec quatre horloges ou plus. Le problème reste ouvert pour des automates temporisés à trois horloges utilisant ce type de contraintes.
L'utilisation de contraintes périodiques a également été étudiée dans [CG00]. Par exemple, une contrainte de type $x = 2n+1$, où x est une horloge et n un entier, est une contrainte périodique. Le pouvoir d'expression des modèles utilisant de telles contraintes est strictement supérieur à celui du modèle classique si les actions silencieuses sont autorisées. Sinon, le pouvoir d'expression est le même. De plus, ce modèle reste décidable.

[8]Rappelons que nous appelons classe initiale, la classe des automates temporisés classiques comme définis dans la section précédente.

Ajout d'autres opérations sur les horloges

Le modèle initial des automates temporisés n'autorise que des remises à zéro d'horloges sur les transitions. Les automates temporisés avec mises à jour [BDFP04, Bou02] sont une extension de ce modèle permettant d'effectuer d'autres opérations sur les horloges, appelées des mises à jour. Les mises à jour simples ont la forme suivante :

$$x :\sim c \mid x :\sim y + c \text{ où } \sim \in \{<, \leq, =, \geq, >\},\ c \in \mathbb{N}, \text{ et } x \text{ et } y \text{ sont des horloges.}$$

Il est possible, sur les transitions, d'utiliser des conjonctions de telles mises à jour simples. Ces mises à jour permettent, entre autres, d'exprimer des opérations telles que incrémenter une horloge ($x := x + 1$), affecter à une horloge la valeur d'une autre ($x := y$), ou encore donner à une horloge une valeur supérieure à une constante, de manière non déterministe ($x :> c$).
Le modèle général est indécidable, mais [BDFP04, Bou02] donnent une caractérisation de quatre sous-classes décidables, ainsi que de quatre autres sous-classes indécidables, dépendant du type de contraintes (générales ou non-diagonales) et du type de mises à jour utilisées. Les sous-classes décidables ont en général le même pouvoir d'expression que la classe initiale des automates temporisés ou que la classe des automates temporisés avec transitions silencieuses. En effet, deux de ces sous-classes sont aussi expressives que le modèle initial, et une autre l'est autant que la classe incluant les transitions silencieuses. La dernière sous-classe distinguée, utilisant des contraintes générales et impliquant des mises à jour $x :< c$, est, quant à elle, strictement plus expressive que ces deux classes.

Les automates hybrides

Les automates temporisés sont une sous-classe des automates hybrides. Les automates hybrides [ACHH93, Hen96] peuvent donc être vus comme des automates temporisés, qui utilisent en plus d'autres variables continues dont l'évolution est cette fois définie par des lois dynamiques données sous la forme d'équations différentielles. Le modèle général des automates hybrides est indécidable. Cependant, quelques sous-classes décidables ont été exhibées [HK94, HKPV98], citons notamment les automates rectangulaires initialisés.
La Fig. 2.9 représente un exemple d'automate hybride (modèle général) modélisant le fonctionnement d'un thermostat, dont le rôle est de garder la température d'une pièce entre m et M degrés. La température est modélisée par la variable x. Lorsque la température est inférieure à $m°$, le thermostat allume un radiateur, puis l'éteint dès que cette température atteint $M°$. Dans l'état q_0, le radiateur est éteint et la température diminue selon la fonction $x(t) = \theta e^{-Kt}$, où t représente le temps qui s'écoule (de manière continue), θ est la température initiale et K est une constante dépendant de la pièce. Dans l'état q_1, le radiateur est allumé, et la température augmente selon la fonction $x(t) = \theta e^{-Kt} + h(1 - e^{-Kt})$, où h est une constante dépendant de la puissance du radiateur.

2.3.2 Une variante : les automates temporisés avec deadlines

Nous nous concentrons ici sur une variante d'automates temporisés que nous allons ensuite étudier dans le cadre de notre travail. Cette variante a été présentée dans

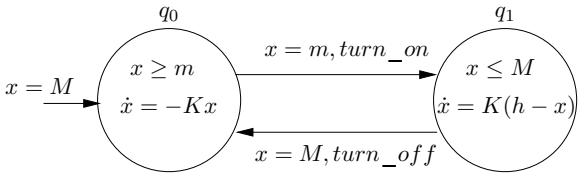


FIG. 2.9 – Automate hybride modélisant le thermostat

[SY96, Bor98, BST97, BS00] où elle est appelée notamment *automates temporisés avec deadlines* ou encore *modèle à urgence*.
Dans les automates temporisés classiques, les conditions de progression du temps sont données par les invariants que l'on trouve associés aux états de contrôle de l'automate. Ainsi, le temps peut progresser de la même manière pour toutes les transitions partant d'un même état, dans la limite imposée par l'invariant. Dans le modèle à urgence, ces conditions ne sont plus associées aux états, mais directement aux transitions de l'automate. Chaque transition possède alors un degré d'urgence, modélisé par une échéance (*deadline*). L'échéance d'une transition représente le moment où le temps ne peut plus progresser, autrement dit où il est bloqué, pour cette transition. On dit alors que la transition devient urgente et elle doit donc être prise immédiatement.

Plus particulièrement, trois degrés d'urgence d'actions (et donc, trois types d'échéances) ont été caractérisés, qui, d'après les auteurs, modélisent la plupart des situations pouvant être rencontrées en pratique : les actions *paresseuses*, les actions *immédiates* et les actions *retardables*. Détaillons ces trois types d'actions :

- Les actions *paresseuses*, notées λ, sont les transitions pour lesquelles le degré d'urgence est le plus bas. Le temps peut toujours progresser.
- Les actions *immédiates*, notées ϵ, sont les transitions qui ne permettent pas de laisser passer le temps avant d'être prises. Autrement dit, elles doivent être prises dès lors qu'elles sont activables.
- Les actions *retardables*, notées δ, sont les transitions les plus courantes. Dans ce cas, le temps peut s'écouler tant que la transition reste activable dans le futur. Son déclenchement peut ainsi être retardé jusqu'au dernier instant où sa garde est vraie, mais est forcé lorsque cette limite est atteinte.

Formellement, l'échéance d'une transition est modélisée par une contrainte d'horloges, représentant l'intervalle de temps pendant lequel la transition est urgente. L'échéance d'une transition paresseuse est `false`. Celle d'une transition immédiate est égale à sa garde, signifiant que la transition est urgente, et donc doit être prise, dès qu'elle est activable. Enfin, l'échéance d'une transition retardable est le *front descendant* de sa garde, ce qui signifie que la transition devient urgente au dernier moment où sa garde permet de l'activer. Le schéma de la Fig. 2.10, tiré de [BS00], résume ces différents types d'échéances.

Un automate temporisé avec deadlines est donc défini comme un automate temporisé classique où les invariants sont supprimés, et où les transitions sont équipées d'une

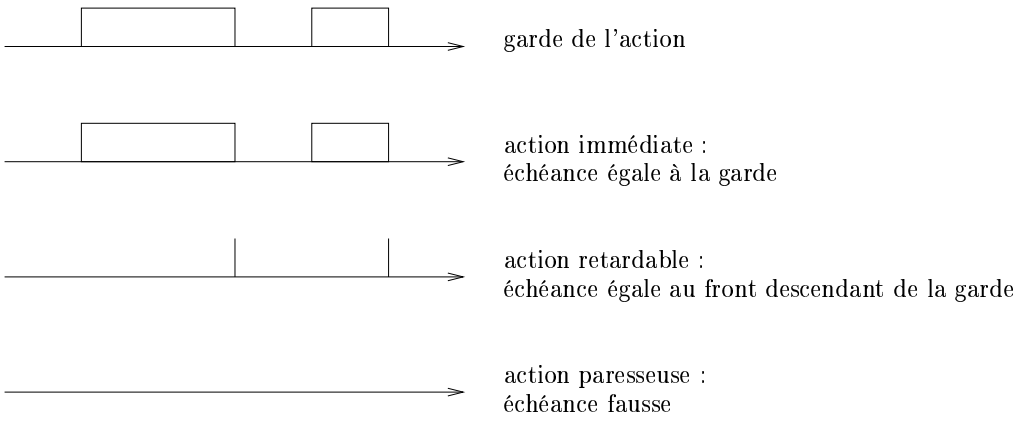


FIG. 2.10 – Les différents types d'échéances

échéance en plus de leur garde, étiquette et remises à zéro. Dans la définition 5, nous définissons les automates temporisés avec deadlines en n'utilisant que les trois types d'échéances définis précédemment. Bien entendu, il est possible de définir un modèle plus général en remplaçant ces types directement par des contraintes d'horloges.

DÉFINITION 5 (AUTOMATES TEMPORISÉS AVEC DEADLINES) *Soit Props un ensemble de propositions atomiques. Un automate temporisé avec deadlines est un tuple* $A = \langle \mathrm{Q}, \mathrm{q_0}, \mathrm{Labels}, \mathrm{X}, \mathrm{T}, \mathrm{L} \rangle$ *où :*

- Q *est l'ensemble des états de contrôle de l'automate,*
- q_0 *est son état initial,*
- Labels *est un alphabet fini de noms d'actions,*
- X *est un ensemble fini d'horloges,*
- $T \subset Q \times \mathcal{C}_{df}(X) \times \{\lambda, \epsilon, \delta\} \times \mathtt{Labels} \times 2^X \times Q$ *est l'ensemble fini des transitions de l'automate,*
- L *est la fonction de décor des états, associant à chaque état un ensemble de propositions atomiques de Props.*

Chaque transition est représentée par un t-uplet $e = (q, g, d, a, \gamma, q')$ où q, g, a, γ, q' sont définis comme le modèle classique et $d \in \{\lambda, \epsilon, \delta\}$ est le type d'échéance de la transition. La notation $\mathtt{deadline}(e)$ représentera la contrainte déduite du type d'échéance associé à la transition e (`false` pour une transition de type λ, g pour une transition de type ϵ, et $g\downarrow$ pour une transition de type δ, où l'opérateur $\downarrow$ représente le front descendant de g).
A partir de ces échéances, il est possible de définir une condition de progression du temps *classique* associée à chaque état q de l'automate : $\mathtt{Invar}(q) = \neg \bigvee_{e \in \mathtt{out}(q)} \mathtt{deadline}(e)$. La sémantique des automates temporisés avec deadlines utilise d'ailleurs cette transformation. Comme pour le modèle classique des automates temporisés, cette sémantique est donnée par un graphe. Ce graphe est défini de la même manière que dans le cas classique, sauf pour les transitions de temps, définies de la manière suivante. La transition de temps $(q, v) \xrightarrow{t} (q, v + t)$ existe dans le graphe sémantique si $\forall t' \cdot (t' < t \Rightarrow v + t' \in \mathtt{Invar}(q))$.

La différence de sémantique se situe dans le fait que le temps peut progresser de t s'il peut progresser jusqu'à t exclus ($t' < t$), alors que dans le modèle classique, le temps peut progresser de t s'il peut progresser jusqu'à t inclus ($t' \leq t$).

Des restrictions sur le modèle des automates temporisés avec deadlines permettent d'assurer l'absence de blocages dans ces automates : les conditions de progression du temps déduites des deadlines doivent être ouvertes à droite, et les deadlines doivent être contenues dans les gardes. Pour plus de détails quant à ces restrictions, se référer à [BS00].

[Bor98] introduit également la notion de priorités entre actions sur ce modèle à urgence. Ceci permet de réduire le non-déterminisme entre les actions en définissant par exemple, entre deux transitions issues du même état, laquelle doit être effectuée prioritairement. Il est alors possible d'interdire qu'une action a s'effectue, si une autre action b peut l'être. Il est également possible de jouer sur les contraintes de temps en disant par exemple que l'action a ne peut être effectuée que si l'action b ne peut pas l'être dans les n unités de temps qui suivent.

L'ordre de priorité défini entre les actions d'un automate est une relation $<$: $\mathit{Labels} \times \mathbb{N} \cup \{\infty\} \times \mathit{Labels}$. La notation $a <_k b$ signifie que a ne sera pas effectuée si b peut l'être dans moins de k unités de temps, tandis que $a <_\infty b$ signifie que a ne sera jamais effectuée si b peut l'être à un moment dans le futur. Ces priorités sont intégrées à un automate en modifiant les gardes des actions non prioritaires :

$$g'_i = g_i \wedge \bigwedge_{j \neq i, a_i <_k a_j} \neg \nearrow_k g_j$$

où g'_i est la garde modifiée. Ces priorités ne sont appliquées qu'entre les actions a_i et a_j issues d'un même état de contrôle. La modification des gardes utilise les opérateurs $\nearrow$ et $\swarrow$. Intuitivement, $\nearrow_k g$ rassemble toutes les valuations pour lesquelles g sera vraie dans moins de k unités de temps. De façon duale, l'opérateur $\swarrow_k g$ rassemble toutes les valuations pour lesquelles g a été vraie il y a moins de k unités de temps. Par exemple :

- $\nearrow_\infty (3 \leq x \leq 4) = 0 \leq x \leq 4$ et $\nearrow_1 (3 \leq x \leq 4) = 2 \leq x \leq 4$
- $\swarrow_\infty (3 \leq x \leq 4) = 3 \leq x$ et $\swarrow_1 (3 \leq x \leq 4) = 2 \leq x \leq 5$

Par la suite, on écrira simplement $\nearrow$ à la place de $\nearrow_\infty$ et $\swarrow$ à la place de $\swarrow_\infty$. Formellement, ces deux opérateurs sont définis de la manière suivante (v et v' sont des valuations) :

$$\nearrow_k \ g = \{v \mid \exists t \cdot (t \in \mathbb{R}^+ \wedge 0 \leq t \leq k \wedge g(v+t))\}$$

$$\swarrow_k \ g = \{v \mid \exists t \cdot (t \in \mathbb{R}^+ \wedge 0 \leq t \leq k \wedge \exists v' \cdot (v = v' + t \wedge g(v')))\}$$

2.4 D'autres modèles pour les systèmes temporisés

Les modèles pour les systèmes temporisés sont en général des extensions de formalismes établis pour modéliser les systèmes non temporisés. Pour cette raison, il en existe donc un très grand nombre. Nous nous concentrons dans cette section à présenter deux extensions temporisées : celles des réseaux de Petri et celles des algèbres de processus.

Extension temporisée des réseaux de Petri

Les réseaux de Petri [Esp97, JM95] sont particulièrement adaptés à la modélisation de systèmes concurrents. Un réseau de Petri comporte des places, des transitions, des arcs (orientés) et des jetons qui se *déplacent* d'une place à une autre par l'intermédiaire des transitions. Les transitions sont reliées aux places par les arcs. Un état d'un réseau de Petri est alors constitué d'une place munie d'un certain nombre de jetons. Dans la définition de base des réseaux de Petri, une transition ne peut être franchie que lorsque toutes les places en entrée de la transition contiennent au moins un jeton. Un jeton sera alors placé dans chaque place cible de la transition.

De nombreuses extensions temporisées des réseaux de Petri ont été définies dans la littérature, par exemple [MF76, Sif77, BD91, GMMP91, AN01]. Une vue d'ensemble de ces extensions peut être trouvée dans [Bow96]. Certaines associent les contraintes de temps (modélisées par des intervalles) aux places [Sif77], d'autres aux transitions [MF76]. L'approche récente de [AN01] considère des réseaux de Petri temporisés, où les contraintes sont associées aux arcs. Ils sont définis de la manière suivante. A chaque jeton est associée une horloge, représentant "l'âge" du jeton. Un intervalle à bornes entières est ajouté sur chaque arc du réseau de Petri temporisé. Ainsi une transition peut être prise si l'âge des jetons des places en entrée de la transition appartient à l'intervalle placé sur l'arc qui mène de la place à la transition. Les jetons générés par la transition, et donc placés dans les places cible de la transition, auront un âge compris dans l'intervalle placé sur les arcs menant de la transition aux places cible.

Certains travaux ont étudié la transformation de certains types de réseaux de Petri temporisés en automates temporisés, dans l'objectif de pouvoir utiliser les outils de vérification existants pour les automates temporisés [SY96, LR03]. D'autres travaux montrent comment calculer et représenter l'espace d'états d'un réseau de Petri temporisé (où les contraintes de temps sont associées aux transitions) en utilisant des zones [GRR03, GRR06].

Extension temporisée des algèbres de processus

Les algèbres de processus, comme CSP (Communicating Sequential Processes) [Hoa85], CCS (Calculus of Communicating Systems) [Mil89] ou encore ACP (Algebra of Communicating Processes) [BK84], sont particulièrement adaptées à la modélisation des systèmes parallèles, concurrents ou encore à la description de protocoles. Elles considèrent des *processus* ou *agents* qui communiquent suivant des règles de composition dépendant du type de synchronisation utilisé. Les algèbres de processus citées précédemment diffèrent notamment par la notion de communication qu'elles considèrent. Nous détaillons ici l'extension temporisée de CSP, appelée TCSP (*Timed CSP*). Les extensions temporisées pour CCS et ACP ont été étudiées respectivement dans [Che92, MT90, Yi90] et dans [BB94, BM01].

TCSP a été défini et étudié notamment dans [RR88, DS95, Sch99]. Par rapport à la définition non temporisée de CSP, de nouveaux opérateurs sur les processus ont été ajou-

tés (parfois étendus d'opérateurs existants dans CSP pour prendre en compte la notion de temps). Par exemple, *Wait* t permet de modéliser le fait que le processus doit attendre pendant un délai de t unités de temps. L'opérateur $\triangleright$ a été étendu avec un paramètre de temps t, noté $\overset{t}{\triangleright}$, pour modéliser les *timeouts* : le processus défini par $P \overset{t}{\triangleright} Q$ se comporte comme Q à l'instant t si aucune synchronisation n'a été effectuée avec P avant ce temps t (le choix est non déterministe lorsqu'une synchronisation avec P arrive exactement au temps t). L'opérateur de transfert $\overset{t}{\lightning}$ a également été étendu : le processus défini par $P \overset{t}{\lightning} Q$ se comporte comme P jusqu'au temps t, puis comme le processus Q.
TCSP considère une nature du temps continue. Les différents modèles sémantiques pour TCSP ont été généralement étendus des modèles sémantiques pour CSP, en prenant en compte les aspects temporisés. Le modèle le plus simple, que ce soit dans CSP ou TCSP, est le modèle de traces. Pour TCSP, chaque processus est associé à un ensemble de traces finies, chacune de ces traces étant une séquence d'événements observables temporisés (chaque événement est associé à la date à laquelle il est déclenché). D'autres modèles, comme le *failure model* (prenant en compte les blocages pouvant survenir dans le processus), le *stability model* (prenant en compte la divergence des traces) ou le *failures-stability model* ont également été étendus.

2.5 Conclusion

Nous avons présenté dans ce chapitre différents formalismes pouvant être utilisés pour modéliser des systèmes temporisés. Ce sont en général des extensions temporisées de formalismes définis pour les systèmes ne considérant pas les contraintes de temps. Les automates temporisés peuvent être vus comme une extension des automates finis classiques, utilisant des horloges pour modéliser la notion de temps. De la même manière, les réseaux de Petri temporisés, ou les algèbres de processus temporisées sont également des extensions temporisées des formalismes correspondants.
Nous nous sommes en particulier attachés à présenter l'un de ces formalismes, couramment étudié et utilisé comme formalisme de modélisation des systèmes temporisés. Nous en avons donc présenté la définition originale, ainsi que certaines extensions. Nous en avons également présenté en détail une variante, les automates temporisés avec deadlines. Ce sont ces deux modèles, les automates temporisés classiques et les automates temporisés avec deadlines, que nous allons considérer au long de ce document.

Ce chapitre était donc dédié à la présentation des formalismes de modélisation des systèmes temporisés. Dans le chapitre suivant, nous présentons les principes de la vérification par model-checking dans ce cadre temporisé.

3

Vérification des systèmes temporisés

Sommaire

Dans ce chapitre, nous nous concentrons sur l'étape de vérification des systèmes temporisés. Rappelons que, étant donné le modèle d'un système et une propriété de ce système,

la vérification consiste à s'assurer que le modèle que l'on a fait du système satisfait la propriété. Nous nous focalisons en particulier sur la vérification par model-checking.
De nombreux formalismes ont été définis pour spécifier les propriétés des modèles temporisés. Nous en retenons deux particulièrement. Tout d'abord, une logique linéaire, MITL (Metric Interval Temporal Logic) [AFH96], sur laquelle nous basons notre travail, et une logique arborescente, TCTL (Timed Computation Tree Logic)[ACD93], sur laquelle se focalisent les outils actuels de vérification de systèmes temporisés. Pour ces deux logiques, nous présentons également les méthodes de vérification par model-checking, ainsi que des outils implantant ces méthodes.

3.1 Spécification de propriétés pour les modèles temporisés

Dans le chapitre précédent, nous avons introduit les modèles pouvant être utilisés pour représenter formellement des systèmes temporisés. Dans cette section, nous nous attachons à présenter les formalismes permettant de décrire les propriétés que doivent respecter ces modèles.

3.1.1 Les différents types de propriétés

Avant d'introduire ces différents formalismes, commençons par répertorier les différents types de propriétés pouvant être modélisées. Nous reprenons ici la classification de [SBB+99] et nous attachons donc à cinq familles de propriétés : les propriétés de sûreté, de vivacité, d'atteignabilité et d'équité, ainsi que des propriétés de vivacité bornée (également appelée réponse bornée), spécifiques aux systèmes temporisés.

Les propriétés d'atteignabilité

Comme leur nom l'indique, les propriétés d'atteignabilité expriment le fait que certains états du système peuvent être atteints. Si l'on reprend l'exemple du passage à niveau, une propriété d'atteignabilité portant sur le train pourrait être *le train doit toujours pouvoir revenir à une position "loin" du passage à niveau* (P_1).

Les propriétés de sûreté

Les propriétés de sûreté expriment le fait que, sous certaines conditions, quelque chose ne doit jamais arriver. Sur l'exemple du passage à niveau, on a la propriété de sûreté suivante : *La barrière n'est jamais ouverte quand un train est sur le passage à niveau* (P_2).
Notons que les propriétés de sûreté sont assimilées à des propriétés de non-atteignabilité, qui consistent donc à s'assurer que certains états du système ne sont pas atteignables. Sur l'exemple précédent, il s'agit de s'assurer qu'on ne pourra jamais atteindre un état où le train est sur le passage à niveau et la barrière est ouverte. Notons également que, bien qu'elles soient distinguées par la classification de [SBB+99], les propriétés d'absence

de blocage, exprimant le fait que le système ne se trouve jamais dans un état où il ne peut progresser, peuvent être considérées comme des propriétés de sûreté. Tout comme [SBB+99], dans le reste du document, nous ne les considérerons pas comme faisant partie des propriétés de sûreté.

Les propriétés de vivacité

Les propriétés de vivacité expriment le fait que quelque chose finira obligatoirement par avoir lieu, là aussi sous certaines conditions. Par exemple : *si le contrôleur reçoit un signal d'approche du train, alors la barrière devra finir par être fermée* (P_3).

Les propriétés d'équité

Ces propriétés expriment le fait que, sous certaines conditions, quelque chose aura lieu un nombre de fois infini. Par exemple, *la barrière sera levée infiniment souvent* (P_4) est une propriété d'équité.

Les propriétés de vivacité bornée (ou réponse bornée)

Ce type de propriétés est spécifique aux systèmes temporisés, car elles font intervenir la notion de temps. En effet, elles expriment le fait que, sous certaines conditions, quelque chose finira par avoir lieu dans un délai imposé. Sur l'exemple du passage à niveau, la propriété de vivacité précédente peut être affinée en une propriété de vivacité bornée : *si le contrôleur reçoit un signal d'approche du train, alors la barrière sera fermée au maximum dans les deux unités de temps qui suivent* (P_5).

Les logiques temporelles

La formalisation de ces propriétés peut être effectuée de plusieurs manières. Dans le cas non temporisé, une méthode largement répandue est d'utiliser des logiques *temporelles*, permettant d'énoncer des propriétés concernant le comportement dynamique du système, c'est-à-dire l'enchaînement des états lors de l'exécution du système. Ces logiques utilisent donc des opérateurs temporels spécifiques pour spécifier les propriétés dynamiques, en plus des opérateurs booléens classiques. On distingue en général deux types de logiques temporelles [SBB+99] :

- Les logiques du temps linéaire (ou plus simplement les logiques linéaires) : elles permettent d'énoncer ce que l'on appelle des formules de chemins. Ces formules portent sur l'ensemble des exécutions du système, sans s'intéresser à la manière dont elles sont organisées en arbre. En d'autres termes, elles examinent les exécutions une à une et considèrent que chaque état d'une exécution a un seul futur possible : celui donné par l'exécution examinée. Elles ne prennent pas en compte les autres exécutions pouvant partir de cet état. En particulier, les propriétés d'atteignabilité, exprimant le fait qu'il est toujours possible (c-à-d, il existe un chemin) à partir d'un état du système d'atteindre un état donné, ne peuvent pas être exprimées à l'aide d'une logique linéaire. Il est cependant possible d'exprimer des propriétés de sûreté,

de vivacité et d'équité. Un exemple de logique linéaire est la LTL (Linear Temporal Logic) [Pnu81].
- Les logiques du temps arborescent (ou plus simplement les logiques arborescentes) : contrairement aux logiques linéaires, elles permettent d'exprimer des propriétés portant sur l'organisation en arbre des exécutions. De ce fait, les propriétés d'atteignabilité peuvent être exprimées par ce type de logique, ainsi que les propriétés de sûreté et les propriétés de vivacité. En règle générale, il n'est pas possible d'exprimer des propriétés d'équité avec ce type de logique. A titre d'exemple, une logique arborescente particulièrement utilisée est CTL (Computation Tree Logic) [CE81, EH82, QS82].

Dans le cas temporisé, des logiques temporisées ont également été définies afin de prendre également en compte la notion de temps. Elles permettent non seulement d'énoncer des propriétés temporelles, mais également de spécifier des délais à respecter dans l'enchaînement des états du système.
Dans ce document, nous étudions particulièrement deux logiques temporisées : une logique temporisée linéaire, MITL (Metric Interval Temporal Logic) et une logique temporisée arborescente, TCTL (Timed Computation Tree Logic).

3.1.2 Une logique linéaire : Metric Interval Temporal Logic (MITL)

MITL est une logique temporisée linéaire, définie dans [AFH96]. Elle peut être considérée comme une extension de la logique linéaire (non temporisée) LTL.

Syntaxe

Les formules MITL sont définies inductivement selon la grammaire suivante :

$$\varphi ::= \texttt{true} \mid ap \mid \neg\varphi \mid \ \varphi \vee \varphi \mid \varphi \, \mathcal{U}_I \, \varphi$$

où ap est une proposition atomique, et I est un intervalle non singulier de $\mathbb{R}^+$ avec des bornes entières ou infinies. Un intervalle est dit singulier s'il est de la forme $[a, a]$, autrement dit, c'est un intervalle fermé et dont les bornes gauche et droite sont égales. L'opérateur $\mathcal{U}$, pour Until, est un opérateur temporel. Intuitivement, $\varphi_1 \, \mathcal{U}_I \, \varphi_2$ est vraie sur une exécution si φ_1 est vraie jusqu'à ce que φ_2 le soit, φ_2 devant être vraie à un instant t dans l'intervalle de temps I suivant le moment où φ_1 a été vraie et φ_1 devant être vraie jusqu'à cet instant t (non compris).

D'autres opérateurs temporels peuvent également être définis à partir de cette syntaxe :
- $\Diamond_I \varphi \equiv \texttt{true} \, \mathcal{U}_I \, \varphi$, signifiant que φ sera fatalement vraie durant l'intervalle I,
- $\Box_I \varphi \equiv \neg\Diamond_I \neg\varphi$, signifiant que φ est toujours vraie durant l'intervalle I.

Notons que, pour plus de lisibilité, les intervalles contraignant les opérateurs temporels peuvent également être écrits sous la forme d'expressions telles que ≥ 0, qui représente l'intervalle $[0..\infty[$.

EXEMPLE 3.1. Les propriétés dynamiques portant sur l'exemple du passage à niveau et présentées dans la section précédente peuvent être exprimées formellement en MITL.

Par exemple, la propriété de sûreté P_2 *"La barrière n'est jamais ouverte quand un train est sur le passage à niveau"* est exprimée en MITL par : $\Box_{\geq 0}(in \Rightarrow \neg is\_up)$. Notons que le temps n'apparaît pas de manière quantitative dans cette formule, et que la propriété aurait pu être exprimée à l'aide d'une logique non temporisée comme LTL. En revanche, la propriété de vivacité bornée P_5 *"si le contrôleur reçoit un signal d'approche du train, alors la barrière sera fermée au maximum dans les deux unités de temps qui suivent"* fait intervenir un aspect quantitatif du temps, et ne peut donc pas être exprimée en LTL. Elle est exprimée en MITL par $\Box_{\geq 0}(is\_up \wedge c_1 \Rightarrow \Diamond_{[0,2[} is\_down)$.

Sémantique

Dans [AFH96], où est définie la MITL, la sémantique des automates temporisés n'est pas donnée en termes de graphe et d'exécutions comme nous l'avons présentée dans la section 2.2, mais en termes de *séquences d'états temporisés*. Dans la définition originale, les formules MITL sont donc interprétées sur ces séquences d'états temporisées, plutôt que sur des exécutions.

Une séquence d'états temporisée (SET) est une séquence $\sigma = (q_0, I_0) \xrightarrow{e_0} (q_1, I_1) \xrightarrow{e_1} \cdots$. Les q_i et e_i sont respectivement des états de contrôle et des transitions de l'automate temporisé. Les I_i sont des intervalles fermés tels que :

- I_0 est fermé à gauche, et sa borne gauche est égale à 0,
- pour tout $i \geq 0$, les intervalles I_i et I_{i+1} sont adjacents, i.e., la borne droite de I_i est égale à la borne gauche de I_{i+1},
- chaque instant $t \in \mathbb{R}^+$ appartient à un intervalle I_i de la SET.

Intuitivement, chaque intervalle de la SET représente le temps pendant lequel on peut laisser le temps s'écouler avant de prendre une transition discrète. Par rapport aux exécutions, on peut donc dire que les transitions de temps qui apparaissent dans ces exécutions sont en quelque sorte *"contenues"* dans les intervalles.

La relation entre les exécutions que nous avons définies et les SET est la suivante. On dit qu'une exécution

$$\rho = (q_1, v_1) \xrightarrow{t_1} (q_1, v_1') \xrightarrow{e_1} (q_2, v_2) \xrightarrow{t_2} (q_2, v_2') \xrightarrow{t_2'} (q_2, v_2'') \xrightarrow{e_2} \cdots$$

est inscrite dans la SET

$$\sigma = (q_1, I_1) \xrightarrow{e_1} (q_2, I_2) \xrightarrow{e_2} \cdots$$

si pour tout $i \geq 1$, $\texttt{time}(\rho, (q_i, v_i^\dagger)) \in I_i$, $\forall v_i^\dagger \in \{v_i, v_i', v_i'', \cdots\}$.

Notons qu'une exécution est inscrite dans une seule SET, et qu'une SET peut contenir une infinité d'exécutions, du fait que nous ne concaténons pas les transitions de temps successives dans une exécution. Par exemple, l'exécution du train

$$(far, 0) \xrightarrow{3.5} (far, 3.5) \xrightarrow{approach} (near, 0) \xrightarrow{2} (near, 2) \xrightarrow{1} (near, 3) \xrightarrow{enter} (in, 3) \xrightarrow{0} (in, 3) \xrightarrow{exit} (far, 3) \cdots$$

est inscrite dans la SET

$$(far, [0, 3.5]) \overset{approach}{\rightarrow} (near, [3.5, 6.5]) \overset{enter}{\rightarrow} (in, [6.5, 6.5]) \overset{exit}{\rightarrow} \cdots$$

L'exécution dans laquelle les transitions $(near, 0) \xrightarrow{2} (near, 2)$ et $(near, 2) \xrightarrow{1} (near, 3)$ sont concaténées en une transition $(near, 0) \xrightarrow{3} (near, 3)$ existe également dans le graphe sémantique du train. Elle est inscrite dans la même SET. Nous notons $\sigma(\rho)$ la SET dans laquelle est inscrite l'exécution ρ, et (σ, i) le i^{e} couple de la SET σ. Etant donné un temps $t \in I_i$, nous notons σ^t le suffixe d'une SET σ au temps t, où $\sigma^t = (q_i, I_i - t) \xrightarrow{e_i} (q_{i+1}, I_{i+1} - t) \xrightarrow{e_{i+1}} (q_{i+2}, I_{i+2} - t) \cdots$.

Les formules MITL sont donc interprétées sur des séquences d'états temporisées σ, de la manière suivante :

- $\sigma \models \texttt{true}$ est vrai,
- $\sigma \models ap$ ssi $ap \in L(\texttt{disc}((\sigma, 0)))$,
- $\sigma \models \neg\varphi$ ssi il n'est pas vrai que $\sigma \models \varphi$,
- $\sigma \models \phi \vee \psi$ ssi $\sigma \models \phi$ ou $\sigma \models \psi$,
- $\sigma \models \phi \, \mathcal{U}_I \, \psi$ ssi il existe $t \in I$ t.q. $\sigma^t \models \psi$, et $\forall t' \in]0, t[$, $\sigma^{t'} \models \phi$.

Par extension, on dit qu'une formule MITL est satisfaite sur un automate temporisé A, noté $A \models \varphi$, si $\forall \rho \cdot (\rho \in \Gamma(A) \Rightarrow \sigma(\rho) \models \varphi)$.

3.1.3 Une logique arborescente : Timed Computation Tree Logic (TCTL)

TCTL a été tout d'abord introduite dans [ACD93]. Tout comme MITL peut être vue comme une extension de la logique linéaire non temporisée LTL, TCTL est considérée comme une extension de la logique arborescente non temporisée CTL. Nous utilisons ici la syntaxe et la sémantique telle qu'elles ont été présentées dans [TY01, Tri98, Bou02]. Cette définition est, bien entendu, équivalente à celle donnée dans [ACD93].

Syntaxe

La syntaxe de TCTL est proche de la syntaxe de MITL donnée dans la section précédente. La différence résulte principalement dans le fait que, TCTL étant une logique arborescente, tous les opérateurs temporels comme *Until* – dont la notation en TCTL est `U` – doivent être précédés par un quantificateur existentiel `E` ou universel `A` sur les chemins. Les formules TCTL sont donc définies selon la grammaire suivante :

$$\varphi ::= \texttt{true} \mid ap \mid \neg\varphi \mid \varphi \vee \varphi \mid \texttt{E}\varphi\texttt{U}_I\varphi \mid \texttt{A}\varphi\texttt{U}_I\varphi$$

où I est un intervalle de $\mathbb{R}^+$ à bornes entières ou infinies et ap est une proposition atomique.

Tout comme pour MITL, les opérateurs $\Diamond$ et $\Box$ peuvent être définis à partir de cette syntaxe. Ils sont notés respectivement `F` et `G`, et sont également toujours précédés des quantificateurs `E` ou `A` :

- $\texttt{EF}_I\varphi \equiv \texttt{E}(\texttt{true}\texttt{U}_I\varphi)$,
- $\texttt{AF}_I\varphi \equiv \texttt{A}(\texttt{true}\texttt{U}_I\varphi)$,
- $\texttt{EG}_I\varphi \equiv \neg(\texttt{AF}_I(\neg\varphi))$,
- $\texttt{AG}_I\varphi \equiv \neg(\texttt{EF}_I(\neg\varphi))$.

Sémantique

TCTL étant une logique arborescente, elle est interprétée sur les configurations d'un automate temporisé. Etant donnée une configuration s, les formules TCTL sont donc interprétées sur s de la manière suivante :

- $s \models \texttt{true}$ est vrai,
- $s \models ap$ ssi $ap \in L(\texttt{disc}(s))$,
- $s \models \neg\varphi$ ssi il n'est pas vrai que $s \models \varphi$,
- $s \models \varphi_1 \vee \varphi_2$ ssi $s \models \varphi_1$ ou $s \models \varphi_2$,
- $s \models \texttt{E}\varphi_1\mathcal{U}_I\varphi_2$ ssi $\exists \rho = s \xrightarrow{t_1} s_2 \xrightarrow{e_1} s_3 \cdots$ tel que $\texttt{time}(\rho) = \infty$ et $\exists i$ t.q. $\texttt{time}(\rho, i) \in I$ et $(\rho, i) \models \varphi_2$ et $\forall j < i, (\rho, j) \models \varphi_1$,
- $s \models \texttt{A}\varphi_1\mathcal{U}_I\varphi_2$ ssi $\forall \rho = s \xrightarrow{t_1} s_2 \xrightarrow{e_1} s_3 \cdots$ tel que $\texttt{time}(\rho) = \infty$ et $\exists i$ t.q. $\texttt{time}(\rho, i) \in I$ et $(\rho, i) \models \varphi_2$ et $\forall j < i, (\rho, j) \models \varphi_1$.

Intuitivement, une formule $\texttt{E}\varphi_1\mathcal{U}_I\varphi_2$ est vraie pour une configuration s s'il existe une exécution issue de s contenant une configuration s' satisfaisant φ_2 et atteinte durant l'intervalle de temps I, et que toutes les configurations rencontrées avant s' –y compris s – satisfont φ_1. Une formule $\texttt{A}\varphi_1\mathcal{U}_I\varphi_2$ est vraie pour une configuration s si la même condition est remplie pour tous les chemins issus de s.

Tout comme pour MITL, la relation de satisfaction des formules TCTL est étendue aux automates temporisés. Une formule TCTL φ est valide sur un automate temporisé $A = \langle Q, q_0, \mathit{Labels}, X, T, \texttt{Invar}, L\rangle$ si toutes ses configurations satisfont φ.

EXEMPLE 3.2. Les propriétés P_2 et P_5 du passage à niveau sont exprimées respectivement en TCTL par $\texttt{AG}_{\geq 0}(in \Rightarrow \neg is\_up)$ et $\texttt{AG}_{\geq 0}(is\_up \wedge c_1 \Rightarrow \texttt{AF}_{[0,2[}is\_down)$.

3.2 Vérification par model-checking des propriétés des modèles temporisés

Une fois les propriétés du système formalisées, il est possible d'appliquer des algorithmes de model-checking pour assurer que le modèle du système satisfait bien ces propriétés. Suivant le formalisme utilisé pour l'expression des propriétés, les techniques et algorithmes de model-checking diffèrent. Notamment, l'algorithme de model-checking pour MITL est différent de celui utilisé pour TCTL.

3.2.1 Model-checking MITL

L'algorithme de model-checking pour MITL a été présenté dans [AFH96]. Considérons un automate temporisé A et une formule MITL φ. Vérifier que A satisfait la formule φ

consiste à s'assurer que toutes les exécutions de A satisfont φ.

Pour résoudre ce problème, l'algorithme proposé dans [AFH96] consiste à construire un automate de Büchi temporisé (plus précisément, c'est une condition de Büchi généralisée) reconnaissant le langage de la négation de φ. Appelons cet automate $B_{\neg\varphi}$. L'algorithme consiste alors à construire le produit $A \times B_{\neg\varphi}$ et à tester que le langage reconnu par ce produit est vide.

Nous ne donnons pas ici les détails pour la construction de $\mathcal{B}_{\neg\varphi}$, ni l'algorithme pour tester le vide d'un automate. Notons tout de même que les automates de Büchi temporisés construits pour le model-checking MITL ont une syntaxe quelque peu différentes de ceux que nous avons présentés précédemment, comme le montre l'exemple 3.3. Les états sont ici décorés, non par un ensemble de propositions atomiques, mais par une contrainte propositionnelle sur ces propositions atomiques. Le produit d'un tel automate $\mathcal{B}_{\neg\varphi}$ avec un automate temporisé A permettra de restreindre les exécutions de A en ne gardant que celles qui satisfont $\mathcal{B}_{\neg\varphi}$ (si de telles exécutions existent). Cette satisfaction est définie de la manière suivante (nous reprenons ici la définition donnée dans [Tri98]). Une exécution ρ de A satisfait $\mathcal{B}_{\neg\varphi}$ s'il existe une exécution ρ' de $\mathcal{B}_{\neg\varphi}$ telle que $\forall i \geq 0$,

- $\texttt{time}(\rho,(\rho,i)) = \texttt{time}(\rho',(\rho',i))$,
- $L(\texttt{disc}((\rho,i))) \models_p L(\texttt{disc}((\rho',i)))$, où $L(\texttt{disc}((\rho',i)))$ est une contrainte propositionnelle donnant le décor de la configuration (ρ',i).

L'opérateur de satisfaction $\models_p$ est ici défini de manière usuelle sur les contraintes propositionnelles :

- $L(q) \models_p ap$ si $ap \in L(q)$, où ap est une proposition atomique,
- $L(q) \models_p p_1 \wedge p_2$ si $L(q) \models_p p_1$ et $L(q) \models_p p_2$,
- $L(q) \models_p \neg p$ s'il n'est pas vrai que $L(q) \models_p p$.

La taille du produit $A \times B_{\neg\varphi}$ est exponentielle dans la taille φ, et polynomiale dans la taille de A. La complexité du test du vide étant PSPACE, cela donne le résultat suivant pour la complexité du model-checking MITL.

THÉORÈME 2 (COMPLEXITÉ DU MODEL-CHECKING MITL [AFH96]) *Le problème du model-checking pour MITL est EXPSPACE-complet.*

EXEMPLE 3.3. Reprenons la propriété de vivacité bornée "si le contrôleur reçoit un signal d'approche du train, alors la barrière sera fermée au maximum dans les deux unités de temps qui suivent". L'expression de cette propriété en MITL est $\Box_{\geq 0}(is\_up \wedge c_1 \Rightarrow \Diamond_{[0,2[} is\_down)$. L'automate de Büchi temporisé construit pour la négation de cette formule est donnée par la Fig. 3.1. Notons que nous ne faisons apparaître aucun nom d'actions sur les transitions de cet automate car ces informations ne sont pas utilisées pour le model-checking.
Un automate temporisé satisfait cet automate de Büchi s'il possède une exécution qui passe par un état satisfaisant $is\_up \wedge c_1$, puis qui ne passe que par des états qui ne satisfont pas is_down pendant les deux unités de temps qui suivent.

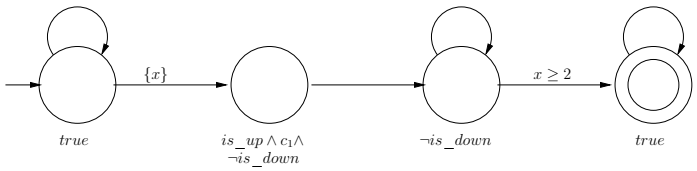


FIG. 3.1 – Automate de Büchi temporisé représentant la négation de la propriété P_5

3.2.2 Model-checking TCTL

L'algorithme de model-checking pour TCTL a été tout d'abord décrit dans [ACD93]. Il est basé sur une construction *a priori* du graphe des régions d'un automate temporisé sur lequel on souhaite vérifier des propriétés.

La méthode que nous allons décrire provient de [HNSY94, Yov98], et permet d'éviter de passer par l'automate des régions pour mener la vérification. Elle est basée sur la construction de *l'ensemble caractéristique* d'une formule TCTL φ. Cet ensemble caractéristique contient l'ensemble des configurations (*étendues*) satisfaisant φ. A partir de là, un automate satisfait φ si toutes ses configurations satisfont φ. Une configuration étendue est un couple $((q,v),w)$, où (q,v) est une configuration de l'automate temporisé, et $w : Z \rightarrow \mathbb{R}^+$ est une fonction associant aux horloges de Z une valeur. L'ensemble d'horloges Z est disjoint de l'ensemble d'horloges de l'automate. Il est utilisé dans la construction de l'ensemble caractéristique des formules de type $E\_U_I\_$ ou $A\_U_I\_$[9], où une horloge $z \in Z$ est associée à chaque opérateur $E\_U_I\_$ ou $A\_U_I\_$. Ces horloges permettent de mesurer le temps écoulé à partir du premier opérande de $E\_U_I\_$ ou $A\_U_I\_$.

L'ensemble caractéristique d'une formule φ est noté $[[\varphi]]$. Pour les formules TCTL ne contenant pas les opérateurs $E\_U_I\_$ ou $A\_U_I\_$, ces ensembles caractéristiques sont construits inductivement sur la structure de la formule de la manière suivante (cette définition provient de [Bou02]) :

- $[[\texttt{true}]] = \{((q,v),w)\}$
- $[[ap]] = \{((q,v),w) \mid ap \in L(q)\}$
- $[[\neg\varphi]] = [[\texttt{true}]] \backslash [[\varphi]]$
- $[[\varphi_1 \vee \varphi_2]] = [[\varphi_1]] \cup [[\varphi_2]]$

La construction pour l'opérateur $E\_U_I\_$ est la suivante :

$$[[E\varphi_1 U_I \varphi_2]] = EU([z := 0][[\varphi_1]], [[\varphi_2]] \cap \{((q,v),w) | w(z) \in I\})$$

où z est l'horloge associée à l'opérateur U_I considéré et $EU(F_1,F_2) = \bigcup_{i\geq 0} E_i$ avec :

$$\begin{cases} E_0 = F_2 \\ E_{i+1} = \texttt{Pred}[F_1](E_i) \cup \texttt{Pred}(E_i) \end{cases}$$

où $\texttt{Pred}[F_1](E_i)$ représente l'ensemble des configurations permettant d'atteindre E_i en laissant passer un temps t, tel que toutes les configurations atteintes par un temps inférieur

[9] la notation "_" représente ici un opérande (formule TCTL) des opérateurs $E\_U_I\_$ et $A\_U_I\_$.

à t soient encore dans F_1. $\texttt{Pred}(E_i)$ représente l'ensemble des configurations permettant d'atteindre E_i en prenant une transition discrète.
Nous ne donnons pas ici la construction pour $A\_U_I\_$, qui est un peu plus compliquée. Elle peut néammoins être trouvée en détail dans [HNSY94]. Le théorème suivant donne la complexité du problème du model-checking pour TCTL.

THÉORÈME 3 (COMPLEXITÉ DU MODEL-CHECKING TCTL [ACD93]) *Le problème du model-checking pour TCTL est PSPACE-complet.*

3.2.3 Outils

Nous présentons à présent les principaux outils de model-checking existants pour les systèmes temporisés. Ces outils se focalisent sur le model-checking de propriétés exprimées en TCTL ou dans un fragment de TCTL. A notre connaissance, il n'existe pas d'outils implantant la technique de model-checking pour MITL.

KRONOS

KRONOS est un outil de vérification de systèmes temporisés. Il a été développé au laboratoire VERIMAG à Grenoble. Les modèles sont donnés sous forme de réseaux d'automates temporisés, synchronisés selon un opérateur de composition (celui présenté dans la section 4.1.1) et les propriétés à vérifier sont exprimées en TCTL et vérifiées par model-checking. De nombreux articles traitent de cet outil et des études de cas qui ont été réalisées, notamment [DY95, DOTY96, Yov97, BDM+98]. Il est disponible à l'adresse :

`http://www-verimag.imag.fr/TEMPORISE/kronos/`

UPPAAL

UPPAAL est développé par les universités d'Aalborg au Danemark et d'Uppsala en Suède. Les systèmes temporisés sont modélisés à l'aide de réseaux d'automates temporisés dont la syntaxe est étendue, avec notamment la possibilité d'utiliser des variables entières bornées et de définir des transitions ou des états urgents. UPPAAL permet principalement de vérifier des propriétés d'atteignabilité ou de vivacité bornée. Ces propriétés peuvent être exprimées formellement en utilisant un fragment de TCTL (qui n'autorise pas les emboîtements d'opérateurs temporels).
Outre ses possibilités au niveau de la vérification, notons également qu'UPPAAL dispose d'une interface graphique très conviviale, et propose un module de simulation du modèle créé. Tout comme KRONOS, de nombreux articles traitent du fonctionnement d'UPPAAL et des études de cas qui ont été menées. Citons entre autres [BGK+96, LPY97a, LPY97b, HSLL97, BDL+01]. UPPAAL est disponible à l'adresse suivante :

`http://www.uppaal.com`

HyTech

HyTech a été réalisé par Henzinger, Ho et Wong-Toi à Berkeley, l'université de Californie. C'est un outil de vérification pour les automates hybrides linéaires (et donc en particulier les automates temporisés). Il considère des réseaux d'automates se synchronisant sur les actions de même nom. Il vérifie essentiellement des propriétés d'atteignabilité et de sûreté. HyTech permet également de traiter des systèmes paramétrés. Notons de plus qu'une interface graphique a été fournie par l'équipe d'Uppaal afin de pouvoir décrire graphiquement les automates en entrée de HyTech. De nombreuses publications traitent de HyTech, par exemple [HH95, HHWT95, HHWT97]. L'outil est disponible à l'adresse suivante :

`http://embedded.eecs.berkeley.edu/research/hytech/`

Cmc

L'outil Cmc (Compositional Model-Checking) [LL98] a été développé au LSV à l'ENS Cachan par François Laroussinie. Il permet de vérifier des réseaux d'automates temporisés, où les propriétés sont spécifiées en utilisant la logique L_ν [LL95]. Il utilise pour cela la méthode de model-checking compositionnel décrite dans [LL95]. Succintement, étant donné un réseau d'automates $A_1||A_2||\cdots||A_{n-1}||A_n$, et une propriété P à vérifier sur ce modèle, le model-checking compositionnel consiste à modifier la propriété P en *incorporant* successivement les automates $A_n, A_{n-1}, \cdots, A_2, A_1$ à cette propriété. A chaque itération, la propriété obtenue est celle qui doit être vérifiée sur les A_i restants, de manière à ce que le résultat de la vérification de cette nouvelle propriété soit le même que le résultat qu'on aurait obtenu avec la propriété initiale sur le réseau complet des A_i. A la fin du processus, il s'agira de pouvoir vérifier que l'automate qui n'effectue aucune action, appelé *Nil*, satisfait la propriété obtenue après incorporation de tous les A_i. Cette méthode permet de ne jamais avoir à traiter le modèle global composé de tous les A_i. L'outil Cmc est disponible à l'adresse suivante :

`http://www.lsv.ens-cachan.fr/~fl/cmcweb.html`

3.3 Conclusion

Nous avons présenté deux formalismes permettant de spécifier les propriétés des systèmes temporisés : la logique linéaire Mitl et la logique arborescente Tctl, ainsi que les techniques de vérification de ces propriétés par model-checking et des outils les mettant en œuvre.
Comme nous l'avons expliqué en introduction de ce document, la complexité du model-checking, notamment pour les propriétés linéaires, rend la méthode difficile à appliquer sur des systèmes de grande taille. Nous avons choisi de modéliser les systèmes temporisés à base de composants, et d'utiliser des méthodes de développement incrémental pour vérifier les propriétés, afin de palier au problème d'explosion combinatoire.
Dans le chapitre suivant, nous présentons donc les deux opérateurs de composition temporisés que nous considérons dans ce document, et présentons en détail les deux méthodes

de développement incrémental que nous avons évoquées en introduction de ce document : le raffinement et l'intégration de composants.

4

Modélisation de systèmes temporisés basée sur les composants

Sommaire

Lorsque l'on considère des systèmes complexes, il peut s'avérer difficile de créer directement un modèle complet du système. Une solution de plus en plus répandue, aussi bien dans le cadre temporisé que dans le cadre non temporisé, consiste à commencer par découper le système en un ensemble de composants, où chaque composant représentera un *morceau* du système à modéliser. Ainsi, on commencera d'abord par modéliser chacun des composants. Le modèle complet pourra alors être obtenu en assemblant tous ces composants en tenant compte de leurs interactions et synchronisations. Ce type de modélisation

est appelée *modélisation basée sur les composants.*
L'assemblage des composants est réalisé à l'aide d'opérateurs de composition. Des opérateurs de composition sont notamment définis dans la littérature des algèbres de processus. Les algèbres de processus sont un formalisme pour la spécification de systèmes concurrents. La composition parallèle en est donc une règle importante. On peut citer les paradigmes de composition des deux algèbres de processus suivants : CSP (Communicating Sequential Processes) [Hoa85] ou encore CCS (Calculus of Communicating Systems) [Mil89].
Dans CSP, les actions de deux processus communicants sont séparés en deux ensembles, l'un correspondant aux actions qui se synchronisent et l'autre aux actions qui sont entrelacées. Ainsi, dans le processus résultant de la composition parallèle de ces deux processus, les actions synchronisées ne sont jamais entrelacées et ne sont effectuées que si les deux processus sont prêts à effectuer une même action synchronisée. Dans CCS, chaque action possède une action complémentaire. Une synchronisation a lieu lorsque l'un des processus peut effectuer une action et l'autre l'action complémentaire. L'action résultant de cette synchronisation est appelée τ et est considérée comme une action interne du processus obtenu qui ne peut pas être synchronisée avec une autre action[10]. De plus, dans CCS, toutes les actions sont entrelacées, même lorsqu'une synchronisation a lieu entre deux actions. Ceci est fait pour prendre en compte le fait qu'un troisième processus peut vouloir se synchroniser avec l'une des actions se synchronisant déjà. Notons qu'un opérateur de restriction est également défini dans CCS pour pouvoir supprimer ce type de transitions entrelacées.

Nous présentons dans la section 4.1 deux opérateurs de composition définis pour les automates temporisés. Le premier utilise le paradigme de composition de CSP, tandis que le second, défini dans [Bor98, BST97, BS00], peut être vu comme une extension de la composition parallèle de CCS, donnant priorité aux actions synchronisées par rapport à l'entrelacement. De plus, il possède la propriété de construire des automates non bloquants lorsqu'il est appliqué entre des automates sans blocage (ce qui n'est pas le cas en général avec le premier opérateur).

Du point de vue de la vérification de ces systèmes à composants, les méthodes classiques consistent à construire le modèle complet du système en assemblant tous ses composants, puis à mener la vérification sur le modèle obtenu. Or, lorsque l'on considère des systèmes complexes, ce modèle possède généralement un grand nombre d'états. Les méthodes de vérification comme le model-checking, dont le principe est d'explorer l'espace d'états du modèle, peuvent donc être difficiles (voire impossibles) à appliquer.
Les méthodes de modélisation incrémentale peuvent représenter une solution à ce problème. La démarche consiste à construire le modèle complet du système non plus directement, mais pas à pas. Une vue abstraite du système est tout d'abord considérée, puis elle est progressivement *restreinte* par l'ajout de détails, chaque ajout de détails représentant une étape dans le processus de modélisation. La vérification de propriétés peut alors être menée à chacune de ces étapes, sur des modèles de taille plus petite que le modèle complet.

[10]Dans les chapitres suivants, nous utiliserons également la notion d'action τ pour dénoter les actions non-observables d'un système, à ne pas confondre avec l'utilisation de τ dans CCS.

Bien évidemment, le processus de modélisation doit également garantir que les propriétés vérifiées à un certain niveau de détails sont préservées lors de la suite du processus. Nous présenterons dans les sections 4.2 et 4.3 deux méthodes de modélisation incrémentale : tout d'abord par une méthode spécifique aux modèles à base de composants, *l'intégration de composants*, puis une méthode plus générale, *le raffinement*.

4.1 Opérateurs de composition pour les systèmes temporisés à composants

Nous commençons tout d'abord par présenter deux opérateurs de composition que nous utiliserons tout au long de ce document. Considérons un système temporisé modélisé par un ensemble de composants, $C_1, C_2, \cdots, C_n$. Les interactions entre les différents composants sont données sous la forme d'actions qui doivent se synchroniser. Chaque composant possède des actions synchronisées (celles qui doivent se synchroniser avec des actions d'autres composants) et des actions locales qui lui sont propres.

4.1.1 Composition parallèle *à la CSP*

Cette composition parallèle, notée $||$, utilise le paradigme de composition de CSP, en quelque sorte étendu pour prendre en compte des aspects temporisés. Elle agit entre deux composants C_1 et C_2 modélisés par des automates temporisés, et est définie comme un produit synchronisé où les synchronisations sont effectuées sur les actions de même étiquette, les actions locales de chaque composant sont entrelacées et le temps passe de manière synchrone entre tous les composants.

DÉFINITION 6 (COMPOSITION PARALLÈLE) *Considérons deux ensembles de propositions atomiques* Props_1 *et* Props_2. *Soient* $A_1 = \langle \mathrm{Q}_1, \mathrm{q}_{0_1}, \mathrm{Labels}_1, \mathrm{X}_1, \mathrm{T}_1, \mathtt{Invar}_1, \mathrm{L}_1 \rangle$ *et* $A_2 = \langle \mathrm{Q}_2, \mathrm{q}_{0_2}, \mathrm{Labels}_2, \mathrm{X}_2, \mathrm{T}_2, \mathtt{Invar}_2, \mathrm{L}_2 \rangle$ *deux automates temporisés dont les états sont décorés respectivement par des propositions atomiques de* Props_1 *et* Props_2. *On suppose que leurs ensembles d'horloges sont disjoints* ($X_1 \cap X_2 = \emptyset$) *et leurs alphabets d'actions ne le sont pas* ($\mathrm{Labels}_1 \cap \mathrm{Labels}_2 \neq \emptyset$). *La composition parallèle de* A_1 *et* A_2, *notée* $A_1||A_2$, *crée un nouvel automate temporisé* A *sur* $Labels_1 \cup Labels_2$ *dont les composantes sont définies de la manière suivante :*

- *Les états de* A *sont des couples* (q_1, q_2) *où* q_1 *est un état de* A_1 *et* q_2 *est un état de* A_2. *Le décor d'un état* (q_1, q_2) *est* $L(q_1) \cup L(q_2)$ *et son invariant est donné par* $\mathtt{Invar}(q_1) \wedge \mathtt{Invar}(q_2)$. *L'état initial de* A *est le couple* (q_{0_1}, q_{0_2}).
- *L'alphabet de* A *est* $\mathrm{Labels}_1 \cup \mathrm{Labels}_2$.
- *L'ensemble d'horloges de* A *est* $X_1 \cup X_2$.
- *Les transitions de* A *sont de deux types. Etant données une transition* $e_1 = (q_1, g_1, a_1, \gamma_1, q_1')$ *de* A_1 *et une transition* $e_2 = (q_2, g_2, a_2, \gamma_2, q_2')$ *de* A_2,
  1. *Si* $a_1 = a_2$, *on obtiendra dans* A *la transition synchronisée* $e_1||e_2$ *définie par*
  $$e_1||e_2 = ((q_1, q_2), g_1 \wedge g_2, a_1, \gamma_1 \cup \gamma_2, (q_1', q_2'))$$
  2. *Si* $a_1 \neq a_2$, *on obtiendra dans* A *les deux transitions libres*

$$e = ((q_1, q_2), g_1, a_1, \gamma_1, (q_1', q_2))$$
$$e' = ((q_1, q_2), g_2, a_2, \gamma_2, (q_1, q_2'))$$

EXEMPLE 4.1. Reprenons l'exemple du passage à niveau. Ce système est modélisé par au moins trois composants : un ou plusieurs trains, la barrière du passage à niveau et son contrôleur. La composition parallèle des composants train, barrière et contrôleur définit l'automate temporisé donné par la Fig. 4.1.

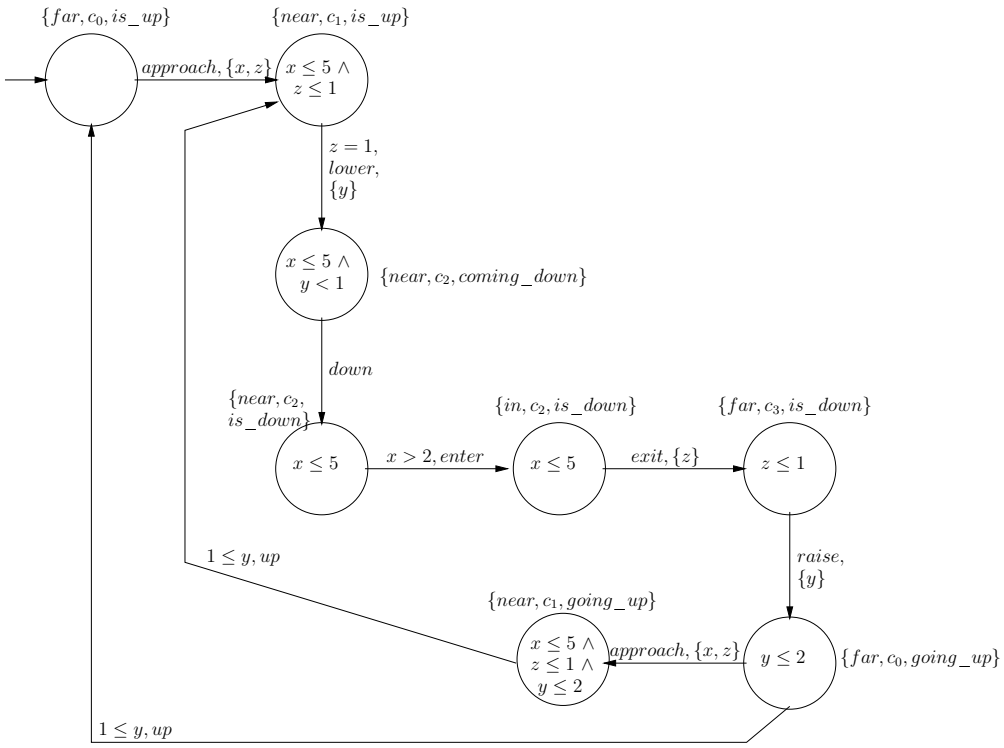


FIG. 4.1 – Composition parallèle du train, de la barrière et du contrôleur

La définition 6 est une définition syntaxique. Nous l'exprimons également au niveau sémantique, en considérant deux graphes sémantiques $\mathcal{G}_1$ et $\mathcal{G}_2$. La composition parallèle de $\mathcal{G}_1$ et $\mathcal{G}_2$ crée un graphe sémantique $\mathcal{G}$, dont les configurations sont des couples (s_1, s_2), où s_1 et s_2 sont des configurations de $\mathcal{G}_1$ et $\mathcal{G}_2$ respectivement. La configuration initiale est le couple (s_{0_1}, s_{0_2}), où s_{0_1} et s_{0_2} sont respectivement les configurations initiales de $\mathcal{G}_1$ et $\mathcal{G}_2$. Les transitions sont définies de la manière suivante :

- Soit $t \in \mathbb{R}^+$. Si (s_1, s_2) est une configuration de $\mathcal{G}$, $s_1 \xrightarrow{t} s_1'$ est une transition de $\mathcal{G}_1$ et $s_2 \xrightarrow{t} s_2'$ est une transition de $\mathcal{G}_2$, alors $(s_1, s_2) \xrightarrow{t} (s_1', s_2')$ est une transition de $\mathcal{G}$,

- Si (s_1, s_2) est une configuration de $\mathcal{G}$, que $s_1 \xrightarrow{e_1} s_1'$ est une transition de $\mathcal{G}_1$, $s_2 \xrightarrow{e_2} s_2'$ est une transition de $\mathcal{G}_2$ et $\texttt{label}(e_1) = \texttt{label}(e_2)$, alors $(s_1, s_2) \xrightarrow{e_1 || e_2} (s_1', s_2')$ est une transition de $\mathcal{G}$,

- Si (s_1, s_2) est une configuration de $\mathcal{G}$, $s_1 \xrightarrow{e_1} s_1'$ (respectivement $s_2 \xrightarrow{e_2} s_2'$) est une transition de $\mathcal{G}_1$ (resp. de $\mathcal{G}_2$), alors $(s_1, s_2) \xrightarrow{e_1} (s_1', s_2)$ (resp. $(s_1, s_2) \xrightarrow{e_2} (s_1, s_2')$) est

une transition de $\mathcal{G}$.

LEMME 4 *[Tri98] Le graphe sémantique de la composition parallèle de A_1 et A_2 (c-à-d $\mathcal{G}(A_1||A_2)$) est identique au graphe obtenu par composition parallèle des graphes sémantiques de A_1 et A_2 (c-à-d $\mathcal{G}(A_1)||\mathcal{G}(A_2)$).*

L'inconvénient principal de ce type de composition est qu'en général, elle introduit des blocages. Si on considère deux automates temporisés sans blocage, elle ne garantit pas que l'automate résultant $A_1||A_2$ soit également sans blocage. Pour cette raison, d'autres opérateurs ont été définis et possèdent cette propriété. Nous en présentons particulièrement un dans la section suivante.

4.1.2 Composition parallèle *à la CCS* avec priorités et / ou modes de synchronisation

Cette composition parallèle, que nous noterons | a été définie dans [Bor98, BST97, BS00]. L'objectif était de définir une composition parallèle, toujours basée sur des synchronisations entre les actions des composants, garantissant les trois propriétés suivantes :

1. la préservation de la *réactivité temporelle* lors de la composition : la réactivité temporelle est le fait de garantir que, dans chaque état, soit on peut déclencher une action, soit on peut laisser le temps s'écouler,
2. la préservation de l'activité : si une action peut être effectuée après un certain temps dans un composant, alors une action pourra également être effectuée dans le produit après un certain temps (qui n'est pas nécessairement le même),
3. le progrès maximal : le progrès maximal est le fait de favoriser les synchronisations plutôt que l'entrelacement des actions locales des composants.

Cette composition a été initialement définie sur des automates temporisés avec deadlines, avec la possibilité de prendre en compte les priorités pouvant éventuellement exister entre les actions. On peut là aussi la présenter comme un produit synchronisé où le temps s'écoule de manière synchrone et toutes les actions sont entrelacées, aussi bien les actions locales que les actions qui se synchronisent. Les synchronisations sont données par une fonction de synchronisation, notée |, qui, étant données deux actions, définit l'action synchronisée résultant de la synchronisation de ces deux actions ou le symbole spécial $\perp$ si les deux actions ne se synchronisent pas.

$$| : Labels_1 \times Labels_2 \rightarrow Labels_{sync} \cup \{\perp\}$$

$Labels_1$ et $Labels_2$ sont les alphabets respectifs de chacun des composants, et $Labels_{sync}$ est un alphabet disjoint de $Labels_1$ et $Labels_2$ contenant les actions résultant des synchronisations. Comme toutes les actions sont entrelacées, il est à noter que les deux actions participant à une synchronisation apparaissent également dans la composition, en plus de la synchronisation. Une priorité infinie est donc donnée à l'action synchronisée pour garantir la propriété de progrès maximal, comme le montre la Fig. 4.2. La préservation de l'activité est assurée quant à elle par l'entrelacement. On peut donc voir cette composition

comme une composition *à la CCS*, donnant en plus la priorité aux actions synchronisées par rapport aux actions entrelacées correspondantes, sans utiliser d'opérateur de restriction "coupant" des transitions (et pouvant donc introduire des blocages). La définition 7 présente formellement cette composition parallèle.

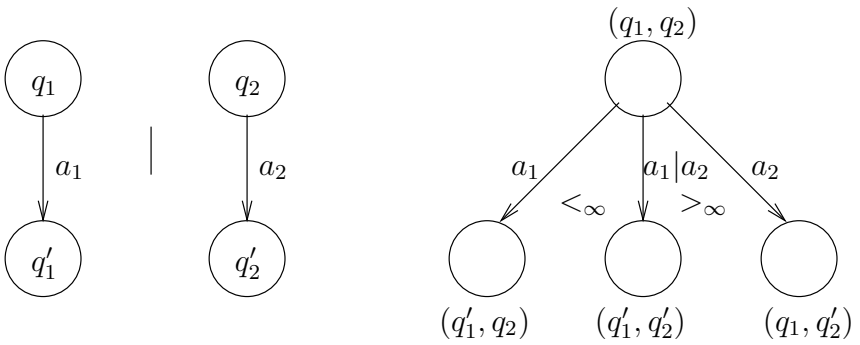


FIG. 4.2 – Synchronisation pour la composition parallèle |

DÉFINITION 7 (COMPOSITION PARALLÈLE NON BLOQUANTE |) *Considérons deux automates temporisés avec deadlines* $A_1 = \langle \mathrm{Q}_1, \mathrm{q}_{0_1}, \mathrm{Labels}_1, \mathrm{X}_1, \mathrm{T}_1, \mathrm{L}_1 \rangle$ *et* $A_2 = \langle \mathrm{Q}_2, \mathrm{q}_{0_2}, \mathrm{Labels}_2, \mathrm{X}_2, \mathrm{T}_2, \mathrm{L}_2 \rangle$ *dont les ensembles d'horloges et les alphabets sont disjoints. Un ordre de priorité* $<^1$ *(resp.* $<^2$*) peut également être défini entre les actions de* A_1 *(resp.* A_2*). On dispose d'une fonction de synchronisation* $| : \mathrm{Labels}_1 \times \mathrm{Labels}_2 \to \mathrm{Labels}_{sync}$ *entre les actions de* A_1 *et* A_2*. La composition parallèle de* A_1 *et* A_2*, notée* $A_1|A_2$*, crée un automate temporisé avec deadlines* $A = \langle \mathrm{Q}, \mathrm{q}_0, \mathrm{Labels}, \mathrm{X}, \mathrm{T}, \mathrm{L} \rangle$ *dont les composantes sont définies de la manière suivante :*

- *Les états de* A *sont des couples* (q_1, q_2) *où* $q_1 \in Q_1$ *et* $q_2 \in Q_2$*. L'état initial est le couple* (q_{0_1}, q_{0_2})*. Le décor d'un couple* (q_1, q_2) *est donné par* $L_1(q_1) \cup L_2(q_2)$*.*
- $\mathrm{Labels} = \mathrm{Labels}_1 \cup \mathrm{Labels}_2 \cup \mathrm{Labels}_{sync}$ *est l'alphabet des noms d'actions,*
- $X = X_1 \cup X_2$ *est l'ensemble d'horloges,*
- T *est l'ensemble des transitions. Soit un état* (q_1, q_2) *de* A*,* $e_1 = (q_1, g_1, d_1, a_1, \gamma_1, q'_1)$ *une transition de* A_1 *et* $e_2 = (q_2, g_2, d_2, a_2, \gamma_2, q'_2)$ *une transition de* A_2*,*

  1. *on obtiendra les deux transitions libres*

$$e = ((q_1, q_2), g_1, d_1, a_1, \gamma_1, (q'_1, q_2))$$
$$e' = ((q_1, q_2), g_2, d_2, a_2, \gamma_2, (q_1, q'_2))$$

  2. *Si* $a_1|a_2 \neq \perp$*, on obtiendra également la transition synchronisée*

$$e_1|e_2 = ((q_1, q_2), g_1 \wedge g_2, max(d_1, d_2), a_1|a_2, \gamma_1 \cup \gamma_2, (q'_1, q'_2))$$

  où $max(d_1, d_2)$ *est défini par le fait que* $\lambda < \delta < \varepsilon$*.*

Un ordre de priorités $<^{sync}$ *est également défini pour garantir le progrès maximal et prendre en compte les ordres* $<^1$ *et* $<^2$*. C'est l'ordre minimal induit par les règles :*

- $a_1, a_2 \in \mathrm{Labels}_1$ *(resp.* Labels_2*) et* $a_1 <^1_k a_2$ *(resp.* $a_1 <^2_k a_2$*) alors* $a_1 <^{sync}_k a_2$*,*
- $a_1 \in \mathrm{Labels}_1$ *et* $a_2 \in \mathrm{Labels}_2$ *et* $a_1|a_2 \neq \perp$ *alors* $a_1 <^{sync}_\infty a_1|a_2$ *et* $a_2 <^{sync}_\infty a_1|a_2$*,*

– $a_1, a_2 \in \text{Labels}_1$ *(resp.* Labels_2*) et* $a_1 <_k^1 a_2$ *alors* $\forall a_3 \in \text{Labels}_2$ *(resp.* Labels_1*) t.q.* $a_1|a_3 \neq \perp$ *et* $a_2|a_3 \neq \perp$*, on a* $a_1|a_3 <_k^{sync} a_2|a_3$.

Cette composition peut également être étendue en utilisant différents modes de synchronisation [BST97]. A chaque action est alors associé un mode de synchronisation parmi les suivants : AND, MIN ou MAX. Le mode AND correspond à une synchronisation traditionnelle et se rapporte à celui défini précédemment. Le mode MAX représente une synchronisation avec attente, c-à-d qu'une synchronisation a lieu entre deux actions si l'une des actions est déclenchable et l'autre l'a été dans le passé (la première action attend donc que la seconde puisse avoir lieu pour se synchroniser). Le mode MIN peut être interprété comme une synchronisation avec interruption : lorsqu'une des actions participant à la synchronisation peut être déclenchée, elle force alors la synchronisation, à condition que l'autre action puisse être effectuée dans le futur.
Par rapport à la définition 7, les modifications apportées par ces modes concernent l'obtention des gardes des actions synchronisées :

– Le mode AND étant le mode classique, la garde d'une action synchronisée est toujours $g_1 \wedge g_2$.
– Pour le mode MAX, la garde de l'action synchronisée est $(g_1 \wedge \swarrow g_2) \vee (\swarrow g_1 \wedge g_2)$.
– Pour le mode MIN, la garde de l'action synchronisée est $(g_1 \wedge \nearrow g_2) \vee (\nearrow g_1 \wedge g_2)$.

4.2 Modélisation incrémentale par intégration de composants

Comme nous l'avons dit en introduction de ce chapitre, utiliser des composants s'avère particulièrement utile pour faciliter la phase de modélisation lorsque l'on considère des systèmes complexes. Toutefois, les propriétés à vérifier sur le modèle sont vérifiées sur le modèle complet, obtenu par composition parallèle des différents composants. Lorsque le modèle obtenu est de taille importante, les techniques de model-checking peuvent être difficile à appliquer.
Une manière de contourner ce problème est de modéliser les systèmes de manière incrémentale et de vérifier les propriétés pouvant l'être à chaque étape de cette modélisation, plutôt que de traiter directement le modèle complet. Nous présentons tout d'abord un premier type de modélisation incrémentale, l'intégration de composants.

4.2.1 Principe

Considérons toujours un système modélisé par un ensemble de composants $C_1, C_2, \cdots, C_n$. L'intégration de composants est une méthode de modélisation incrémentale qui consiste à tout d'abord considérer un composant du système (ou un ensemble de composants), par exemple C_1, puis à ajouter successivement à C_1 les autres composants, jusqu'à aboutir au modèle complet $C_1||C_2||\cdots||C_n$. Remarquons que nous utilisons ici la notation $||$ pour dénoter un opérateur de composition quelconque, par exemple l'un de ceux présentés précédemment.

4.2.2 Intégration de composants et vérification de propriétés

L'intérêt de ce type de modélisation incrémentale réside notamment dans son application pour la vérification des propriétés *locales* aux composants ou à des groupes de composants. Par exemple, il pourrait être possible de vérifier les propriétés locales de C_1 uniquement sur ce composant, plutôt que de les vérifier sur le modèle complet. Un composant étant de taille plus petite que le système complet, le coût de la vérification uniquement sur ce composant serait moindre que sur le système complet. Bien-sûr, il est nécessaire que l'intégration des autres composants n'altère pas le résultat de la vérification. En d'autres termes, les propriétés locales doivent être *préservées* par l'intégration de composants. Cette propriété est appelée la *composabilité*. Nous l'illustrons par la Fig. 4.3.

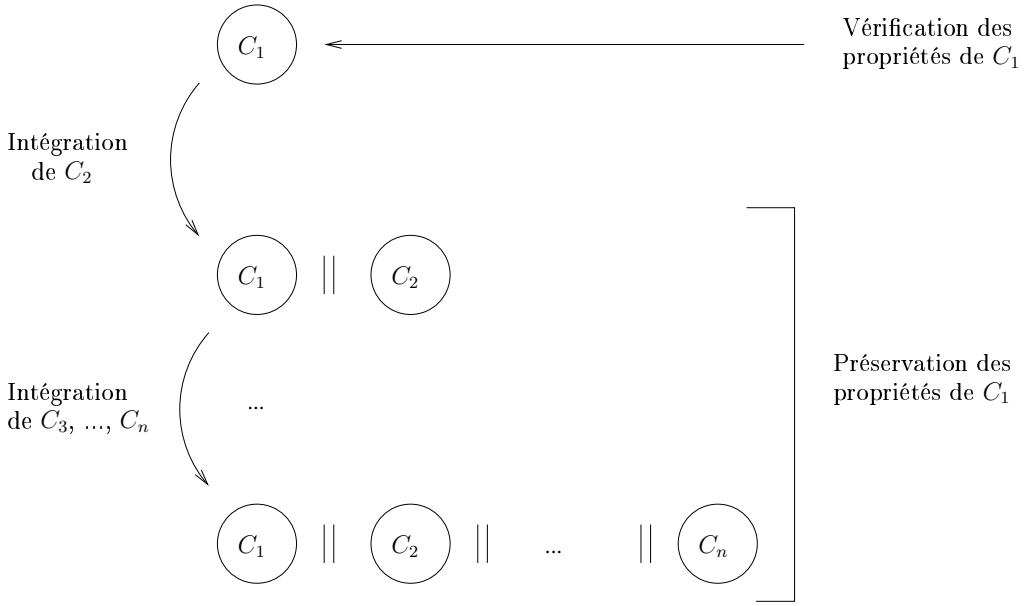


FIG. 4.3 – Intégration de composants et composabilité

4.3 Modélisation incrémentale par raffinement

Le raffinement est un autre type de méthode incrémentale. Pour modéliser un système complexe, cette méthode consiste à considérer tout d'abord un modèle abstrait du système, puis à le raffiner en ajoutant progressivement de nouveaux détails. Nous commençons dans cette section par présenter différents concepts de raffinement introduits dans la littérature, que ce soit dans le cadre non temporisé ou temporisé, avant de présenter la notion que nous considérons. Nous expliquerons ensuite l'intérêt du raffinement dans le cadre particulier des systèmes à composants, et pour la préservation de propriétés. Nous terminerons cette

section en expliquant les raisons pour lesquelles l'intégration de composants, que nous avons présentée précédemment, peut être vue comme un cas particulier de raffinement.

4.3.1 Quelques notions de raffinement

De nombreuses notions de raffinement ont été proposées dans la littérature, dans le cadre de systèmes non temporisés. Une présentation et une comparaison des principales formes de raffinement peut notamment être trouvée dans [GFL05]. Présentons-en certaines avant d'énoncer les notions de raffinement qui ont été définies pour les automates temporisés, et celle que nous considérons.

Raffinement wp de Dijkstra

La notion de *plus faible précondition* (*wp - weakest precondition*) a été introduite par Dijkstra dans le cadre de la vérification de la correction de programmes [Dij76] et s'appuie sur la logique de Hoare. Une expression de la logique de Hoare est un triplet $<P>$ S $<Q>$ où S est un programme, P sa précondition (qui caractérise les états possibles avant l'exécution de S) et Q sa postcondition (qui caractérise les états possibles après exécution de S). Le programme S est dit totalement correct si, à partir de tout état satisfaisant P, S termine et mène dans un état satisfaisant Q. La correction est dite partielle quand la terminaison n'est pas garantie.
Le prédicat *wp(S,Q)* permet de calculer la plus faible précondition garantissant la terminaison de S et permettant d'établir Q après exécution de S. Le programme S est alors correct si $wp(S, Q) \Rightarrow P$.

La notion de raffinement s'exprime alors également en termes de plus faible précondition, de la manière suivante : un programme S est raffiné par un programme S' si $\forall Q, wp(S, Q) \Rightarrow wp(S', Q)$.
De nombreuses notions de raffinement sont basées sur cette sémantique en termes de plus faible précondition, notamment le raffinement des systèmes d'événements B que nous présentons à présent.

Raffinement de systèmes d'événements B

Le langage B est un langage de spécification logico-ensembliste introduit en 1996 par J.-R. Abrial, auquel est associé une méthode de développement combinant preuve et raffinement : la méthode B [Abr96a, Abr97]. Le principe est de modéliser le système incrémentalement par raffinement depuis une spécification abstraite jusqu'à un dernier énoncé appelé *implémentation* qui conduit directement à un programme exécutable par génération de code (dans un langage tel que C par exemple). Chaque étape du processus est *validée* par preuve.
Traditionnellement, la méthode B est utilisée pour modéliser des systèmes ouverts, dont les opérations sont appelées par l'environnement. Le B événementiel [Abr96b] est une extension de la notation B, plus adaptée à la prise en compte de systèmes réactifs fermés. On parle ici de systèmes d'événements B. La figure 4.4 montre les différentes clauses que

comporte un système d'événements (d'autres sont disponibles, mais dans cette présentation succinte, nous nous contentons de présenter les clauses principales et qui doivent obligatoirement apparaître).

Machine ES
Sets set1 ; set2 ; $\cdots$
Variables var1, var2, $\cdots$
Invariant prop(var1,var2,$\cdots$)
Initialisation init(var1,var2,$\cdots$)
Operations
event1 $\widehat{=}$ `select` guard `then` subst `end` ;
$\cdots$
End

FIG. 4.4 – Description d'un système d'événements B

Les clauses **Sets** et **Variables** permettent de déclarer les différents ensembles et variables qui vont être utilisés dans la spécification. La clause **Invariant** permet de décrire des propriétés *statiques* de la spécification, c'est à dire des propriétés devant être respectées dans tout état du système. Elle donne également le typage des variables. La clause **Initialisation** permet de donner une valeur initiale aux variables, tandis que le clause **Operations** permet de détailler tous les événements effectués par le système. Chaque événement dont la garde est vraie est déclenché de manière spontanée. Si plusieurs événements sont activables, l'un d'entre eux est choisi de manière non déterministe. En revanche, si aucun événement n'est déclenchable, le système est bloqué. Chaque événement, ainsi que l'initialisation, est décrit à l'aide de substitutions généralisées, pouvant être vues comme une extension du langage des commandes gardées de Dijkstra [Dij75, Dij76]. Rappelons que l'invariant doit être vérifié dans tout état observable du système, et qu'il doit donc être maintenu par l'application des substitutions définissant l'initalisation et chaque événement du système. Cette vérification est effectuée par preuve.

Comme nous l'avons dit précédemment, la méthode B s'accompagne d'un processus de modélisation incrémentale par raffinement. Syntaxiquement, un système d'événements raffiné est composé des mêmes clauses qu'un système d'événements abstrait, modulo le fait qu'une clause **Refinement** remplace la clause **Machine**, et une nouvelle clause **Refines** apparaît pour indiquer le nom du système d'événements qui est raffiné. De nouvelles informations sont ajoutées à l'invariant du SE raffiné, donnant le lien entre les variables du SE raffiné et celles du SE abstrait. Cette partie de l'invariant est appelé *invariant de collage*. Des obligations de preuve sont définies afin de s'assurer de la correction du raffinement (nous ne donnons ici qu'une description, la définition formelle pouvant être trouvée dans [Abr96a]) :

1. Les ensembles de variables des SE abstrait et raffiné sont disjoints.
2. Tous les événements abstraits sont raffinés (c-à-d, ils existent toujours dans le SE

raffiné).

3. La définition des événements du SE raffiné qui raffinent des événements du SE abstrait n'est pas contradictoire avec la définition de ces événements dans le SE abstrait. Notamment, la garde de l'événement doit être renforcée par le raffinement, afin de pouvoir garantir que s'il peut être déclenché dans le SE raffiné, il pouvait également l'être dans le SE abstrait.
4. Tous les nouveaux événements satisfont les invariants du SE raffiné et du SE abstrait.
5. Les nouveaux événements n'introduisent pas de blocage.
6. Les nouveaux événements n'introduisent pas de nouveaux cycles.

Notons que les obligations de preuve pour les points 5 et 6 ne sont pas générées directement par le prouveur. C'est à l'utilisateur d'ajouter dans la spécification les informations nécessaires à la génération d'obligations de preuve permettant de garantir ces deux points. Pour démontrer l'absence de blocages, l'utilisateur peut ajouter une assertion dans le système raffiné qui spécifie que la disjonction des gardes des événements abstraits implique la disjonction des gardes des événements raffinés. Concernant l'absence de nouveaux cycles, on utilise un variant qui décroit strictement à chaque exécution d'un nouvel événement introduit par le raffinement.

Cette notion de raffinement est à la base du raffinement des systèmes de transitions que nous présentons à présent.

Raffinement de systèmes de transitions

Le raffinement de systèmes de transitions a été défini dans [BJK00]. A l'origine, il a été défini pour exprimer le raffinement de systèmes d'événements B sur les systèmes de transitions (ST) qui représentent leur sémantique. Il adopte donc syntaxiquement les mêmes principes que le raffinement de systèmes d'événements B, à savoir :

1. Le raffinement peut introduire de nouvelles variables, tandis que d'autres variables sont utilisées pour dénoter les variables du ST abstrait (par exemple, ces variables peuvent être obtenues par un simple renommage des variables abstraites). L'ensemble de variables du ST raffiné doit donc être disjoint de l'ensemble de variables du ST abstrait.
2. Le raffinement peut introduire de nouvelles actions, tandis que les actions abstraites du ST abstrait existent toujours sur le ST raffiné.
3. Un prédicat de collage est utilisé pour lier les états du ST raffiné aux états du ST abstrait.

Formellement, cette relation de raffinement a été définie comme une sorte de τ-simulation, qui préserve les propriétés exprimées en LTL [DJK03]. Nous expliquerons cette notion de τ-simulation précisément dans le chapitre suivant.

Cette définition du raffinement de systèmes de transitions est à la base de la notion de raffinement que nous considérons pour les automates temporisés. Toutefois, d'autres notions de raffinement pour les automates temporisés ont été étudiées. Commençons par les présenter avant d'exposer le point de vue que nous adoptons.

4.3.2 Le raffinement pour les automates temporisés

Dans [BGP97a, BGP97b], Bérard, Gastin et Petit considèrent le raffinement d'automates temporisés. Ils étudient notamment la fermeture des classes d'automates temporisés classiques et avec ε-transitions sous l'opération de raffinement. Ils examinent en particulier deux notions de raffinement, qui dépendent de la sémantique considérée pour les actions temporisées : soit les actions sont instantanées, soit elles ont une durée. Dans le premier cas, chaque action est associée au temps auquel elle a été déclenchée. Dans ce cas, une action temporisée (a,t), c'est-à-dire l'action a se déclenchant au temps t, sera raffinée par une séquence d'actions $(a_1,t)(a_2,t)\cdots(a_n,t)$, où la première action a_1 est déclenchée au même moment que l'action abstraite a, et les actions suivantes sont déclenchées instantanément (avec un délai d'attente nul). Dans le cas des actions ayant une durée, les actions sont associées au temps auquel elles se terminent. Le délai qu'une action a utilisé est donc la différence entre le temps auquel elle est associé, et le temps auquel est associé l'action précédente. Dans ce cadre, une action (a,t) de délai d est raffinée par une séquence d'actions $(a_1,t_1)(a_2,t_2)\cdots(a_n,t_n)$ de délais respectifs $d_1,d_2,\cdots,d_n$, où par exemple $d_2 = t_2 - t_1$, tels que $d_1 + d_2 + \cdots + d_n = d$.

Comme nous l'avons présenté dans le chapitre 2, nous considérons des actions instantanées (une action est associée au temps auquel elle est déclenchée), mais le concept de raffinement que nous utilisons se rapproche du concept du raffinement de [BGP97a, BGP97b] pour les actions ayant une durée. En effet, nous considérons qu'une action (a,t) est raffinée par une séquence d'actions $(a_1,t_1)(a_2,t_2)\cdots(a_n,t_n)(a,t)$, où $t_1 \leq t_2 \leq \cdots \leq t_n \leq t$ et t_1 est supérieur au temps associé à l'action qui précédait a dans la version abstraite. L'action raffinée a ne se déclenche donc pas plus tard qu'elle ne le faisait dans sa version abstraite.
Tout comme le raffinement des systèmes de transitions, nous utilisons un prédicat de collage pour lier les états raffinés aux états abstraits. Ce prédicat est une formule propositionnelle de la forme suivante :

$$P_c ::= ap_1 \mid ap_2 \mid \neg p \mid p \vee p$$

où ap_1 et ap_2 sont des propositions atomiques décorant respectivement les états abstraits et les états raffinés.

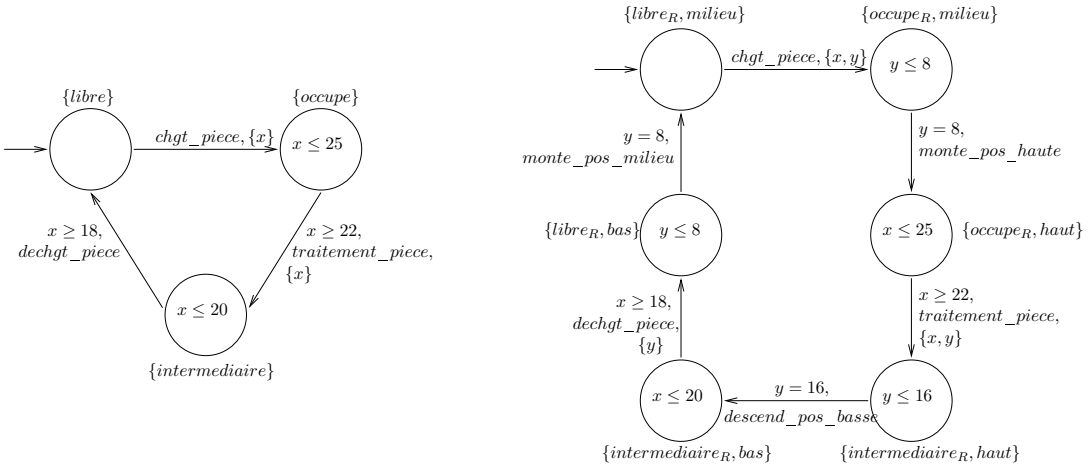
EXEMPLE 4.2. Nous considérons à présent un nouvel exemple nous permettant d'illustrer cette notion de raffinement. Il traite du comportement simplifié d'une presse, qui charge des pièces pour les traiter puis les évacue. La presse peut avoir trois positions : elle est en position *milieu* pour recevoir une pièce, en position *haute* pour traiter la pièce et en position *basse* pour l'évacuer.

Dans un premier temps, nous ne considérons que les actions de la presse qui concernent la pièce : chargement, traitement, et déchargement. La presse peut donc être dans trois états, représentant son "activité" : soit libre, quand elle ne contient pas de pièce, soit occupée, quand elle traite une pièce, soit dans une position dite intermédiaire, dénotant le fait qu'elle contient toujours une pièce mais qu'elle a terminé de la traiter. Une seule horloge, x, est utilisée pour modéliser les délais entre les différents états de la presse ainsi que leur durée. Nous obtenons ainsi un premier automate temporisé, présenté par la Fig. 4.5(a).
Raffiner cet automate consiste alors à introduire les mouvements verticaux : montée depuis la position centrale jusqu'en position haute pour traiter la pièce, puis descente en position basse pour l'évacuer et enfin remontée en position centrale pour accueillir une autre pièce. Les propositions atomiques abstraites sont renommées : $libre_R, occupe_R, intermediaire_R$ et de nouvelles sont introduites pour modéliser la position de la presse : haut, milieu ou bas. Une nouvelle horloge, y, est également utilisée pour modéliser les durées des déplacements verticaux. L'automate raffiné obtenu est présenté Fig. 4.5(b).

Le prédicat de collage entre ces deux automates consiste à lier les propositions atomiques raffinées et abstraites portant sur l'activité de la presse. Ce prédicat est le suivant :

$$libre_R \Rightarrow libre \wedge occupe_R \Rightarrow occupe \wedge intermediaire_R \Rightarrow intermediaire.$$

Il signifie qu'un état raffiné décoré par $libre_R$ (respectivement $occupe_R$ et $intermediaire_R$) devra être lié à un état abstrait décoré par $libre$ (respectivement $occupe$ et $intermediaire$).



(a) (b)dans le cas non temporisé

FIG. 4.5 – Automates temporisés abstrait (a) et raffiné (b) de la presse

4.3.3 Principe du raffinement pour les systèmes à composants.

Dans le cadre des modèles à base de composants, le raffinement consiste à créer un modèle abstrait de chaque composant, puis à raffiner chacun de ces modèles pour obtenir un modèle plus concret de chaque composant. Le but est alors que la relation de raffinement soit préservée par la composition. En d'autres termes, si chaque composant $C'_1, C'_2, \cdots, C'_n$ raffine respectivement les composants $C_1, C_2, \cdots, C_n$ alors le modèle raffiné complet $C'_1||C'_2||\cdots||C'_n$ doit raffiner le modèle abstrait complet $C_1||C_2||\cdots||C_n$ (l'opérateur || représente toujours ici un opérateur de composition quelconque). Ainsi, la relation de raffinement ne doit être vérifiée qu'entre chaque composant abstrait et raffiné. Cette propriété est appelée la *compositionnalité du raffinement* et permet de ne jamais avoir à mémoriser le système concret (raffiné) complet. La Fig. 4.6 illustre ce point de vue.

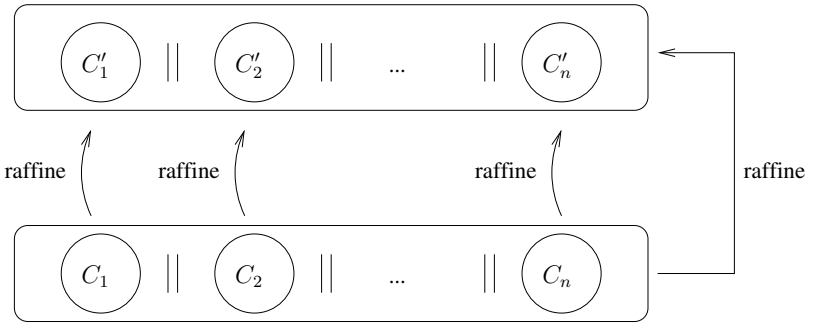


FIG. 4.6 – Compositionnalité du raffinement

4.3.4 Raffinement et vérification de propriétés

En termes de vérification de propriétés, l'objectif est de mener la vérification des propriétés globales du système sur un modèle de plus petite taille : le modèle complet abstrait, composé de tous les composants abstraits. Le processus de raffinement doit ensuite permettre d'assurer que les propriétés sont préservées sur le modèle concret. Le raffinement peut également être utilisé pour assurer la préservation des propriétés locales des composants sur leur version raffinée.

Le raffinement dans le cadre des systèmes à composants synchronisés (non temporisés) a été étudié notamment dans [KL03, KL04, Lan05]. Deux approches complémentaires ont été proposées pour la vérification compositionnelle du raffinement et des propriétés. La relation de raffinement entre composants est la relation de τ-simulation entre systèmes de transitions présentée dans la section 4.3.1.

4.3.5 L'intégration de composants, un cas particulier de raffinement ?

L'intégration de composants peut être vue comme un cas particulier de raffinement. En effet, en intégrant un composant C_2 à un composant C_1, l'automate obtenu $C_1||C_2$ contiendra de nouveaux décors d'états et horloges (celles de C_2) et de nouvelles actions (celles de C_2 non synchronisées avec des actions de C_1).
Quel que soit l'opérateur de composition, les états de $C_1||C_2$ sont en général composés d'un état de C_1 et d'un état de C_2, décoré par les deux décors des états desquels il est issu. Ainsi, le prédicat de collage consiste dans ce cas à lier un état (q_1, q_2) de $C_1||C_2$ à l'état q_1 de C_1. Il est donc simplement défini par `true`, c'est-à-dire l'identité des états du point de vue de C_1.

4.4 Conclusion

Nous avons présenté dans ce chapitre les deux opérateurs de composition pour les automates temporisés que nous considérerons tout au long de ce document. Le premier, que l'on note ||, adopte le paradigme de composition de CSP. Le second, noté |, se rapproche du paradigme de CCS, en adoptant le concept de priorités entre actions pour garantir le progrès maximal. Pour ces raisons, nous avons qualifié de *composition à la CSP* le premier type de composition, et la seconde *composition à la CCS avec priorités*. Nous utiliserons ces dénominations tout au long du document.

Puis, nous avons introduit en détail deux méthodes de développement incrémental qui peuvent être utilisées pour les modèles à base de composants : l'intégration de composants et le raffinement. Ces deux méthodes consistent à considérer un modèle abstrait du système, puis à le détailler en considérant de nouveaux décors, actions et horloges. Le cas de l'intégration de composants est un cas particulier de raffinement où les nouveaux éléments sont ceux du composant ajouté. Un prédicat de collage est utilisé pour donner le lien entre les états de l'automate obtenu par intégration ou par raffinement, à ceux de l'automate abstrait. Chacune de ces méthodes a pour objectif (i) de simplifier la phase de modélisation d'un système, mais surtout (ii) de rendre la vérification de propriétés plus applicable, dans le sens où elle sera menée sur les modèles abstraits, de taille plus petite que celle des modèles complets qui possèdent souvent en pratique un nombre d'états trop important pour la vérification.

Cependant, pour que ces méthodes soient intéressantes en pratique, il faut que les propriétés établies à un certain niveau d'abstraction le restent sur le modèle détaillé. En d'autres termes, ces propriétés doivent être préservées par le raffinement et par la composition, dans le cadre de l'intégration de composants.

La partie suivante est dédiée à l'étude de cette question et présente les premières contributions de cette thèse. Dans le chapitre 5, nous définissons des relations entre automates

temporisés qui garantissent la préservation de propriétés (en particulier des propriétés de vivacité). Puis, dans le chapitre 6, nous montrons l'intérêt de ces relations pour les méthodes de développement incrémental.

Deuxième partie

Contributions : modélisation incrémentale et préservation de propriétés

5

Des relations de simulation pour la préservation de propriétés

Sommaire

Dans le chapitre 4, nous avons présenté deux méthodes incrémentales pour modéliser des systèmes à base de composants. Ces méthodes ont pour but de faciliter la conception du système et la vérification des propriétés du modèle :

- en vérifiant les propriétés locales des composants uniquement sur les composants concernés dans le cadre de l'intégration de composants,
- en vérifiant les propriétés du modèle à un niveau d'abstraction plus élevé dans le cadre du raffinement.

Dans les deux cas, il est nécessaire que les propriétés vérifiées de cette manière soient préservées par l'intégration de composants et par le raffinement.

Revenons à l'exemple du passage à niveau, et aux propriétés P_2 et P_5 portant sur ce système. Rappelons que P_2 exprime le fait que *la barrière n'est jamais ouverte quand le train est sur le passage à niveau.* Cela revient à dire que *la barrière n'est jamais ouverte entre le moment où le contrôleur a commandé son abaissement et celui où il reçoit un signal de sortie du train* (puisqu'un train peut être sur le passage à niveau entre le moment où il signale son approche et celui où il signale sa sortie). La propriété P_5 exprime que *si le contrôleur reçoit un signal d'approche du train, alors la barrière sera fermée au maximum dans les deux unités de temps qui suivent.* Ces deux propriétés sont exprimées en MITL de la manière suivante :

(P_2) $\Box_{\geq 0}(c_2 \Rightarrow \neg is\_up)$

(P_5) $\Box_{\geq 0}(is\_up \wedge c_1 \Rightarrow \Diamond_{[0,2[} is\_down)$

Elles ne portent donc que sur les composants *barrière* et *contrôleur*. Lors d'une vérification classique, elles seraient vérifiées sur le modèle complet du système, composé de la barrière et du contrôleur, mais également de plusieurs trains. Or, comme elles ne portent que sur les deux premiers composants, il est intéressant de ne les vérifier que sur ces composants, dont la composition est de taille plus petite que le modèle complet, et donc, où le model-checking est plus facilement applicable. Le modèle complet sera ensuite obtenu par intégration du (ou des) composant(s) train. Il est alors nécessaire de garantir que les propriétés sont préservées sur le modèle complet.

Pour s'assurer de cette préservation, une solution communément admise est de comparer les comportements des deux systèmes, c'est-à-dire celui sur lequel les propriétés ont été vérifiées, et celui sur lequel elles doivent être préservées. Cette comparaison doit être effectuée selon certains critères, définis suivant le type de propriétés à préserver.
Dans la littérature, on trouve un certain nombre de relations d'équivalence ou de préordres permettant de comparer deux systèmes de transitions, les relations d'équivalence permettant de tester "l'égalité" entre les deux systèmes modulo cette relation, tandis que les préordres constituent plus généralement une relation d'implémentation entre deux systèmes. Dans [Gla90], R.J. Van Glabbeck présente douze relations d'équivalence pour les systèmes de transitions (également appelées *équivalences comportementales*) et les ordonne selon une hiérarchie de temps linéaire-arborescent, c'est à dire selon leur capacité à distinguer ou non la structure de branchement des exécutions des deux systèmes comparés. La Fig. 5.1, issue de [Gla90] présente cette hiérarchie. Les flèches dans la figure signifient que l'équivalence au départ de la flèche est plus fine que l'équivalence cible. En particulier, l'équivalence de bisimulation est celle qui différencie le plus de systèmes, tandis que l'équivalence de traces est celle qui en différencie le moins.

Dans [Gla93], R.J. Van Glabbeck reconsidère ces relations d'équivalences en tenant compte de l'activité interne des sytèmes à comparer. Cette activité interne est signalée par la présence d'actions non observables nommées τ. De nouveaux critères de comparaison sont introduits liés à la présence de ces actions. Parmi ceux-ci, on peut citer notamment la sensibilité à la divergence. Une relation est sensible à la divergence (*divergence-sensitive*)

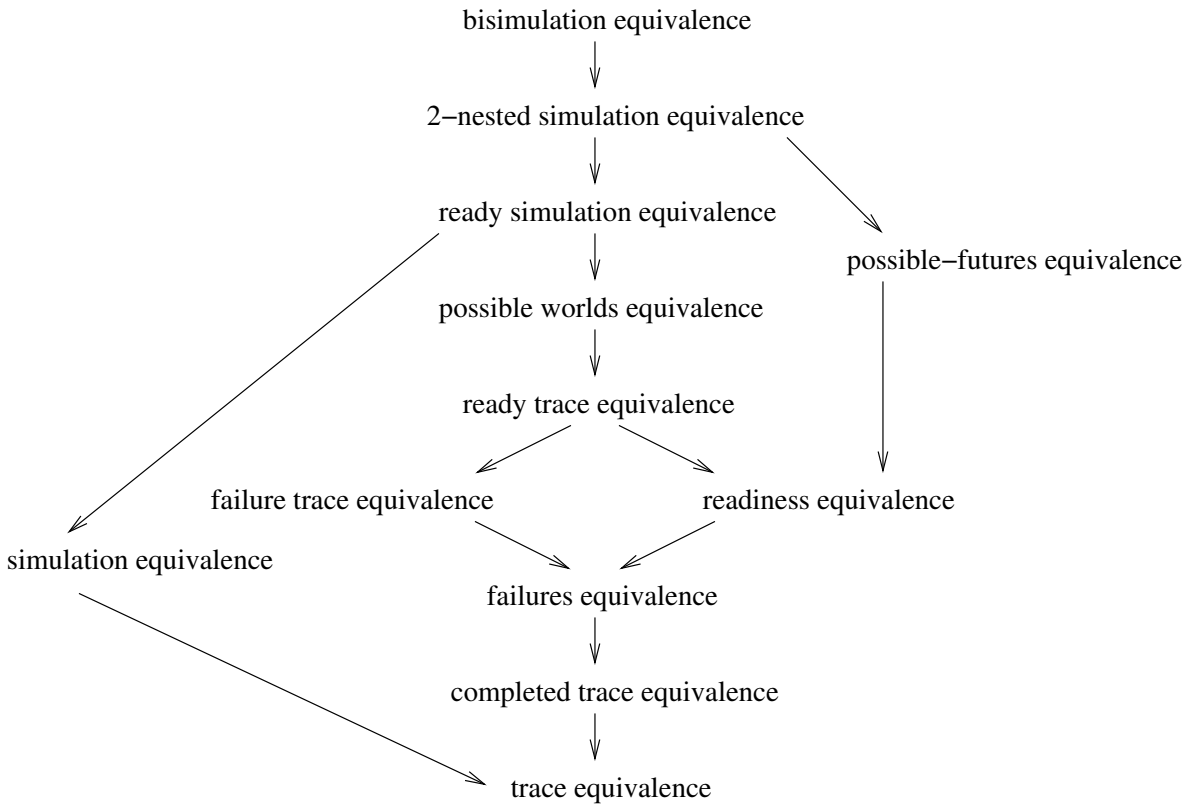


FIG. 5.1 – Hiérarchie des équivalences comportementales

si elle différencie deux systèmes dont l'un possède des séquences infinies d'activité interne et l'autre non.

Un préordre est associé à chacune des équivalences présentées dans [Gla90, Gla93]. Nous nous intéressons à ces préordres, notamment à celui de τ-simulation (c'est-à-dire le préordre de simulation prenant en compte les actions non observables τ des systèmes). Dans le cas non temporisé, la τ-simulation a été utilisée pour la modélisation incrémentale pour garantir la préservation de propriétés. On peut citer notamment le cas du raffinement des systèmes de transitions [BJK00] que nous avons évoqué au chapitre précédent, où une relation de τ-simulation est également utilisée pour définir ce raffinement et préserve les propriétés LTL.

Dans ce chapitre, nous définissons deux relations de τ-simulation pour les automates temporisés. Tout d'abord, une relation, appelée *τ-simulation temporisée*, assurant la préservation des propriétés de sûreté (section 5.1). Puis, une seconde relation, appelée *τ-simulation temporisée sensible à la divergence et respectant la stabilité*, préservant toutes les propriétés exprimées à l'aide de la logique MITL (section 5.2). Dans la section 5.3, nous étudions en détail les propriétés et formalismes qui sont préservés ou ne sont pas préservés par cette deuxième relation. Enfin, la section 5.4 conclut ce chapitre en effectuant une synthèse des travaux qui y sont présentés, et en passant en revue d'autres équivalences et

préordres définis dans la littérature pour les systèmes temporisés.

5.1 La τ-simulation temporisée

Nous définissons tout d'abord une première relation, la τ-simulation temporisée, préservant les propriétés de sûreté. Rappelons également que nous n'incluons pas les propriétés d'absence de blocage dans les propriétés de sûreté.

5.1.1 Présentation informelle

Considérons deux automates temporisés $A_1 = \langle Q_1, q_{0_1}, Labels_1, X_1, T_1, \texttt{Invar}_1, L_1 \rangle$ et $A_2 = \langle Q_2, q_{0_2}, Labels_2, X_2, T_2, \texttt{Invar}_2, L_2 \rangle$, tels que A_2 est obtenu à partir de A_1 soit par intégration de composants, soit par raffinement. Les actions apparaissant dans A_2 et n'existant pas dans A_1 sont considérées comme non observables, et renommées par τ. Les actions de A_2 apparaissant dans A_1 sont appelées actions observables de A_1, i.e. $Labels_2 = Labels_1 \cup \{\tau\}$. Ainsi, l'ensemble d'actions de A_2 est composé par les actions de A_1 et τ. La relation de τ-simulation temporisée est définie comme la relation de τ-simulation de [Gla93], prenant en compte les aspects temporisés de A_2 et de A_1, en imposant que les actions observables de A_2 se déclenchent au plus tard au même instant qu'elles ne le faisaient dans A_1. Informellement, la τ-simulation temporisée respecte donc les deux points suivants :

(i) Si A_2 peut effectuer une action observable après un certain laps de temps, alors A_1 pouvait effectuer la même action après le même laps de temps (ce qui implique que les actions observables ne peuvent pas se déclencher plus tard dans A_2 qu'elles ne le faisaient dans A_1),

(ii) Les actions non observables bégaient.

On impose également que la relation respecte une relation de base $\mathcal{B}$ entre les états de A_2 et de A_1. Cette relation est obtenue à partir d'un prédicat de collage de la manière suivante : un couple d'états de A_2 et A_1 appartient à $\mathcal{B}$ si ce couple satisfait le prédicat de collage. Ce prédicat était quant à lui défini à un niveau syntaxique sur les propositions atomiques de A_2 et A_1 (voir les sections 4.3.1 et 4.3.2) :

(iii) Les états en relation doivent appartenir à la relation de base $\mathcal{B}$.

La relation doit également respecter les valuations des horloges communes :

(iv) Les valuations des horloges communes de deux états en relation doivent être les mêmes.

5.1.2 Formalisation

La satisfaction abordée dans la description du point (iii) se définit formellement de la manière suivante. Soit q_2 un état de A_2, q_1 un état de A_1, et P_c un prédicat de collage entre A_2 et A_1. On dit que (q_2, q_1) satisfait le prédicat de collage P_c, noté $(q_2, q_1) \models_c P_c$,

si :

$$\bigwedge_{ap_2 \in L_2(q_2)} ap_2 \wedge P_c \Rightarrow \bigwedge_{ap_1 \in L_1(q_1)} ap_1.$$

Définissons à présent formellement la τ-simulation temporisée. Cette relation, notée $\mathcal{S}$, est définie sur les configurations de A_2 et A_1 par les quatre points indiqués précédemment. Le point (i) est représenté par les clauses 1 et 2 de la définition 8, le point (ii) par la clause 3 de la définition, le point (iii) par la clause 4, et enfin le point (iv) par la clause 5. Les clauses 1, 2 et 3 sont illustrées dans la figure 5.2, où s_1 et s'_1 sont des configurations de A_1, et s_2 et s'_2 des configurations de A_2, a est une action observable, τ est une action non observable et t un délai de temps.

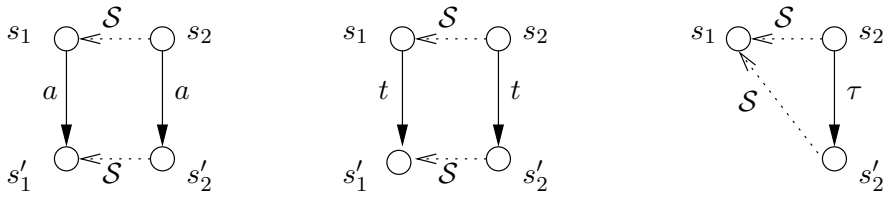


FIG. 5.2 – Les clauses 1, 2 et 3 de la τ-simulation temporisée

DÉFINITION 8 (τ-SIMULATION TEMPORISÉE) *Soient* $A_1 = \langle \mathrm{Q}_1, \mathrm{q}_{0_1}, \mathrm{Labels}_1, \mathrm{X}_1, \mathrm{T}_1, \mathtt{Invar}_1, \mathrm{L}_1 \rangle$ *et* $A_2 = \langle \mathrm{Q}_2, \mathrm{q}_{0_2}, \mathrm{Labels}_1 \cup \{\tau\}, \mathrm{X}_2, \mathrm{T}_2, \mathtt{Invar}_2, \mathrm{L}_2 \rangle$ *deux automates temporisés tels que* $X_1 \subseteq X_2$. *On appelle* S_1 *et* S_2 *les ensembles de configurations respectifs de* A_1 *et* A_2 *et* $\mathcal{B}$ *la relation dite de base entre* A_2 *et* A_1 *(définie à partir d'un prédicat de collage entre* A_2 *et* A_1*). La relation de* τ*-simulation temporisée* $\mathcal{S}$ *est la plus grande relation incluse dans* $S_2 \times S_1$. *On a* $(q_2, v_2)\mathcal{S}(q_1, v_1)$ *si les clauses suivantes sont vérifiées :*

1. *Simulation stricte :*
   $(q_2, v_2) \xrightarrow{e_2} (q'_2, v'_2) \wedge \mathtt{label}(e_2) \in Labels_1 \Rightarrow$
   $\exists (q'_1, v'_1) \cdot ((q_1, v_1) \xrightarrow{e_1} (q'_1, v'_1) \wedge \mathtt{label}(e_1) = \mathtt{label}(e_2) \wedge (q'_2, v'_2)\ \mathcal{S}\ (q'_1, v'_1))$.
2. *Egalité des délais :*
   $(q_2, v_2) \xrightarrow{t} (q_2, v'_2) \Rightarrow \exists (q_1, v'_1) \cdot ((q_1, v_1) \xrightarrow{t} (q_1, v'_1) \wedge (q_2, v'_2)\ \mathcal{S}\ (q_1, v'_1))$.
3. *Bégaiement des* τ*-transitions :*
   $(q_2, v_2) \xrightarrow{e_2} (q'_2, v'_2) \wedge \mathtt{label}(e_2) = \tau \Rightarrow (q'_2, v'_2)\ \mathcal{S}\ (q_1, v_1)$.
4. *Respect du décor des états :*
   $(q_2, q_1) \in \mathcal{B}$[11].
5. *Egalité des valuations des horloges communes :*
   $v_2\rfloor_{X_1} = v_1$.

[11] Rappelons que, comme nous l'avons indiqué informellement auparavant, $(q_2, q_1) \in \mathcal{B}$ ssi $(q_2, q_1) \models_c P_c$ où P_c est un prédicat de collage donné entre A_2 et A_1.

Nous considérons la plus grande relation dans $S_2 \times S_1$ satisfaisant toutes ces clauses. Notons qu'une telle relation existe. En effet, prenons deux relations quelconques R_1 et R_2, chacune sur $S_2 \times S_1$, satisfaisant les clauses de la définition 8. On a trivialement que l'union de ces deux relations satisfait également les mêmes clauses. La relation $\mathcal{S}$ peut donc être vue comme l'union de toutes les relations sur $S_2 \times S_1$ et satisfaisant ces clauses, et les contient donc toutes. C'est donc la plus grande relation sur $S_2 \times S_1$ satisfaisant ces clauses.

REMARQUE 5.1. La clause *respect du décor des états* a été définie à partir d'une relation de base $\mathcal{B}$ entre les ensembles d'états Q_2 et Q_1 des automates A_2 et A_1. Cette relation est elle-même définie à partir d'un prédicat de collage fourni entre A_2 et A_1. L'utilisation d'une telle relation permet de donner une définition exclusivement "sémantique" de la τ-simulation temporisée. Par la suite, et pour plus de lisibilité, notamment dans les preuves, nous utiliserons directement la définition de la relation $\mathcal{B}$ pour la clause *respect du décor des états*, c'est-à-dire $(q_2, q_1) \models_c P_c$, où P_c représentera le prédicat de collage entre A_2 et A_1.

Nous étendons à présent cette notion de τ-simulation temporisée aux automates temporisés : on dit qu'un automate temporisé A_1 τ-simule un automate temporisé A_2 si leurs configurations initiales sont en relation w.r.t. $\mathcal{S}$.

DÉFINITION 9 (τ-SIMULATION TEMPORISÉE ÉTENDUE AUX AUTOMATES TEMPORISÉS) *Soient A_1 et A_2 deux automates temporisés, dont les configurations initiales sont respectivement s_{0_1} et s_{0_2}. On dit que A_1 τ-simule A_2, noté $A_2 \preceq_{\mathcal{S}} A_1$ si $s_{0_2} \mathcal{S} s_{0_1}$.*

REMARQUE 5.2. Les conditions suivantes sont des conditions syntaxiques nécessaires sur les automates temporisés A_1 et A_2 pour une vérification réussie de la τ-simulation temporisée :

- Considérons une action observable de A_2. Par définition des actions observables, cette action existe également dans A_1. Soit une action observable nommée e_2 dans A_2 et e_1 dans A_1. L'action e_2 doit remettre à zéro les mêmes horloges de A_1 que e_1, c'est-à-dire $\texttt{reset}(e_2) \cap X_1 = \texttt{reset}(e_1)$, où X_1 est l'ensemble d'horloges de A_1.
- Les actions non observables de A_2 ne peuvent remettre à zéro que des horloges de A_2 qui n'existent pas dans A_1, c'est-à-dire $\texttt{reset}(e_2) \cap X_1 = \emptyset$. De plus, leurs gardes ne peuvent porter que sur ces horloges.

5.2 La τ-simulation temporisée sensible à la divergence et respectant la stabilité

La relation $\mathcal{S}$ définie dans la section précédente préserve uniquement les propriétés de sûreté. Nous présentons donc à présent une nouvelle relation, appelée *τ-simulation temporisée sensible à la divergence et respectant la stabilité*, et prouvons qu'elle préserve toutes les propriétés linéaires pouvant être exprimées en MITL. En particulier, les propriétés de vivacité et de réponse bornée sont préservées par cette relation.

5.2.1 Intérêt de la sensibilité à la divergence et du respect de la stabilité

La relation précédente entre deux automates A_2 et A_1 garantit que les séquences d'actions observables de A_2, éventuellement entrecoupées par des actions τ, sont des séquences d'actions qui existaient dans A_1, et que chacune de ces actions intervient au maximum après le même délai que dans A_1.
Considérons la propriété de vivacité P_5 portant sur le passage à niveau, et plus particulièrement sur les composants barrière et contrôleur. Cette propriété exprime le fait que *Si le contrôleur reçoit un signal d'approche du train, alors la barrière sera fermée au maximum dans les deux unités de temps qui suivent.* Elle est exprimée en MITL par la formule suivante : $\Box_{\geq 0}(is\_up \wedge c_1 \Rightarrow \Diamond_{[0,2[} is\_down)$.

Intuitivement, pour que cette propriété soit préservée lorsque l'on ajoute les composants *train*, il faut que les exécutions du modèle complet ne soient pas "coupées" entre le moment où $c_1 \wedge is\_up$ est vrai, et le moment où is_down est rencontré. Or, ce modèle complet peut contenir des actions non observables τ, pouvant s'insérer entre les actions observables. Les séquences d'actions observables peuvent donc être coupées, soit par l'introduction d'un blocage lors de l'intégration des composants *train*, soit par l'introduction d'une séquence infinie d'actions non observables.
Ainsi, pour que l'intégration des composants *train* préserve cette propriété, elle ne doit pas introduire de blocages, ni introduire de séquences infinies d'actions non observables. Dans [Gla93], ces deux critères sont appelés respectivement *respect de la stabilité* (*stability-respecting*) et *sensibilité à la divergence* (*divergence-sensitive*). La relation de τ-simulation temporisée, enrichie de ces deux critères, préserve ainsi toutes les propriétés linéaires MITL.

5.2.2 Formalisation

Nous définissons à présent formellement cette nouvelle relation de τ-simulation temporisée, enrichie par deux clauses exprimant les deux critères présentés précédemment : le respect de la stabilité et la sensibilité à la divergence. On considère toujours deux automates temporisés $A_1 = \langle Q_1, q_{0_1}, Labels_1, X_1, T_1, \texttt{Invar}_1, L_1\rangle$ et $A_2 = \langle Q_2, q_{0_2}, Labels_1 \cup \{\tau\}, X_2, T_2, \texttt{Invar}_2, L_2\rangle$, tels que A_2 soit obtenu à partir de A_1 soit par raffinement, soit par intégration de composants.

Sensibilité à la divergence. Détecter les séquences infinies d'actions non observables consiste à détecter les τ-cycles non-zénon dans A_2. Une exécution de A_2 contient un τ-cycle si, à partir d'un point, elle ne passe plus que par des transitions étiquetées par τ ou des transitions de temps. On dit que A_2 ne contient pas de τ-cycles non-zénon s'il ne contient pas de telles exécutions qui sont également non-zénon, c'est-à-dire :

$$\forall \rho, k \cdot (\rho \in \Gamma(A_2) \wedge \texttt{time}(\rho) = \infty \wedge k \geq 0 \Rightarrow$$

$$\exists k', e \cdot (k' \geq k \wedge (\rho, k') \xrightarrow{e} (\rho, k'+1) \wedge \texttt{label}(e) \neq \tau)).$$

Respect de la stabilité. Cette clause exprime que A_2 ne doit pas contenir de blocages qui n'existaient pas dans A_1. Formellement, si deux configurations s_2 et s_1 sont en relation w.r.t. $\mathcal{S}$ et que s_1 n'est pas un état de blocage, alors s_2 n'en est pas un non plus. Pour définir cette clause, nous utilisons le prédicat `free` défini dans le chapitre 2. Rappelons que, étant donné un état q d'un automate temporisé, $\texttt{free}(q)$ représente l'ensemble des valuations de q pour lesquelles le temps peut s'écouler et une transition discrète peut être prise ensuite.

Ces deux notions sont illustrées par la figure 5.3.

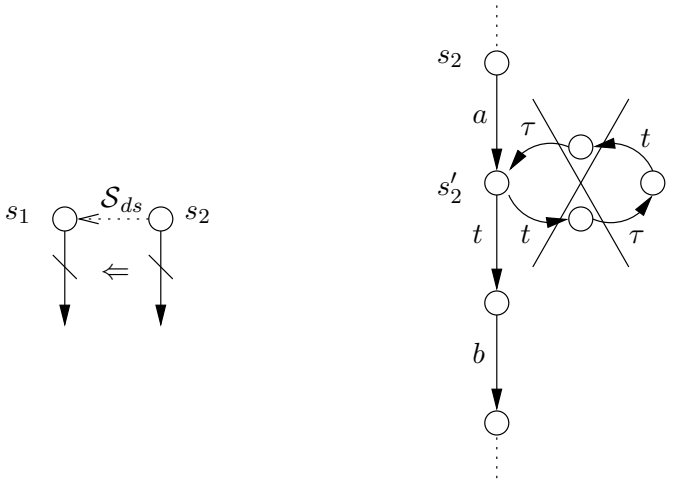


FIG. 5.3 – Respect de la stabilité et sensibilité à la divergence

Nous définissons à présent cette nouvelle relation, comme une τ-simulation temporisée, à laquelle sont ajoutées les deux clauses présentées : la sensibilité à la divergence et le respect de la stabilité. Nous appelons cette relation τ-simulation temporisée DS (pour sensible à la Divergence et respectant la Stabilité).

DÉFINITION 10 (τ-SIMULATION TEMPORISÉE DS) *Considérons deux automates temporisés* $A_1 = \langle \mathrm{Q}_1, \mathrm{q}_{0_1}, \mathrm{Labels}_1, \mathrm{X}_1, \mathrm{T}_1, \texttt{Invar}_1, \mathrm{L}_1 \rangle$ *et* $A_2 = \langle \mathrm{Q}_2, \mathrm{q}_{0_2}, \mathrm{Labels}_1 \cup \{\tau\}, \mathrm{X}_2, \mathrm{T}_2, \texttt{Invar}_2, \mathrm{L}_2 \rangle$*, tels que* $X_1 \subseteq X_2$*. S_1 et S_2 sont les ensembles de configurations respectifs de A_1 et A_2. La relation de τ-simulation temporisée DS, notée $\mathcal{S}_{ds}$, est la plus grande relation incluse dans $S_2 \times S_1$. On a $(q_2, v_2)\mathcal{S}_{ds}(q_1, v_1)$ si les clauses suivantes sont vérifiées :*

1. *τ-simulation temporisée : $(q_2, v_2)\mathcal{S}(q_1, v_1)$,*
2. *Sensibilité à la divergence : A_2 ne contient pas de τ-cycles non-zénon,*
3. *Respect de la stabilité : $v_2 \notin \texttt{free}(q_2) \Rightarrow v_1 \notin \texttt{free}(q_1)$.*

Comme précédemment, nous étendons cette relation aux automates temporisés.

DÉFINITION 11 (τ-SIMULATION TEMPORISÉE DS ÉTENDUE AUX AUTOMATES TEMPORISÉS) *Soient A_1 et A_2 deux automates temporisés, dont les configurations initiales sont s_{0_1} et s_{0_2}. On dit que A_1 DS-τ-simule A_2, noté $A_2 \preceq_{\mathcal{S}_{ds}} A_1$, si $s_{0_2}\mathcal{S}_{ds}s_{0_1}$.*

Dans la suite du document, nous parlerons également de simulation entre exécutions. On dira qu'une exécution ρ_1 DS-τ-simule une exécution ρ_2, noté $\rho_2 \preceq_{\mathcal{S}_{ds}} \rho_1$, si $(\rho_2, 0) \preceq_{\mathcal{S}_{ds}} (\rho_1, 0)$.

5.3 Préservation de propriétés

Nous avons défini la relation $\mathcal{S}_{ds}$ pour préserver un plus large éventail de propriétés. Nous prouvons à présent que $\mathcal{S}_{ds}$ préserve toutes les propriétés pouvant être exprimées à l'aide de la logique MITL, puis étudions le cas des automates de Büchi temporisés, de la propriété d'absence de blocage, du non-zénonisme et de l'atteignabilité.

5.3.1 Préservation de la MITL

Considérons deux automates temporisés A_1 et A_2, tels que $A_2 \preceq_{\mathcal{S}_{ds}} A_1$. Nous allons prouver que, pour chaque exécution de A_2, si l'exécution de A_1 qui la simule vérifie une propriété MITL φ, alors cette exécution de A_2 vérifie également φ. Ce résultat est énoncé dans le lemme 6 et dans le théorème 4.

Avant cela, rappelons que la satisfaction d'une formule MITL est définie sur des séquences d'états temporisées et sur leurs suffixes, et non pas sur des exécutions. Le résultat suivant sera nécessaire à la preuve du lemme 6.

LEMME 5 *Considérons deux exécutions ρ_1 et ρ_2 telles que $\rho_2 \preceq_{\mathcal{S}_{ds}} \rho_1$. Soient σ_1 et σ_2 les séquences d'états temporisées dans lesquelles sont inscrites respectivement ρ_1 et ρ_2 (c-à-d $\sigma_1 = \sigma(\rho_1)$ et $\sigma_2 = \sigma(\rho_2)$). Considérons également les suffixes σ_1^t et σ_2^t des SET σ_1 et σ_2 au temps t. On a alors :*

$$\forall \rho_2' \text{ inscrite dans } \sigma_2^t,\ \exists \rho_1' \text{ inscrite dans } \sigma_1^t \text{ tel que } \rho_2' \preceq_{\mathcal{S}_{ds}} \rho_1'.$$

PREUVE. Considérons que l'exécution ρ_2' est le suffixe de ρ_2 au temps t. On étudie deux cas :

1. Il existe une configuration (q_2, v_2) dans ρ_2 telle que $\texttt{time}(\rho_2, (q_2, v_2)) = t$. Comme $\rho_2 \preceq_{\mathcal{S}_{ds}} \rho_1$ alors il existe une configuration (q_1, v_1) dans ρ_1 telle que $(q_2, v_2)\mathcal{S}_{ds}(q_1, v_1)$. Par la clause 2 de la définition 8, on peut dire que le même temps s'écoule dans ρ_2 jusqu'à la configuration (q_2, v_2) que dans ρ_1 jusqu'à la configuration (q_1, v_1). Donc $\texttt{time}(\rho_1, (q_1, v_1)) = t$.
   Comme $\rho_2 \preceq_{\mathcal{S}_{ds}} \rho_1$, alors le suffixe ρ_2' de ρ_2 depuis la configuration (q_2, v_2) est en relation avec le suffixe ρ_1' de ρ_1 à partir de la configuration (q_1, v_1). On a donc $\rho_2' \preceq_{\mathcal{S}_{ds}} \rho_1'$.
2. Il n'existe pas de configuration (q_2, v_2) dans ρ_2 telle que $\texttt{time}(\rho_2, (q_2, v_2)) = t$. Cela signifie donc que cette valeur de temps arrive durant une transition de temps, c-à-d qu'il existe une transition de temps $(q, v) \xrightarrow{t'} (q', v')$ dans ρ_2 telle que $\texttt{time}(\rho_2, (q, v)) < t$ et $\texttt{time}(\rho_2, (q', v')) > t$. Il suffit alors de "couper" cette transition en deux nouvelles transitions de temps, où la configuration intermédiaire sera la configuration (q_2, v_2)

recherchée. Formellement, on crée deux transitions $(q,v) \xrightarrow{t'_1} (q_2,v_2) \xrightarrow{t'_2} (q',v')$, telles que $\texttt{time}(\rho_2,(q_2,v_2)) = t$.
Comme $\rho_2 \preceq_{\mathcal{S}_{ds}} \rho_1$, il n'existe pas non plus dans ρ_1 de configuration (q_1,v_1) telle que $\texttt{time}(\rho_1,(q_1,v_1)) = t$. Tout comme pour ρ_2, cette valeur de temps arrive durant une transition de temps. Comme précédemment, il est possible de "couper" cette transition de manière à créer la configuration (q_1,v_1) recherchée. Ainsi, on aura $(q_2,v_2)\mathcal{S}_{ds}(q_1,v_1)$ et comme $\rho_2 \preceq_{\mathcal{S}_{ds}} \rho_1$, les suffixes ρ'_2 de ρ_2 à partir de la configuration (q_2,v_2) et ρ'_1 de ρ_1 depuis la configuration (q_1,v_1) seront en relation, $\rho'_2 \preceq_{\mathcal{S}_{ds}} \rho'_1$.

Le lemme est donc vrai pour le cas particulier du suffixe de ρ_2 au temps t reconstruit à partir de σ_2^t. Sans perdre en généralité, on peut donc dire que le lemme est également vrai pour chaque exécution inscrite dans σ_2^t (car chacune de ces exécutions peut être réécrite par ρ'_2 en coupant ou en concaténant des transitions de temps).

□

Rappelons que le décor des états de A_2 n'est pas donné sur le même ensemble de propositions atomiques (appelé $Props_2$) que celui de A_1 (appelé $Props_1$). Un prédicat de collage est défini entre A_2 et A_1 pour établir une correspondance entre les propositions atomiques de $Props_2$ et celles de $Props_1$. Comme les formules MITL portant sur A_1 sont définies sur $Props_1$, elles ne pourront être satisfaites par préservation sur A_2 que modulo le prédicat de collage. On définit donc la satisfaction d'une formule MITL par préservation de la manière suivante.

DÉFINITION 12 (SATISFACTION D'UNE FORMULE MITL PAR PRÉSERVATION) *Soient* $A_2 = \langle \mathrm{Q}_2, \mathrm{q}_{0_2}, \mathrm{Labels}_1 \cup \{\tau\}, \mathrm{X}_2, \mathrm{T}_2, \texttt{Invar}_2, \mathrm{L}_2 \rangle$ *et* $A_1 = \langle \mathrm{Q}_1, \mathrm{q}_{0_1}, \mathrm{Labels}_1, \mathrm{X}_1, \mathrm{T}_1, \texttt{Invar}_1, \mathrm{L}_1 \rangle$ *deux automates temporisés définis respectivement sur des ensembles de propositions atomiques* $Props_2$ *et* $Props_1$*, et* P_c *leur prédicat de collage. Soit* σ *une séquence d'états temporisée obtenue à partir d'une exécution de* A_2*. Les formules* φ*,* ϕ *et* ψ *sont définies à partir de propositions atomiques de* $Props_1$ *et* $ap \in Props_1$*. On dit que* σ *satisfait* φ *par préservation, noté* $\sigma \models_p \varphi$ *si :*

- $\sigma \models_p \texttt{true}$ *est vrai,*
- $\sigma \models_p ap$ *ssi* $\bigwedge_{ap_2 \in L_2(\texttt{disc}((\sigma,0)))} ap_2 \wedge P_c \Rightarrow ap$*,*
- $\sigma \models_p \neg\varphi$ *ssi il n'est pas vrai que* $\sigma \models_p \varphi$*,*
- $\sigma \models_p \phi \vee \psi$ *ssi* $\sigma \models_p \phi$ *ou* $\sigma \models_p \psi$*,*
- $\sigma \models_p \phi \,\mathcal{U}_I\, \psi$ *ssi il existe* $t \in I$ *t.q.* $\sigma^t \models_p \psi$*, et* $\forall t' \in]0,t[$*,* $\sigma^{t'} \models_p \phi$*.*

Un automate temporisé A satisfait une propriété φ par préservation si toutes ses exécutions satisfont φ par préservation, c'est à dire :

$$\forall \rho \cdot (\rho \in \Gamma(A) \Rightarrow \sigma(\rho) \models_p \varphi).$$

Prouvons à présent que la relation $\mathcal{S}_{ds}$ préserve les propriétés MITL.

LEMME 6 (PRÉSERVATION D'UNE FORMULE MITL PAR $\mathcal{S}_{ds}$ SUR UNE EXÉCUTION) *Soient deux automates temporisés* A_1 *et* A_2 *tels que* $A_2 \preceq_{\mathcal{S}_{ds}} A_1$*,* ρ_1 *une exécution de* A_1 *et* ρ_2 *une exécution de* A_2*. On considère également une formule MITL* φ *portant sur* A_1*. On a :*

$$\rho_2 \preceq_{\mathcal{S}_{ds}} \rho_1 \wedge \sigma(\rho_1) \models \varphi \Rightarrow \sigma(\rho_2) \models_p \varphi.$$

PREUVE. On prouve (par induction sur la structure de la formule) que la relation $\mathcal{S}_{ds}$ préserve les propriétés exprimées en MITL. Pour plus de lisibilité, dans la suite de la preuve, on écrira σ_1 pour $\sigma(\rho_1)$ et σ_2 pour $\sigma(\rho_2)$.

1. Prouvons qu'une proposition atomique ap est préservée.
   Comme $\rho_2 \preceq_{\mathcal{S}_{ds}} \rho_1$, on sait que $(\rho_2, 0)\ \mathcal{S}_{ds}\ (\rho_1, 0)$. La relation $\mathcal{S}_{ds}$ respecte le décor des états donc :

   $$\bigwedge_{ap_2 \in L_2(\texttt{disc}((\rho_2,0)))} ap_2 \wedge P_c \Rightarrow \bigwedge_{ap_1 \in L_1(\texttt{disc}((\rho_1,0)))} ap_1.$$

   Comme $\sigma_1 \models ap$ alors

   $$\bigwedge_{ap_1 \in L_1(\texttt{disc}((\sigma_1,0)))} ap_1 \Rightarrow ap.$$

   σ_1 étant la séquence d'états temporisée obtenue à partir de ρ_1, leur états initiaux discrets sont les mêmes : $\texttt{disc}((\sigma_1, 0)) = \texttt{disc}((\rho_1, 0))$. Ainsi on a

   $$\bigwedge_{ap_1 \in L_1(\texttt{disc}((\rho_1,0)))} ap_1 \Rightarrow ap.$$

   Ainsi, par transitivité de l'implication on a :

   $$\bigwedge_{ap_2 \in L_2(\texttt{disc}((\rho_2,0)))} ap_2 \wedge P_c \Rightarrow ap.$$

   Comme σ_2 est la séquence d'états temporisée obtenue à partir de ρ_2, leur états initiaux discrets sont les mêmes : $L_2(\texttt{disc}((\sigma_2, 0))) = L_2(\texttt{disc}((\rho_2, 0)))$. On a donc

   $$\bigwedge_{ap_2 \in L_2(\texttt{disc}((\sigma_2,0)))} ap_2 \wedge P_c \Rightarrow ap.$$

   et donc $\sigma_2 \models_p ap$.

2. Prouvons qu'une formule du type $\neg\varphi$ est préservée.
   Par définition de la satisfaction par préservation d'une formule du type $\neg\varphi$ et l'hypothèse d'induction que φ est préservée.

3. Prouvons qu'une formule du type $\phi \vee \psi$ est préservée.
   Par définition de la satisfaction par préservation d'une formule du type $\phi \vee \psi$ et l'hypothèse d'induction que ϕ et ψ sont préservées.

4. Prouvons qu'une formule du type $\phi\, \mathcal{U}_I\, \psi$ est préservée.
   Comme $\sigma_1 \models \phi\ \mathcal{U}_I\ \psi$, il existe un suffixe σ_1^t au temps $t \in I$ tel que $\sigma_1^t \models \psi$. Considérons à présent le suffixe σ_2^t de σ_2 au temps t. Rappelons que σ_1 et σ_2 sont les

séquences d'états temporisés obtenues à partir de deux exécutions ρ_1 et ρ_2 telles que $\rho_2 \preceq_{\mathcal{S}_{ds}} \rho_1$. La relation $\mathcal{S}_{ds}$ interdit les séquences infinies d'actions non-observables dans ρ_2 (et dans σ_2), ce qui implique que toute action discrète apparaissant dans ρ_1 (et dans σ_1) apparaîtra également dans ρ_2 (et donc dans σ_2). De plus, comme cette relation $\mathcal{S}_{ds}$ n'autorise pas l'introduction de blocages dans ρ_2, alors si le temps t peut être atteint dans ρ_1 (et donc dans σ_1), alors ce temps t peut être atteint également dans ρ_2. Ainsi, si le suffixe σ_1^t existe, le suffixe σ_2^t existe également.
On veut maintenant prouver que σ_2^t satisfait ψ. On sait que σ_1^t satisfait ψ et que (par hypothèse d'induction) ψ est préservée. Il reste donc à montrer que chaque exécution inscrite dans σ_2^t est en relation w.r.t. $\preceq_{\mathcal{S}_{ds}}$ avec une exécution inscrite dans σ_1^t. Par le lemme 5, on sait que chaque exécution ρ_2' inscrite dans σ_2^t est en relation (w.r.t $\preceq_{\mathcal{S}_{ds}}$) avec une exécution ρ_1' inscrite dans σ_1^t. On a donc $\sigma_2^t \models \psi$.

Comme $\sigma_1 \models \phi \, \mathcal{U}_I \, \psi$, alors $\forall t' \in (0,t)$, $\sigma_1^{t'} \models \phi$. Et donc, de la même façon que précédemment, avec le lemme 5 et l'hypothèse d'induction que ϕ est préservée, $\forall t' \in (0,t)$, $\sigma_2^{t'} \models \phi$.

Ainsi, $\sigma_2 \models \phi \, \mathcal{U}_I \, \psi$.

□

THÉORÈME 4 (PRÉSERVATION DE PROPRIÉTÉS MITL PAR $\preceq_{\mathcal{S}_{ds}}$) *Soient φ une formule MITL, A_1 et A_2 deux automates temporisés. Si $A_1 \models \varphi$ et $A_2 \preceq_{\mathcal{S}_{ds}} A_1$ alors $A_2 \models_p \varphi$.*

PREUVE. La preuve est immédiate. Comme $A_2 \preceq_{\mathcal{S}_{ds}} A_1$, alors $\forall \rho_2 \cdot (\rho_2 \in \Gamma(A_2) \Rightarrow \exists \rho_1 \cdot (\rho_1 \in \Gamma(A_1) \wedge \rho_2 \preceq_{\mathcal{S}_{ds}} \rho_1))$. Comme $A_1 \models \varphi$, alors toutes ses exécutions satisfont cette propriété également, c-à-d $\forall \rho_1 \cdot (\rho_1 \in \Gamma(A_1) \Rightarrow \sigma(\rho_1) \models \varphi)$. Le lemme 6 permet de déduire que φ est également satisfaite, par préservation, pour toutes les exécutions de A_2, et donc que $A_2 \models_p \varphi$. □

5.3.2 Absence de blocage

La préservation de l'absence de blocage est préservée par la τ-simulation temporisée DS, par définition même de la relation (grâce à la clause *Respect de la stabilité* qui impose la non-introduction de blocage). Ainsi, si A_1 ne comporte pas de blocage et que A_1 simule A_2 par rapport à la τ-simulation temporisée DS, alors A_2 ne contient pas de blocage non plus.

PROPOSITION 1 *Si A_1 ne contient pas de blocage et $A_2 \preceq_{\mathcal{S}_{ds}} A_1$, alors A_2 ne contient pas de blocage.*

5.3.3 Non-zénonisme

Le non-zénonisme est une propriété essentielle des systèmes temporisés. Elle est également préservée par la τ-simulation temporisée DS.

PROPOSITION 2 *Si A_1 est non zénon et $A_2 \preceq_{\mathcal{S}_{ds}} A_1$, alors A_2 est non zénon*

Nous démontrons intuitivement cette proposition. Les clauses *Egalité des délais* et *Sensibilité à la divergence* permettent d'assurer cette préservation. En effet, toute exécution infinie ρ_2 de A_2 est simulée par une exécution infinie ρ_1 de A_1. La clause d'égalité des délais permet d'imposer que si ρ_2 peut laisser passer un certain délai de temps à un certain point, alors ρ_1 pouvait laisser passer le même délai au point correspondant. La clause de sensibilité à la divergence implique qu'il n'y a pas d'exécutions infinies ne contenant, à partir d'un point, que des actions non observables. Pour la préservation du non-zénonisme, la première clause implique donc que la somme des délais de ρ_2 est la même que la somme des délais de ρ_1, et que si cette somme converge dans ρ_2, alors elle convergeait également dans ρ_1. La seconde permet d'assurer qu'il ne peut pas y avoir une infinité d'actions non observables dans un délai de temps fini dans ρ_2, ce délai de temps correspondant à un délai de temps entre deux actions observables dans ρ_1. En effet, rappelons que dans ρ_2, les actions non observables sont en quelque sorte insérées entre les actions observables, mais que le délai entre deux actions observables ne doit pas être augmenté (et est fini). Ainsi, en interdisant les séquences infinies d'actions non observables successives, la clause de sensibilité à la divergence permet également de garantir qu'il n'y aura pas une infinité d'actions non observables dans un délai de temps fini entre deux actions observables. Ainsi, si A_1 ne possède pas d'exécutions non zénon et que $A_2 \preceq_{\mathcal{S}_{ds}} A_1$, alors A_2 n'en possède pas non plus.

5.3.4 Atteignabilité

Les propriétés principales qui ne peuvent pas être exprimées par un formalisme linéaire, en particulier MITL, sont les propriétés d'atteignabilité. Si l'on considère toujours deux automates temporisés A_1 et A_2, tels que A_1 (DS-)τ-simule A_2, l'atteignabilité des configurations de A_1 n'est pas préservée par ces τ-simulations temporisées. En effet, ces relations imposent uniquement que les exécutions de A_2 existent également dans A_1, modulo les actions non-observables τ. Il est donc possible que des exécutions de A_1 ne simulent aucune exécution de A_2, et ainsi que les états de ces exécutions ne soient plus atteignables dans A_2.

5.3.5 Automates de Büchi temporisés

Précédemment, nous avons montré que les propriétés linéaires, exprimées en MITL, sont préservées par la τ-simulation temporisée DS. En pratique, il est courant que les propriétés linéaires soient directement énoncées sous la forme d'un automate de Büchi temporisé. C'est pourquoi nous avons cherché à étudier si ces automates sont préservés par la τ-simulation temporisée DS.

Dans [HRS98], les auteurs ont appelé *langages ω-réguliers temporisés sans compteurs* la classe de langages temporisés équivalente à la MITL. Il suit donc que les automates temporisés reconnaissant exactement les formules MITL, reconnaissent également cette classe de langages. Si l'on ajoute aux automates la possibilité de "compter", on obtient

directement la classe entière des *langages ω-réguliers temporisés* (on notera d'ailleurs le parallèle avec le cas non temporisé [Eme90, Tho90]).

C'est justement cette faculté de compter qui n'est pas préservée par les τ-simulations temporisées, dû à l'introduction des actions non observables. Prenons un nouvel exemple simple pour illustrer ceci. Imaginons un système fabriquant deux sortes de pièces : des pièces bleues et des pièces rouges. Une propriété de ce système serait que, entre deux pièces rouges fabriquées, il y a un nombre pair de pièces bleues fabriquées consécutivement. L'automate de Büchi de la Fig. 5.4 représente cette propriété du système et n'est pas préservé par la τ-simulation temporisée DS[12]. En effet, l'introduction d'actions non observables peut empêcher qu'il y ait exactement un nombre pair de pièces bleues consécutives produites.

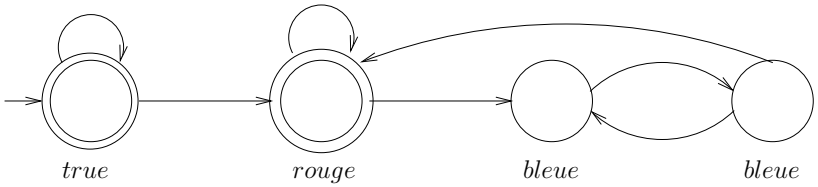


FIG. 5.4 – Automate de Büchi non préservé par la τ-simulation temporisée DS

5.4 Conclusion

Nous effectuons à présent une synthèse de ce chapitre, et présentons d'autres travaux portant sur les relations d'équivalence et les préordres pour les systèmes temporisés.

Synthèse

Nous avons défini dans ce chapitre deux relations de τ-simulation pour les automates temporisés. La première relation, appelée τ-simulation temporisée, préserve les propriétés de sûreté, et la seconde, appelée τ-simulation temporisée sensible à la divergence et respectant la stabilité, préserve toutes les propriétés linéaires pouvant être exprimées en MITL, ainsi que les propriétés d'absence de blocage et de non-zénonisme.

Cette seconde relation permet donc de préserver un large éventail de propriétés. Nous avons toutefois montré que les propriétés d'atteignabilité ne sont pas préservées, ainsi que le formalisme des automates de Büchi temporisés. Par rapport à cela, il est important de noter que, pour les propriétés d'atteignabilité, de nombreux travaux se sont attelés à rendre leur vérification efficace, et en pratique, il existe maintenant des algorithmes et des outils performants pour ce type de propriétés. En ce qui concerne les automates de

[12]Notons que nous avons pris ici un exemple non temporisé pour plus de simplicité, mais que le résultat est le même dans le cas temporisé : ce type d'automate, temporisé ou non, n'est pas préservé.

Büchi, même s'il existe quelques propriétés intéressantes qu'ils peuvent exprimer et qui ne peuvent pas être spécifiées en MITL, il semble que ces propriétés restent peu courantes en pratique.

L'objectif est d'utiliser ces relations dans le cadre de la modélisation incrémentale des systèmes temporisés à composants, afin de garantir que les propriétés vérifiées à une étape du processus soient préservées pendant le reste du processus. En particulier, nous souhaitons utiliser ces relations pour garantir que, lorsqu'on raffine un composant ou lorsqu'on l'intègre avec d'autres, ses propriétés restent valides après raffinement ou après intégration.

Dans le chapitre suivant, nous étudions donc les propriétés de ces relations par rapport aux deux opérateurs de composition définis dans la section 4.1.

D'autres travaux sur les relations d'équivalence et les préordres pour les systèmes temporisés

De nombreux travaux définissent des équivalences comportementales, ainsi que les préordres associés, dans le cadre des systèmes temporisés.

Concernant les équivalences, la *bisimulation temporisée* a été étudiée et sa décidabilité a été prouvée dans [Cer92b, Cer92a]. Des bisimulations faisant abstraction du temps ont été étudiées dans [TY96, Tri98, TY01] : la *ta*[13]*-bisimulation forte*, la *ta-bisimulation observationnelle* et la *ta-delay bisimulation*. Contrairement à la version temporisée, l'abstraction du temps permet d'abstraire les aspects quantitatifs du passage du temps. Ainsi, deux états sont en relation de bisimulation si, (i) quand le premier effectue une action l'autre peut l'effectuer aussi, et inversement et (ii) quand le premier peut laisser du temps s'écouler, le second le peut aussi, et inversement. Contrairement à la bisimulation temporisée qui, dans le point (ii), exige que les délais de temps qui s'écoulent depuis chaque état soient les mêmes, les bisimulations avec abstraction du temps considèrent uniquement que du temps doit s'écouler depuis chaque état, mais sans que ce soit nécessairement la même quantité. Concernant la préservation de propriétés, la *ta-bisimulation forte* préserve l'atteignabilité, les automates de Büchi temporisés, et la CTL (la TCTL est aussi préservée modulo une transformation de l'automate temporisé et de la formule telle que celle présentée dans [Tri98]). La *ta-bisimulation observationnelle* et la *ta-delay bisimulation* préservent l'atteignabilité et les automates de Büchi temporisés. Notons que l'équivalence des régions est une *ta-bisimulation forte*. Ces équivalences de bisimulation préservent donc un large spectre de propriétés. Malgré cela, elles ne sont pas vraiment adaptées à la modélisation incrémentale.

Les préordres, auxquels nous nous intéressons particulièrement dans ce document, y sont plus adaptés et ont également été largement étudiés. Le *préordre* et l'*équivalence de simulation avec abstraction du temps* ont été étudiés dans [HHK95], où un algorithme pour calculer l'équivalence est proposé. Cependant, les propriétés temporisées ne sont

[13] pour *time-abstracting*

pas préservées par ces relations. Le *préordre de simulation temporisée* a été défini dans [TAKB96, Tas98]. Les auteurs ont également montré que vérifier l'existence de cette simulation est un problème EXPTIME. En revanche, les actions non observables ne sont pas prises en compte dans la définition de cette simulation. La notion la plus proche des relations que nous avons présentées dans ce chapitre est la *ready* simulation temporisée de [JLS00], définie sur des automates temporisés *étendus* tels que ceux pris en compte par Uppaal. Ce sont des automates temporisés pouvant posséder des actions urgentes, et des variables partagées de type entier qui peuvent être modifiées par les transitions. Les actions non-observables sont également considérées. La définition de cette relation est équivalente à notre τ-simulation temporisée, et préserve donc également les propriétés de sûreté. Par contre, elle n'a pas été étendue pour prendre en compte les propriétés de vivacité.

Ainsi, même si ces relations de simulation semblent assez proches de celles que nous avons définies, il n'existe pas, à notre connaissance, de simulation temporisée définie prenant en compte les actions non observables τ, et préservant toutes les propriétés MITL, et en particulier les propriétés de vivacité, comme celles que nous avons présentées dans ce chapitre. Notons également dès maintenant que, lors de l'étude de la compatibilité de ces relations avec des opérateurs de composition (comme nous allons le faire dans le chapitre suivant), les actions non observables ne sont plus prises en compte.

6

Exploitation des simulations pour la modélisation incrémentale

Sommaire

Dans le chapitre précédent, nous avons présenté deux relations de simulation : une τ-simulation temporisée préservant les propriétés de sûreté, et une τ-simulation temporisée sensible à la divergence et respectant la stabilité préservant les propriétés MITL, d'absence de blocages et de non-zénonisme. Nous souhaitons exploiter ces relations lors de la modélisation incrémentale d'un système temporisé, et ainsi profiter de leurs propriétés concernant la préservation de propriétés.

Dans ce chapitre, nous nous focalisons sur les deux opérateurs de composition entre automates temporisés présentés dans la section 4.1. Trois propriétés sont essentielles pour dégager l'intérêt des simulations pour la modélisation incrémentale des systèmes à composants :

1. une propriété que nous avons nommée l'intégration, signifiant que tout automate temporisé simule sa composition avec un autre automate temporisé,
2. la compatibilité de la simulation avec l'opérateur de composition, signifiant que, pour trois automates temporisés A, B et C, si B simule A, alors la composition de B et C simule la composition de A et C,
3. la compositionnalité, signifiant que, pour des automates temporisés A, B, C et D, si B simule A et D simule C alors la composition de B et D simule la composition de A et C.

La propriété 1 est essentielle dans le cadre de l'intégration de composants pour garantir la composabilité, tandis que les propriétés 2 et 3 le sont dans le cadre du raffinement pour le vérifier de manière compositionnelle.

Dans la section 6.1, nous montrons que la τ-simulation temporisée possède ces propriétés vis-à-vis de l'opérateur de composition parallèle *à la CSP*, et donc qu'une modélisation incrémentale formalisée par cette relation et effectuée avec cet opérateur préserve automatiquement les propriétés de sûreté. Nous montrons également pourquoi la τ-simulation temporisée DS ne possède pas, quant à elle, ces propriétés vis-à-vis de cet opérateur. En revanche, dans la section 6.2, nous montrons que cette simulation les possède par rapport à l'opérateur de composition parallèle *à la CCS* avec priorités, sous certaines conditions. Nous récapitulons ces résultats dans la section 6.3 et résumons l'apport des τ-simulations pour la préservation de propriétés lors du processus de développement incrémental. Nous présentons également et effectuons une comparaison avec les résultats obtenus pour d'autres relations de simulation définies dans la littérature.

6.1 Systèmes à composants utilisant l'opérateur *à la CSP*

Nous commençons tout d'abord par étudier les propriétés des τ-simulations temporisées par rapport à l'opérateur de composition *à la CSP*, défini dans la section 4.1.1.

6.1.1 Propriétés de la τ-simulation temporisée

Nous étudions tout d'abord les propriétés de la τ-simulation temporisée. Les propositions suivantes montrent son intérêt lorsque l'opérateur de composition considéré est l'opérateur de composition parallèle ||. En effet, la proposition 3 énonce que tout automate temporisé simule tout système obtenu par l'intégration de cet automate dans un environnement, lorsque cette intégration est effectuée avec ||. La proposition 4 montre la compatibilité de la τ-simulation avec l'opérateur ||. Enfin, la proposition 5 garantit la propriété de compositionnalité.

Notons que ces résultats sont des résultats classiques mais nécessaires, étudiés dans le cadre de notre notion de simulation et des opérateurs de composition que nous considérons. Dans le cas non temporisé, on peut par exemple trouver dans [CGP99] des résultats similaires avec une composition définie comme un produit synchrone entre systèmes de transitions et une simulation *stricte* (ne prenant pas en compte les actions non-observables). Dans le cas temporisé, ce résultat est également obtenu dans [TAKB96, Tas98] pour des automates temporisés composés en utilisant également un produit synchrone et une simulation temporisée *stricte*.

PROPOSITION 3 *Soient A et B deux automates temporisés. On a : $A||B \preceq_{\mathcal{S}} A$.*

PREUVE. La preuve découle directement de la définition sémantique de $||$, donnée dans la section 4.1.1.
Soient S_{AB} et S_A les ensembles de configurations respectifs de $A||B$ et A. Soient (q_{0_A}, v_{0_A}) et (q_{0_B}, v_{0_B}) les configurations initiales respectives de A et B. Par construction de $A||B$, sa configuration initiale est donc le couple $((q_{0_A}, v_{0_A}), (q_{0_B}, v_{0_B}))$.
Pour prouver que $A||B \preceq_{\mathcal{S}} A$, il suffit de prouver que $((q_{0_A}, v_{0_A}), (q_{0_B}, v_{0_B}))\mathcal{S}(q_{0_A}, v_{0_A})$. Par définition, $\mathcal{S}_{ds}$ est la plus grande relation incluse dans $S_{AB} \times S_A$ satisfaisant les clauses de la définition 8. Donc, pour toute relation $R \subseteq S_{AB} \times S_A$, si R satisfait également ces clauses, on a $R \subseteq \mathcal{S}$. Considérons donc une relation $R \subseteq S_{AB} \times S_A$ telle que $\forall((q_A, v_A), (q_B, v_B)) \in S_{AB}$,

$$((q_A, v_A), (q_B, v_B))R(q'_A, v'_A) \text{ si } (q_A, v_A) = (q'_A, v'_A).$$

On va alors chercher à démontrer que R satisfait les clauses de la définition 8.

Soit $((q_A, v_A), (q_B, v_B)), (q_A, v_A) \in R$:

1. *Simulation stricte* : soit une transition $((q_A, v_A), (q_B, v_B)) \stackrel{g_2,a,r_2}{\rightarrow} ((q'_A, v'_A), (q'_B, v'_B))$ de $A||B$, telle que a est dans l'alphabet de A, c'est-à-dire $a \in Labels_A$. Par définition de $||$, dans $A||B$, cette transition peut être soit une transition de A ne se synchronisant avec aucune action de B, soit une action synchronisée avec une action de B portant la même étiquette. Dans les deux cas, si cette transition existe alors par construction de $A||B$, la transition $(q_A, v_A) \stackrel{g_1,a,r_1}{\rightarrow} (q'_A, v'_A)$ existe dans A. Par définition de R, on a bien $((q'_A, v'_A), (q'_B, v'_B))R(q'_A, v'_A)$ donc R satisfait la clause *simulation stricte*.

2. *Egalité des délais* : les arguments pour la preuve sont identiques au cas précédent, dû au fait qu'une transition de temps dans $A||B$, étiquetée par $t \in \mathbb{R}^+$ résulte de la "synchronisation" d'une transition de temps de A et d'une transition de temps de B, chacune étiquetée par le même délai t. La relation R satisfait donc la clause *égalité des délais*.

3. *Bégaiement des τ-transitions* : considérons une transition $((q_A, v_A), (q_B, v_B)) \stackrel{\tau}{\rightarrow} ((q'_A, v'_A), (q'_B, v'_B))$ de $A||B$. Cette transition est une transition entrelacée dans $A||B$,

issue de B. Par définition de $||$, cette transition ne modifie pas l'état de A. On a ainsi $(q'_A, v'_A) = (q_A, v_A)$ et, par définition de R, on a bien $((q_A, v_A), (q'_B, v'_B))R(q_A, v_A)$. Donc, R satisfait la clause *bégaiement des τ-transitions*.

4. *Respect du décor des états* : dans le cadre de l'intégration de composants, le prédicat de collage considéré est égal à *true*. En particulier, le prédicat de collage entre $A||B$ et A est *true* et on a directement $(((q_A, v_A), (q_B, v_B)), (q_A, v_A) \models_c \texttt{true}$ par la définition de $\models_c$. La relation R satisfait donc la clause *respect du décor des états*.

5. *Egalité des valuations des horloges communes* : cette clause est satisfaite par définition de R et grâce au fait que les ensembles d'horloges de A et B sont disjoints.

Ainsi, la relation R satisfait toutes les clauses de la définition 8. On a donc $R \subseteq \mathcal{S}$. Par définition de R, on a bien $((q_{0_A}, v_{0_A}), (q_{0_B}, v_{0_B}))R(q_{0_A}, v_{0_A})$ et comme $R \subseteq \mathcal{S}$, on a également $((q_{0_A}, v_{0_A}), (q_{0_B}, v_{0_B}))\mathcal{S}(q_{0_A}, v_{0_A})$.

□

PROPOSITION 4 *Soient A, B et C trois automates temporisés. Si $A \preceq_{\mathcal{S}} B$ alors $A||C \preceq_{\mathcal{S}} B||C$.*

PREUVE. On procède pour cette preuve de la même manière que pour la preuve précédente. Soit une relation $R \subseteq S_{AC} \times S_{BC}$ définie telle que $(s_A, s_C)R(s_B, s_C)$ si $s_A \mathcal{S} s_B$. Comme précédemment, on cherche à prouver que R satisfait les clauses de la définition 8. Soit $((s_A, s_C), (s_B, s_C)) \in R$, où $s_A = (q_A, v_A)$, $s_B = (q_B, v_B)$ et $s_C = (q_C, v_C)$. Les ensembles $Labels_A$, $Labels_B$ et $Labels_C$ représentent respectivement les alphabets de A, B et C.

1. *Simulation stricte* : cette clause concerne trois types de transitions dans $A||C$ (rappelons que les actions de A contiennent celles de B et que, pour $A \preceq_{\mathcal{S}} B$, elle concernait les actions de $Labels_A \cap Labels_B$) :
   (i) Les transitions de C qui ne se synchronisent pas avec une transition de A,
   (ii) Les transitions de $Labels_A \cap Labels_B$ qui ne se synchronisent pas avec une action de C,
   (iii) Les transitions de C qui se synchronisent avec une action de A (et qui apparaissent donc dans $B||C$ soit comme une action entrelacée de C si la synchronisation concerne une action de A non commune à B, soit comme une action de B synchronisée avec une action de C dans le cas contraire).
   Détaillons ces trois cas :
   (i) Considérons une transition $(s_A, s_C) \stackrel{g,c,r}{\rightarrow} (s_A, s'_C)$ telle que $c \in Labels_C \backslash Labels_A$. Par construction de $A||C$, la transition $s_C \stackrel{g,c,r}{\rightarrow} s'_C$ existe dans C. Notons que g ne concerne que les horloges de C et que $v_C \models g$. De ce fait, la transition $(s_B, s_C) \stackrel{g,c,r}{\rightarrow} (s_B, s'_C)$ existe dans $B||C$. Comme $s_A \mathcal{S} s_B$, et par définition de R, on a bien $(s_A, s'_C)R(s_B, s'_C)$.

(ii) Considérons une transition $(s_A, s_C) \stackrel{g,a,r}{\rightarrow} (s'_A, s_C)$ telle que $a \in (Labels_A \cap Labels_B) \backslash Labels_C$ (l'état s_C n'est donc pas modifié). Par définition de $||$, la transition $s_A \stackrel{g,a_r}{\rightarrow} s'_A$ existe dans A. Comme $s_A \mathcal{S} s_B$, on a également une transition $s_B \stackrel{g',a,r'}{\rightarrow} s'_B$ dans B, tel que $s'_A \mathcal{S} s'_B$. Ainsi, $v_B \models g'$. Comme g' ne porte que sur des horloges de B et que les ensembles d'horloges de B et C sont disjoints (par hypothèse de construction $B||C$), alors $(v_B, v_C) \models g'$ et donc la transition $(s_B, s_C) \stackrel{g',a,r'}{\rightarrow} (s'_B, s_C)$ existe dans $B||C$ et $(s'_A, s_C) R (s'_B, s_C)$ par définition de R

(iii) Considérons une transition $(s_A, s_C) \stackrel{g,a,r}{\rightarrow} (s'_A, s'_C)$ telle que $a \in Labels_A \cap Labels_C$. Par définition de $||$, il existe donc une transition $s_A \stackrel{g_1,a,r_1}{\rightarrow} s'_A$ dans A et une transition $s_C \stackrel{g_2,a,r_2}{\rightarrow} s'_C$ dans C. On envisage deux cas : soit $a \in A \cap B$ (a est une action observable de A par rapport à B et existait donc dans B), soit $a \in A \backslash B$ (a est une action non observable de A n'existant pas dans B).
Dans le premier cas, comme $s_A \mathcal{S} s_B$, il existe une transition $s_B \stackrel{g_3,a,r_3}{\rightarrow} s'_B$ dans B telle que $s'_A \mathcal{S} s'_B$. Il y a donc une transition $(s_B, s_C) \stackrel{g',a,r'}{\rightarrow} (s'_B, s'_C)$ dans $B||C$, telle que $(s'_A, s'_C) R (s'_B, s'_C)$ par définition de R.
Dans le second cas, comme $s_A \mathcal{S} s_B$, on a $s'_A \mathcal{S} s_B$. La transition $(s_B, s_C) \stackrel{g_2,a,r_2}{\rightarrow} (s_B, s'_C)$ existe dans $B||C$ car v_B ne concerne pas les horloges de C et $v_C \models g_2$. Par définition de R, on a de plus que $(s'_A, s'_C) R (s_B, s_C)$.

La relation R satisfait donc la clause *simulation stricte*.

2. *Egalité des délais* : soit une transition $(s_A, s_C) \stackrel{t}{\rightarrow} (s'_A, s'_C)$. Par définition de $||$, les transitions $s_A \stackrel{t}{\rightarrow} s'_A$ et $s_C \stackrel{t}{\rightarrow} s'_C$ existent respectivement dans A et C. Comme $s_A \mathcal{S} s_B$, alors la transition $s_B \stackrel{t}{\rightarrow} s'_B$ existe dans B et $s'_A \mathcal{S} s'_B$. On a donc également la transition $(s_B, s_C) \stackrel{t}{\rightarrow} (s'_B, s'_C)$ dans $A||C$ et, par définition de R, on a bien $(s'_A, s'_C) R (s'_B, s'_C)$. La relation R satisfait donc la clause *égalité des délais*.

3. *Bégaiment des τ-transitions* : dans $A||C$, par rapport à $B||C$, les τ-transitions sont celles étiquetées par une action de $A \backslash (B \cup C)$. Considérons donc une transition $(s_A, s_C) \stackrel{\tau}{\rightarrow} (s'_A, s_C)$ (l'état s_C n'est pas modifié car τ représente une action de $A \backslash (B \cup C)$). Comme $s_A \mathcal{S} s_B$, on a $s'_A \mathcal{S} s_B$. Il suit que $(s'_A, s_C) R (s_B, s_C)$ et que R satisfait la clause *bégaiement des τ-transitions*.

4. *Respect du décor des états* : immédiat du fait que $s_A \mathcal{S} s_B$.

5. *Egalité des valuations des horloges communes* : immédiat du fait que $s_A \mathcal{S} s_B$.

□

Les deux propositions précédents permettent de bénéficier de la propriété de compositionnalité.

PROPOSITION 5 (COMPOSITIONNALITÉ) *Soient A, B, C et D quatre automates temporisés. Si $A \preceq_{\mathcal{S}} B$ et $C \preceq_{\mathcal{S}} D$ alors $A||C \preceq_{\mathcal{S}} B||D$.*

PREUVE. Immédiate grâce à la proposition 4. Comme $A \preceq_{\mathcal{S}} B$, alors $A||C \preceq_{\mathcal{S}} B||C$. Comme $C \preceq_{\mathcal{S}} D$ alors $B||C \preceq_{\mathcal{S}} B||D$. Par transitivité de la relation $\preceq_{\mathcal{S}}$, il suit que $A||C \preceq_{\mathcal{S}} B||D$. □

Les propositions précédentes ne sont plus vraies pour la τ-simulation temporisée sensible à la divergence et respectant la stabilité. En effet, des τ-cycles non zénon et des blocages peuvent être introduits par la composition parallèle *à la CSP*, comme nous allons le montrer dans la section suivante.

6.1.2 Propriétés de la τ-simulation temporisée DS

Par rapport à la τ-simulation temporisée, rappelons que la τ-simulation temporisée DS possède deux clauses supplémentaires : la sensibilité à la divergence et le respect de la stabilité. Nous étudions à présent les conséquences de l'introduction de ces deux clauses pour les propriétés d'intégration, de compatibilité et de compositionnalité.
Nous procédons en deux temps. D'abord, nous étudions les conséquences de l'ajout de la sensibilité à la divergence, puis celles de l'ajout du respect de la stabilité.
Nous noterons dans cette section $\preceq_{\mathcal{S}_d}$ la τ-simulation temporisée à laquelle est rajoutée la clause de sensibilité à la divergence, et $\preceq_{\mathcal{S}_s}$ la τ-simulation temporisée avec le respect de la stabilité.

La sensibilité à la divergence

La relation $\preceq_{\mathcal{S}_d}$ ne possède pas la propriété d'intégration. En effet, considérons deux automates A et B, tels que les actions internes de B, appelées τ, sont considérées comme étant non observables dans $A||B$. La clause concernant la sensibilité à la divergence n'est pas automatiquement vérifiée lors de la composition parallèle de A et B. Un contre-exemple est présenté dans la figure 6.1, qui montre deux automates temporisés A et B dont la composition $A||B$ contient des τ-cycles non zénon. En effet, les exécutions de $A||B$ passant par la transition a, puis ne prenant plus que des transitions τ sont des exécutions non-zénon contenant un τ-cycle.

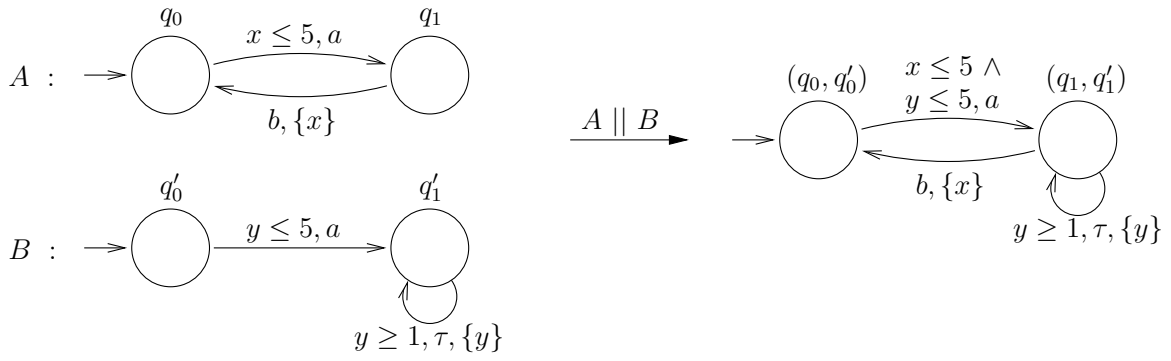


FIG. 6.1 – La composition parallèle || peut introduire des τ-cycles non zénon

Nous venons de voir que la composition parallèle de deux automates peut contenir des cycles non zénon d'actions internes d'un des automates. Cependant, de tels cycles ne peuvent exister que si l'automate en question en contient également. La proposition 6 donne donc une condition suffisante permettant de s'assurer de l'absence de tels cycles dans une composition.

PROPOSITION 6 *Soient A et B deux automates temporisés, où les actions internes de B (non communes à A) sont renommées par τ. Si B ne contient pas de τ-cycles non zénon, alors $A||B$ ne contient pas de τ-cycles non zénon.*

PREUVE. (par l'absurde) Supposons qu'il n'y ait pas de τ-cycles non zénon dans B, et qu'il existe un τ-cycle non zénon dans $A||B$. Il existe donc dans $A||B$ une exécution non zénon qui, à partir d'un point, ne fait plus que des transitions de temps ou des transitions étiquetées par τ, c'est à dire :
$\exists \rho, i \cdot (\rho \in \Gamma(A||B) \wedge \texttt{time}(\rho) = \infty \wedge i \geq 0 \Rightarrow$

$$\forall k \cdot (k \geq i \Rightarrow (\rho, k) \xrightarrow{\tau} (\rho, k+1) \vee (\rho, k) \xrightarrow{t} (\rho, k+1)))$$

Par construction de $A||B$, s'il existe une transition de temps $(s_A, s_B) \xrightarrow{t} (s'_A, s'_B)$ dans $A||B$, alors les transitions $s_A \xrightarrow{t} s'_A$ et $s_B \xrightarrow{t} s'_B$ devaient exister respectivement dans A et B. De la même façon, si une transition $(s_A, s_B) \xrightarrow{\tau} (s_A, s'_B)$ existe dans $A||B$, alors la transition $s_B \xrightarrow{\tau} s'_B$ doit exister dans B. Donc, par construction de $A||B$, il doit donc exister dans B une exécution qui, à partir d'un point n'effectue plus que des transitions de temps ou des transitions étiquetées par τ, qui est également non zénon. Ce qui est contradictoire avec l'hypothèse de départ que B ne contient pas de τ-cycles non zénon. □

Il suit donc la proposition suivante.

PROPOSITION 7 *Soient A et B deux automates temporisés, où les actions internes de B sont renommées par τ. Si B ne contient pas de τ-cycles non-zénon, alors $A||B \preceq_{\mathcal{S}_d} A$.*

PREUVE. Immédiate grâce à la proposition 6. □

La proposition suivante énonce que $\preceq_{\mathcal{S}_d}$ est compatible avec l'opérateur $||$.

PROPOSITION 8 *Soient A, B et C trois automates temporisés. Si $A \preceq_{\mathcal{S}_d} B$ alors $A||C \preceq_{\mathcal{S}_d} B||C$.*

PREUVE. La preuve est immédiate du fait que les actions internes de $B||C$ par rapport à $A||C$ sont celles de B par rapport à A. □

La propriété de compositionnalité n'est, quant à elle, plus garantie. Rappelons que cette propriété exprime que si B simule A et D simule C alors $B||D$ simule $A||C$. Le fait que A et C ne contiennent pas de τ-cycles non zénon n'implique pas que $A||C$ n'en contient pas non plus. En effet, dans $A||C$, de tels cycles peuvent être formés par un entrelacement des actions non observables de A et C.

Le problème des blocages

Si l'introduction de la sensibilité à la divergence à la τ-simulation temporisée permet encore de disposer de certaines propriétés, ce n'est pas le cas lors de l'ajout du respect de la stabilité. En effet, la relation $\preceq_{\mathcal{S}_s}$ ne possède aucune des trois propriétés. Nous nous focalisons ici sur la propriété d'intégration, mais des arguments simples identiques peuvent être donnés pour les autres propriétés.
La figure 6.2 montre un exemple d'introduction de blocages dans le cadre de cette propriété d'intégration : on considère deux automates temporisés A et B dont les états initiaux sont q_0 et q'_0 (pour plus de lisibilité, nous n'avons représenté qu'une transition partant de chaque état initial), et on souhaite intégrer A dans B. Dans A, les configurations $(q_0, \mathbf{V})$, où $\mathbf{v}$ satisfait $x \leq 5$, ne sont pas des blocages, puisqu'à partir de celles-ci, on peut laisser le temps s'écouler et prendre la transition étiquetée par a. Plus formellement, l'ensemble des valuations définies par le prédicat $\texttt{free}(q_0)$ sont donc celles satisfaisant la contrainte $x \leq 5$. Considérons à présent la composition $A||B$. On remarque qu'aucune des valuations possibles pour l'état (q_0, q'_0) ne permet de laisser passer du temps, puis de prendre la transition a. En effet, par définition, la configuration initiale de $A||B$ est $((q_0, q'_0), \mathbf{0})$. Depuis cette configuration, on ne peut atteindre aucune configuration dont la valuation satisfait la garde $x \leq 5 \wedge y > 6$ (car $x = y$), et donc qui permet de prendre la transition a. Formellement, on a donc $\texttt{free}((q_0, q'_0)) = \emptyset$. Par conséquent, toutes les configurations correspondant à l'état (q_0, q'_0) sont des blocages, alors que dans A, seules les configurations $(q_0, x > 5)$ en sont. Cet exemple montre donc que la τ-simulation temporisée respectant la stabilité ne possède pas la propriété d'intégration, et illustre également le fait que l'opérateur || peut introduire des blocages.

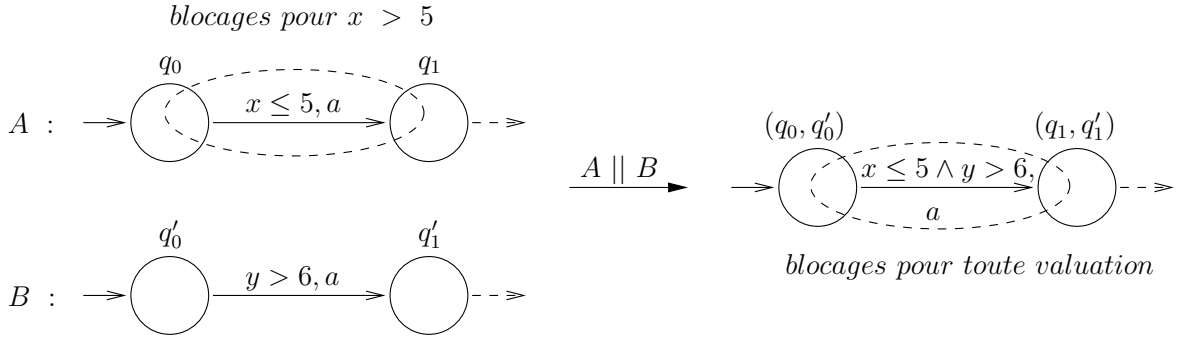


FIG. 6.2 – La composition parallèle || peut introduire des blocages

6.1.3 Bilan

Effectuons à présent un bilan quant à l'intérêt des relations de simulation pour une modélisation incrémentale des systèmes utilisant l'opérateur ||.
La τ-simulation temporisée préserve les propriétés de sûreté. Les propositions 3, 4 et 5 assurent donc que les propriétés de sûreté sont préservées "gratuitement" par l'intégration de composants, et que le raffinement pourra être vérifié de manière compositionnelle

lorsque seul ce type de propriétés est considéré. Ce n'est pas le cas si l'on considère des propriétés de vivacité, du fait que la τ-simulation temporisée DS ne possède pas les propriétés requises vis-à-vis de cet opérateur de composition. La difficulté principale provient du fait que l'opérateur de composition || peut introduire des blocages.
C'est d'ailleurs pour cette raison que d'autres opérateurs de composition ont été définis, possédant justement cette capacité à ne pas créer de blocages. En particulier, c'est le cas de l'opérateur | qui, rappelons-le, a été défini par [Bor98, BST97, BS00] et dont nous avons rappelé la définition dans la section 4.1.2.

6.2 Systèmes à composants utilisant l'opérateur *à la CCS* avec priorités

Rappelons tout d'abord que cette composition parallèle est initialement définie sur des automates temporisés avec deadlines, par l'entrelacement de toutes les transitions de chacun des automates à composer, et la synchronisation de certaines actions, donnée par une fonction de synchronisation. Une priorité infinie est donnée à l'action synchronisée par rapport aux actions entrelacées participant à cette synchronisation. De plus, cette composition a été étendue par l'utilisation de différents modes de synchronisation : les modes AND, MAX et MIN. Nous étudions dans cette section l'intérêt des relations de τ-simulations temporisées lors de l'utilisation de cet opérateur de composition.

6.2.1 Propriétés de la τ-simulation temporisée DS

Nous étudions les propriétés de la τ-simulation temporisée DS avec cette composition parallèle. Cette composition parallèle étant définie sur des automates temporisés avec deadlines, et la τ-simulation sur des automates temporisés, nous devons préciser les points suivants avant de pouvoir débuter cette étude.

Supposons que l'on souhaite intégrer A dans B. Les actions observables dans $A|B$ seront les actions de A ($\in Labels_A$), ainsi que les actions résultant de la synchronisation d'actions de A et de B ($\in Labels_{sync}$). Nous avons supposé dans la définition des simulations que les actions observables de $A|B$ portaient le même nom que les actions observables de A. Nous supposerons, dans le cas de cette composition, que les actions observables de $A|B$ seront étiquetées par des actions de $Labels_A \cup Labels_{sync}$. La clause *simulation stricte* de la τ-simulation temporisée est donc modifiée en conséquence : si une action synchronisée peut être effectuée dans $A|B$, alors l'action de A participant à la synchronisation doit également pouvoir être effectuée (au lieu de la même action, étiquetée par le même nom, dans la définition d'origine).
De plus, dans ce qui suit, nous étendons les notations $\preceq$ et $\preceq_{ds}$ aux automates temporisés avec deadlines (rappelons qu'elles sont au départ définies sur les automates temporisés classiques). Cette extension ne pose pas de problèmes car les simulations temporisées sont définies sur la sémantique des automates temporisés, qui est donnée par un système de transitions, et la sémantique des automates temporisés avec deadlines est donnée par un système de transitions de même nature. La notion de τ-cycles non zénon peut également

être étendue à ce type d'automates temporisés pour la même raison.

Considérons deux automates temporisés avec deadlines A et B. La proposition 9 montre que, sous certaines conditions, A DS-τ-simule son intégration dans B.

PROPOSITION 9 *Soient A et B deux automates temporisés avec deadlines, et $A|B$ l'automate résultant de leur composition avec l'opérateur $|$, où les actions internes de B sont renommées par τ. Si B ne contient pas de τ-cycles non zénon alors $A|B \preceq_{\mathcal{S}_{ds}} A$.*

PREUVE. Cette preuve va être menée de la même façon que la preuve de la proposition 3. On considère toujours S_{AB} et S_A les ensembles de configurations respectifs de $A|B$ et A, et (q_{0_A}, v_{0_A}) et (q_{0_B}, v_{0_B}) les configurations initiales respectives de A et B. Par construction de $A|B$, sa configuration initiale est le couple $((q_{0_A}, v_{0_A}), (q_{0_B}, v_{0_B}))$.
Comme précédemment, il faut prouver que $((q_{0_A}, v_{0_A}), (q_{0_B}, v_{0_B}))\mathcal{S}_{ds}(q_{0_A}, v_{0_A})$. Par définition, $\mathcal{S}_{ds}$ est la plus grande τ-simulation temporisée DS, c'est à dire la plus grande relation incluse dans $S_{AB} \times S_A$ satisfaisant les clauses de la définition 10. Nous considérons donc ici encore une relation $R \subseteq S_{AB} \times S_A$, et cherchons à prouver qu'elle est incluse dans $\mathcal{S}_{ds}$ en démontrant qu'elle satisfait toutes les clauses de la définition de $\mathcal{S}_{ds}$. Nous utilisons la même relation R que dans la preuve de la proposition 3, définie telle que $\forall((q_A, v_A), (q_B, v_B)) \in S_{AB}$,

$$(((q_A, v_A), (q_B, v_B)), (q'_A, v'_A)) \in R \text{ si } (q_A, v_A) = (q'_A, v'_A).$$

Prouvons à présent que R est une τ-simulation temporisée DS.
Soit $(((q_A, v_A), (q_B, v_B)), (q_A, v_A)) \in R$.

1. *Simulation stricte* : soit une transition $((q_A, v_A), (q_B, v_B)) \overset{g,a,r}{\rightarrow} ((q'_A, v'_A), (q'_B, v'_B))$ telle que soit $a \in Labels_A$, soit $a \in Labels_{sync}$, c'est-à-dire que a est soit une action entrelacée de A, soit une action se synchronisant avec une action de B. Dans les deux cas, si cette transition discrète existe au niveau sémantique, alors une transition $(q_A, q_B) \overset{g,d,a,r}{\rightarrow} (q'_A, q'_B)$ existe dans $A|B$ (dans le cas d'une transition entrelacée, notons tout de même que $q'_B = q_B$ et $v'_B = v_B$, mais cela ne modifie en rien la preuve qui suit).

   Si a est une action synchronisée, alors elle résulte de la synchronisation d'une transition $q_A \overset{g_A,d_A,a_A,r_A}{\rightarrow} q'_A$ de A et d'une transition $q_B \overset{g_B,d_B,a_B,r_B}{\rightarrow} q'_B$, où la fonction de synchronisation donne $a_A|a_B = a$. Par définition, on a $g = g_A \wedge g_B$ et $r = r_A \cup r_B$. Si la transition discrète $((q_A, v_A), (q_B, v_B)) \overset{g_2,d_2,a,r_2}{\rightarrow} ((q'_A, v'_A), (q'_B, v'_B))$ existe dans le graphe sémantique de $A|B$, c'est que $(v_A \cup v_B)$ satisfait g et donc que $(v_A \cup v_B)$ satisfait g_A. Comme v_A et g_A ne concernent que les horloges de A, on a trivialement v_A satisfait g_A. De plus, comme $(v'_A \cup v'_B) = [r := 0](v_A \cup v_B)$, et que $r \cap X_A = r_A$, alors il suit que $v'_A = [r_A := 0]v_A$, et donc la transition $(q_A, v_A) \overset{g_A,a_A,r_A}{\rightarrow} (q'_A, v'_A)$ existe dans le graphe sémantique de A. Par définition de R, on a bien $(((q'_A, v'_A), (q_B, v_B)), (q'_A, v'_A)) \in R$.

Si a est une action entrelacée, elle provient du fait qu'une transition $q_A \overset{g_A,d_A,a,r_A}{\rightarrow} q'_A$ existe dans A. La garde g peut soit être égale à g_A, soit avoir été modifiée par l'utilisation de priorités. Dans les deux cas, on a toujours $g \subseteq g_A$. Par le même raisonnement que dans le cas des transitions synchronisées, on peut donc déduire que la transition $(q_A, v_A) \overset{g_A,a,r_A}{\rightarrow} (q'_A, v'_A)$ existe dans le graphe sémantique de A et par définition de R, on a également $(((q'_A, v'_A), (q_B, v_B)), (q'_A, v'_A)) \in R$.
Dans les deux cas, R satisfait donc la clause *simulation stricte*.

2. *Egalité des délais* : considérons une transition $((q_A, v_A), (q_B, v_B)) \overset{t}{\rightarrow} ((q_A, v_A + t), (q_B, v_B + t))$ de $A|B$. Cette transition apparait dans le graphe sémantique $A|B$ si l'invariant de l'état (q_A, q_B) est satisfait par les valuations $(v_A \cup v_B)+t', \forall t' < t$. Rappelons que les invariants de $A|B$ sont déduits des deadlines des transitions sortantes de (q_A, q_B). Cela signifie donc, en termes de deadlines, que la transition précédente existe si les valuations $(v_A \cup v_B)+t'$ ne satisfont aucune des deadlines des transitions discrètes sortantes de (q_A, q_B). Pour que la transition $(q_A, v_A) \overset{t}{\rightarrow} (q_A, v_A + t)$ existe dans le graphe sémantique de A, les valuations $v_A + t', \forall t' < t$ ne doivent satisfaire aucune des deadlines des transitions discrètes issues de q_A.

   Les transitions discrètes issues de (q_A, q_B) peuvent être soit uniquement des transitions entrelacées, soit également des transitions synchronisées. Considérons tout d'abord que seules des transitions entrelacées sont issues de (q_A, q_B). S'il n'existe pas de transitions de B issues de (q_A, q_B), alors seules les deadlines des transitions de A conditionnent le passage du temps depuis $((q_A, v_A), (q_B, v_B))$, et donc il peut s'écouler de la même manière que dans A. On a donc immédiatement que si la transition $((q_A, v_A), (q_B, v_B)) \overset{t}{\rightarrow} ((q_A, v_A + t), (q_B, v_B + t))$ existe dans $A|B$ alors la transition $(q_A, v_A) \overset{t}{\rightarrow} (q_A, v_A + t)$ existe dans A. Si des transitions de B sont également issues de l'état (q_A, q_B) et que leurs deadlines sont atteintes *plus tard* que celles des actions de A, le passage du temps sera là aussi conditionné par les deadlines de A et on revient au cas précédent. Dans le cas contraire, s'il existe une deadline d'une transition de B atteinte *plus tôt* que celles des transitions de A, alors le passage du temps sera conditionné par cette deadline, et puisque la deadline est plus forte, le temps s'écoulera moins que dans A, ce qui est recherché puisqu'on impose que le temps s'écoule au maximum autant que dans A. Il suit donc que la transition $(q_A, v_A) \overset{t}{\rightarrow} (q_A, v_A + t)$ existe dans A.
   Le cas des transitions synchronisées revient aux cas précédents, puisque, par définition de $|$, la deadline d'une transition synchronisée est égale à la plus forte deadline des transitions participant à la synchronisation.
   On a donc, dans tous les cas, s'il existe une transition $((q_A, v_A), (q_B, v_B)) \overset{t}{\rightarrow} ((q_A, v_A + t), (q_B, v_B + t))$ de $A|B$ existe, alors il existe une transition $(q_A, v_A) \overset{t}{\rightarrow} (q_A, v_A + t)$ dans A'. De plus, par définition de R, on a bien $(((q_A, v_A + t), (q_B, v_B + t)), (q_A, v_A + t)) \in R$, et donc R satisfait la clause *égalité des délais*.

3. *Bégaiment des τ-transitions* : soit une transition $((q_A, v_A), (q_B, v_B)) \overset{e_2}{\rightarrow} ((q'_A, v'_A), (q'_B, v'_B))$

de $A|B$ telle que $\texttt{label}(e_2) = \tau$. Les τ-transitions sont les actions non observables de $A|B$, c'est-à-dire les actions de B ne se synchronisant pas avec une action de A. Par définition de $|$, ces actions apparaissent dans $A|B$ uniquement comme des actions entrelacées. Ainsi, $(q'_A, v'_A) = (q_A, v_A)$ et par définition de R, on a bien $(((q_A, v_A), (q'_B, v'_B)), (q_A, v_A)) \in R$. R satisfait donc la clause *bégaiment des τ-transitions*

4. *Respect du décor des états* : immédiat, tout comme pour l'opérateur $||$.

5. *Egalité des valuations des horloges communes* : immédiat, par définition de R et grâce au fait que les ensembles d'horloges de A et B sont disjoints.

6. *Sensibilité à la divergence* : par hypothèse, B ne possède pas de τ-cycles non zénon. Par les mêmes arguments que dans le cas de l'opérateur $||$, $A|B$ n'en contient pas non plus, et R satisfait la sensibilité à la divergence.

7. *Respect de la stabilité* : immédiat, dû à l'entrelacement systématique des actions de A et B dans la définition de $|$.

La relation R satisfait donc toutes les clauses de la définition 10, d'où $R \subseteq \mathcal{S}_{ds}$. Comme, par définition de R, on a $(((q_{0_A}, v_{0_A}), (q_{0_B}, v_{0_B})), (q_{0_A}, q_{0_B})) \in R$, alors on a également $((q_{0_A}, v_{0_A}), (q_{0_B}, v_{0_B}))\mathcal{S}_{ds}(q_{0_A}, q_{0_B})$. □

PROPOSITION 10 *Soient A, B et C trois automates temporisés avec deadlines. Si $A \preceq_{\mathcal{S}_{ds}} B$ alors $A|C \preceq_{\mathcal{S}_{ds}} B|C$.*

PREUVE. Nous ne détaillons pas la preuve des clauses de la τ-simulation temporisée, qui est presque la même que celle de la proposition 4, en prenant en compte les spécificités de la composition parallèle $|$ (notamment l'utilisation de deadlines), comme nous l'avons fait dans la preuve précédente.
Concernant les clauses *respect de la stabilité* et *sensibilité à la divergence*, elles sont également vérifiées. Concernant la sensibilité à la divergence, comme les τ-transitions de $A|C$ sont les actions de $A \backslash (B \cup C)$ (l'ajout de C à A n'ajoute pas de τ-transitions) et que $A \preceq_{\mathcal{S}_{ds}} B$, alors il n'existe pas de τ-cycles non-zénon dans $A|C$. Concernant le respect de la stabilité, la preuve est ici immédiate du fait que $A \preceq_{\mathcal{S}_{ds}} B$.

□

PROPOSITION 11 *Soit A, B, C et D des automates temporisés avec deadlines. Les actions internes de $B|D$ sont renommées par τ. Si $A \preceq_{\mathcal{S}_{ds}} B$, $C \preceq_{\mathcal{S}_{ds}} D$ et $A|C$ ne contient pas de τ-cycles non zénon, alors $A|C \preceq_{\mathcal{S}_{ds}} B|D$.*

PREUVE. Immédiate grâce à la proposition 10. □

6.2.2 Utilisation des modes MIN et MAX pour la synchronisation

Différents modes de synchronisation ont été proposés pour être utilisés avec cette composition parallèle : les modes AND, MAX et MIN. Tous les résultats que nous avons énoncés jusqu'à présent concernaient uniquement le mode AND. Nous étudions à présent ce qu'il en est lors de l'utilisation des modes MAX et MIN. Nous noterons par la suite $|_{AND}$, $|_{MIN}$ et $|_{MAX}$ l'opérateur | suivant s'il utilise un mode de synchronisation AND, MIN ou MAX.

Mode MIN

Le mode MIN correspond à une synchronisation avec interruption, c'est-à-dire que dès que l'une des actions participant à la synchronisation peut être effectuée, elle force l'exécution de la synchronisation si l'autre action peut être effectuée dans le futur. La garde de l'action synchronisée est modifiée pour prendre en compte ce point de vue (cf. section 4.1.2).
Tout comme pour le mode AND, cette modification consiste en fait à renforcer la garde, permettant ainsi aux clauses concernées, la *simulation stricte* et l'*égalité des délais* d'être respectées. Nous avons les propositions suivantes pour le mode MIN.

PROPOSITION 12 *Soient A et B deux automates temporisés avec deadlines, et $A|_{MIN}B$ l'automate résultant de leur composition avec l'opérateur $|_{MIN}$, où les actions internes de B sont renommées par τ. Si B ne contient pas de τ-cycles non zénon alors $A|_{MIN}B \preceq_{\mathcal{S}_{ds}} A$.*

PROPOSITION 13 *Soient A, B et C trois automates temporisés avec deadlines. Si $A \preceq_{\mathcal{S}_{ds}} B$ alors $A|_{MIN}C \preceq_{\mathcal{S}_{ds}} B|_{MIN}C$.*

PROPOSITION 14 *Soit A, B, C et D des automates temporisés avec deadlines. Les actions internes de $B|_{MIN}D$ sont renommées par τ. Si $A \preceq_{\mathcal{S}_{ds}} B$, $C \preceq_{\mathcal{S}_{ds}} D$ et $B|_{MIN}D$ ne contient pas de τ-cycles non zénon, alors $A|_{MIN}C \preceq_{\mathcal{S}_{ds}} B|_{MIN}D$.*

Mode MAX

Le mode MAX correspond à une synchronisation avec attente, c'est-à-dire que si l'une des actions participant à la synchronisation est activable, elle attend que l'autre le soit aussi pour que l'action synchronisée ait lieu. Il suit immédiatement de cette description informelle que la clause *simulation stricte* n'est pas vérifiée. La figure 6.3 illustre ce problème. Dans cet exemple, on considère deux automates temporisés A et B, et leur composition avec l'opérateur $|_{MAX}$ (pour plus de lisibilité, nous n'avons représenté qu'une transition partant de l'état initial de chaque automate). Supposons que les actions a et b se synchronisent. La garde de $a|b$ en mode MAX est alors égale à :

$$(x \leq 5 \wedge \swarrow x \leq 7) \vee (\swarrow x \leq 5 \wedge x \leq 7) = x \leq 5 \vee y \leq 7.$$

Supposons que l'on souhaite intégrer A dans B, et donc vérifier que $A|_{MAX}B \preceq_{\mathcal{S}_{ds}} A$. On voit ici précisément que la clause *simulation stricte* n'est pas vérifiée. En effet, dans $A|_{MAX}B$, la transition $a|b$ peut être prise tant que $x = y \leq 7$, alors que dans A, la transition a ne peut être prise que jusqu'à $x \leq 5$.

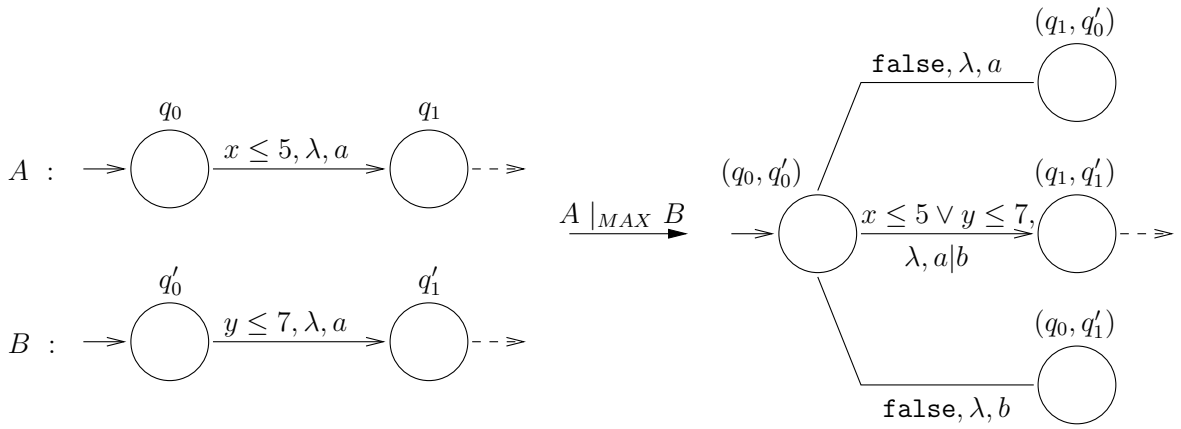


FIG. 6.3 – Le mode MAX ne vérifie pas la *simulation stricte*

Des arguments similaires peuvent être donnés dans le cas de la compatibilité et de la compositionnalité. Ainsi, les propositions 9, 10 et 11 ne sont plus vraies si l'on considère le mode de synchronisation MAX.

REMARQUE 6.1. On remarquera qu'ici, pour plus de simplicité, nous avons considéré qu'un seul mode de synchronisation pouvait être utilisé lors de la composition de deux automates. En pratique, il est tout à fait possible de *mélanger* tous ces modes lors d'une composition : certaines actions se synchronisant en mode AND, d'autres en mode MIN, et d'autres encore en mode MAX. Les résultats concernant les propriétés des τ-simulations temporisées dépendent alors des modes utilisés. Les propositions 9, 10 et 11 sont vraies lorsque les modes de synchronisation utilisés sont AND et/ou MIN. En revanche, ce n'est pas le cas pour le mode MAX.

REMARQUE 6.2. Dans toute cette section, nous nous sommes focalisés sur les propriétés de la τ-simulation temporisée DS par rapport à l'opérateur de composition *à la CCS*. Remarquons que la τ-simulation temporisée possède les trois propriétés d'intégration, compatibilité et compositionnalité. En effet, la τ-simulation temporisée DS est définie à partir de cette τ-simulation temporisée en ajoutant deux clauses : la sensibilité à la divergence et le respect de la stabilité. Comme les trois propriétés sont vraies pour la τ-simulation temporisée DS, modulo des hypothèses pour la sensibilité à la divergence, alors elles le sont également pour la τ-simulation temporisée, sans hypothèses.

6.2.3 Bilan

Comme précédemment, nous effectuons un bilan concernant la préservation de propriétés lors d'une intégration de composants et la vérification compositionnelle du raffinement. Nous remarquons que, sous certaines hypothèses (pas de blocages dans le composant à intégrer, absence de τ-cycles dans l'environnement, utilisation des modes de synchronisation AND et MIN), la τ-simulation temporisée DS est adaptée à l'opérateur | : intégration correcte assurée *gratuitement*, ainsi que la propriété de compatibilité. Ces propriétés garantissent immédiatement qu'une intégration de composants effectuée à l'aide de l'opérateur | préserve toutes les propriétés MITL. La compositionnalité peut également être assurée, mais au prix d'une vérification globale de la sensibilité à la divergence.
La τ-simulation temporisée est quant à elle bien adaptée à l'opérateur | puisque les trois propriétés d'intégration, de compatibilité et de compositionnalité sont garanties dans ce cas.

6.3 Conclusion

Trois propriétés principales des simulations vis-à-vis des opérateurs de composition sont nécessaires pour montrer leur intérêt dans le cadre de la modélisation incrémentale des systèmes temporisés à base de composants : la compatibilité des relations avec les opérateurs de composition, la propriété d'intégration en vue d'une intégration de composants, et enfin la compositionnalité pour le raffinement.
Nous résumons à présent les résultats présentés dans ce chapitre. L'objectif était d'étudier si les relations de simulation définies possèdent les trois propriétés précédentes par rapport aux deux opérateurs de composition considérés dans ce document. Nous exposons par la suite les résultats obtenus concernant ces trois propriétés pour d'autres relations de simulations définies dans la littérature.

Synthèse

Les résultats obtenus sont résumés dans le tableau 6.1. Le tableau 6.2 donne une interprétation de ces résultats en termes de préservation de propriétés lors d'un développement incrémental. Les abréviations *hyp. div. 1* et *hyp. div. 2* pour "hypothèse sur la sensibilité à la divergence 1 et 2" expriment le fait que les propriétés sont vraies sous des hypothèses concernant l'absence de τ-cycles (ces hypothèses sont celles respectivement données dans les propositions 9 et 11).

Ces tableaux montrent que les deux relations sont bien adaptées lorsque l'opérateur considéré est l'opérateur | *à la CCS* avec priorités, quand les modes de synchronisation AND et MIN sont utilisés. En particulier, pour la propriété d'intégration, une condition suffisante simple pour la sensibilité à la divergence permet de garantir cette propriété. Les deux relations sont de plus compatibles avec cet opérateur. La propriété de compositionnalité peut également être garantie, mais au prix d'une vérification de la sensibilité à la divergence qui, elle, est non compositionnelle. Seule l'utilisation du mode MAX ne permet de garantir aucune propriété. Il est cependant à noter qu'en pratique, le mode le

	Opérateur *à la CSP*	Opérateur *à la CCS* + priorités		
		AND	MIN	MAX
	τ-simulation temporisée			
Intégration	OK	OK	OK	nOK
Compatibilité	OK	OK	OK	nOK
Compositionnalité	OK	OK	OK	nOK
	τ-simulation temporisée DS			
Intégration	nOK	OK (hyp. div. 1)	OK (hyp. div. 1)	nOK
Compatibilité	nOK	OK	OK	nOK
Compositionnalité	nOK	OK (hyp. div. 2)	OK (hyp. div. 2)	nOK

TAB. 6.1 – Bilan des propriétés des simulations par rapport aux opérateurs de composition

	Opérateur *à la CSP*	Opérateur *à la CCS* + priorités		
		AND	MIN	MAX
Propriétés préservées par intégration de composants	sûreté	MITL, absence de blocages, non-zénonisme (hyp.div. 1)		-
Propriétés préservées par une vérification compositionnelle du raffinement	sûreté	MITL, absence de blocages, non-zénonisme (hyp. div. 2)		-

TAB. 6.2 – Bilan pour la préservation de propriétés lors du développement incrémental

plus utilisé reste le mode AND.

La τ-simulation temporisée est bien adaptée à l'utilisation de l'opérateur *à la CSP*. En revanche, ce n'est pas le cas de la τ-simulation temporisée DS. Toutefois, il reste possible de bénéficier des atouts de cette simulation en termes de préservation de propriétés, au prix d'une vérification algorithmique de la relation.

Si, dans les autres cas de figure, l'utilisation des τ-simulations comme cadre formel au développement incrémental a un intérêt pratique certain, ce n'est pas nécessairement le cas pour la τ-simulation temporisée DS et l'opérateur *à la CSP*. En effet, il est nécessaire d'étudier si la vérification systématique de la τ-simulation temporisée DS pour garantir la préservation de propriétés n'enlève pas d'intérêt aux méthodes de développement incrémental, lorsqu'elles sont formalisées par cette relation. Par exemple, dans le cas d'une intégration de composants, une question essentielle est de savoir si la vérification locale des propriétés et la vérification de la préservation, par le biais de la τ-simulation temporisée DS, est plus efficace qu'une vérification classique de ces propriétés, directement sur le modèle complet du système. Cette efficacité peut notamment être évaluée en termes de

temps de calcul.

La partie suivante est justement dédiée à cette étude. Nous présentons dans le chapitre 7 les algorithmes mis en place pour vérifier la τ-simulation temporisée DS. Puis, le chapitre 8 présentera le prototype réalisé pour la vérification de cette simulation, pour des systèmes à base de composants utilisant l'opérateur *à la CSP*. Nous finirons dans le chapitre 9 par évaluer l'impact en pratique de l'utilisation de cette simulation en présentant les expérimentations menées.

Comparaison avec d'autres travaux

Dans le chapitre précédent, deux autres relations de simulation temporisée, précédemment définies dans la littérature, ont été identifiées : la simulation temporisée de [TAKB96, Tas98], et la *ready* simulation temporisée de [JLS00]. Ces deux relations peuvent être comparées à la τ-simulation temporisée définie. Rappelons que nous n'avons pas trouvé dans la littérature de notion de simulation se rapprochant de la τ-simulation temporisée DS, c'est-à-dire incluant des conditions supplémentaires pour prendre en compte notamment la préservation de propriétés de vivacité.

Dans ce chapitre, nous avons souligné l'importance des propriétés d'intégration, de compatibilité et de compositionnalité. La simulation temporisée possède ces propriétés vis-à-vis d'un opérateur définissant une composition totalement synchrone. La *ready* simulation temporisée est la notion de simulation qui se rapproche le plus de la τ-simulation temporisée que nous avons définie. En particulier, sa définition prend en compte la présence d'actions non-observables. Elle possède les trois propriétés vis-à-vis d'un opérateur de composition utilisant le même paradigme que celui que nous avons nommé *CSP*. Cependant, à la différence de l'étude effectuée dans la section 6.1.1, une hypothèse est effectuée sur l'absence d'activité interne dans les automates (c'est-à-dire l'absence de τ-transitions) pour bénéficier des trois propriétés. En particulier, la propriété de compositionnalité est exprimée de la manière suivante : si $A \preceq B$ et $C \preceq D$, et que B et D ne possèdent pas d'activité interne, alors $A||C \preceq B||D$, où $\preceq$ est la relation de *ready* simulation temporisée[14].

[14] Rappelons que la *ready* simulation temporisée permet de prendre en compte des variables partagées. Des hypothèses sur les variables partagées de A, B, C et D sont donc également effectuées pour cette propriété de compositionnalité.

Troisième partie

Implantation et études de cas

7

Vérifier les simulations

Sommaire

Dans les chapitres précédents, nous avons présenté des relations de τ-simulations temporisées, ainsi que leurs facultés à préserver des propriétés linéaires, et avons étudié leur compatibilité avec différents opérateurs de composition. Les définitions données jusqu'à présent étaient formulées à un niveau sémantique, c'est à dire sur les configurations des automates temporisés.

Or, un automate temporisé possédant un nombre infini de configurations, il est impossible d'implanter directement la définition sémantique donnée dans le chapitre 5. En pratique, les outils d'analyse et de vérification des automates temporisés utilisent la représentation symbolique basée sur les zones, présentée dans le chapitre 2. Nous redéfinissons dans ce chapitre les relations de simulations sur les zones, et montrons que cette relation symbolique implique la relation sémantique sur les configurations.

7.1 Les τ-simulations temporisées sur les zones

Nous nous concentrons dans ce chapitre sur la τ-simulation temporisée DS. La définition symbolique pour la τ-simulation temporisée (appelée $\mathcal{Z}$) peut être obtenue très simplement, en ne gardant de la τ-simulation temporisée DS que les clauses la concernant.

Nous voulons définir la relation au niveau symbolique dans l'objectif de la vérifier algorithmiquement *à la place* de la relation sémantique. Toutefois, il est nécessaire que

les résultats obtenus pour cette vérification impliquent les résultats qui auraient été obtenus avec la définition sémantique de la relation. Notons ici qu'une implication suffit pour bénéficier des propriétés de préservation, de compatibilité et de compositionnalité de la relation au niveau sémantique. Avant de donner la définition de cette relation symbolique et de prouver formellement cette implication, nous commençons par présenter intuitivement les clauses caractérisant cette relation.

Considérons deux graphes de simulation SG_1 et SG_2, z_1 un état de SG_1 et z_2 un état de SG_2. Commençons tout d'abord par la clause concernant les transitions de temps. Lors de la définition des graphes de simulation, nous avons expliqué intuitivement que le temps s'écoule implicitement *à l'intérieur* des états symboliques du graphe, c'est-à-dire que les transitions de temps sont effectuées en quelque sorte *à l'intérieur* de ces états. Ainsi, dans la définition symbolique de la relation, la clause *égalité des délais* est définie par l'inclusion du polyhèdre de z_2 dans celui de z_1 (modulo une projection sur l'ensemble d'horloges X_1 de SG_1). Ainsi, le temps ne pourra pas s'écouler plus dans z_2 qu'il ne le pouvait dans z_1.
Etudions à présent la clause *simulation stricte*, illustrée par la figure 7.1.

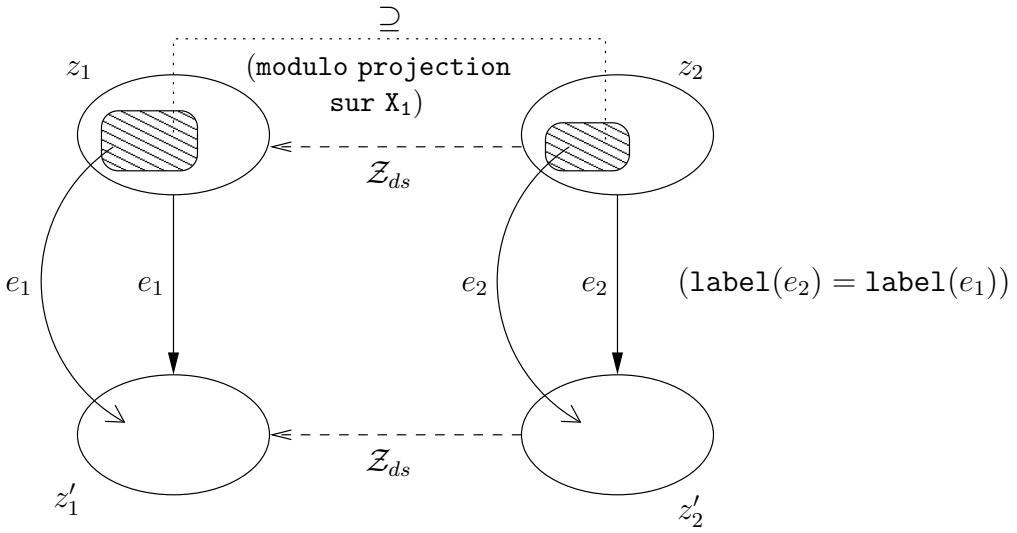


FIG. 7.1 – Clause *simulation stricte* au niveau symbolique

La définition que nous donnons au niveau des états symboliques est basée sur la propriété de post-stabilité du graphe de simulation. Cette propriété permet d'assurer que, étant donnée une transition $z \xrightarrow{e} z'$, tous les successeurs des configurations de z effectuant e sont dans z'. Ainsi, la clause *simulation stricte* est définie de la même façon qu'au niveau sémantique, en imposant que si la transition $z_2 \xrightarrow{e_2} z'_2$ existe, alors une transition $z_1 \xrightarrow{e_2} z'_1$ existe également telle que e_1 et e_2 ont la même étiquette. Le graphe de simulation ne possède pas la propriété de pré-stabilité, c'est à dire qu'une transition symbolique $z \xrightarrow{e} z'$ n'implique pas que tous les états de z puissent effectuer la transition e. Nous devons imposer une condition supplémentaire : les configurations de z_2 effectuant cette transition e_2

correspondent toutes à une configuration de z_1 effectuant la même transition e_1, modulo une projection sur les horloges identiques. Cette condition permet d'assurer qu'il n'existe pas de transition discrète inscrite dans $z_2 \xrightarrow{e_2} z_2'$ ne correspondant à aucune transition discrète inscrite dans $z_1 \xrightarrow{e_1} z_1'$. Dans la figure 7.1, la partie hachurée dans chaque état z_1 et z_2 correspond à l'ensemble de configurations de z_1 et z_2 effectuant respectivement les transitions e_1 et e_2 (ce que nous dénoterons par la suite par un prédicat nommé `src_val`).

Les autres clauses (bégaiement des τ-transitions, respect du décor des états, respect de la stabilité et sensibilité à la divergence) sont une simple extension de la définition sémantique. Notamment, pour la sensibilité à la divergence, nous étendons la notion de τ-cycles non zénon aux graphes de simulation.

τ-cycles non-zénon dans un graphe de simulation. La définition donnée sur les exécutions d'un automate temporisé peut être étendue directement aux chemins d'un graphe de simulation. Ainsi, on dit qu'un graphe de simulation SG ne contient pas de τ-cycles non-zénon si :
$\forall \pi, k \cdot (\pi \in \Pi(SG) \wedge \texttt{non-zenon}(\pi) \ \wedge k \geq 0 \Rightarrow$

$$\exists k', e \cdot (k' \geq k \wedge (\pi, k') \xrightarrow{e} (\pi, k'+1) \wedge \texttt{label}(e) \neq \tau)).$$

Nous définissons à présent la version symbolique de la τ-simulation temporisée DS. Nous avons utilisé les mêmes noms pour les clauses de la définition pour bien faire apparaître la correspondance avec la définition sémantique. L'opérateur $\texttt{src\_val}(z, e, z')$ utilisé dans la clause 1 représente l'ensemble des valuations de l'état z à partir desquelles l'état z' est atteint par la transition e. Formellement, $\texttt{src\_val}(z, e, z') = \texttt{poly}(\texttt{pre}(e, z') \cap z)$

Définition 13 (τ-simulation temporisée DS symbolique) *Soient* $SG_1 = \langle \mathrm{Z}_1, \mathrm{z}_{0_1}, \mathrm{Labels}_1, \mathcal{E}_1 \rangle$ *et* $SG_2 = \langle \mathrm{Z}_2, \mathrm{z}_{0_2}, \mathrm{Labels}_2, \mathcal{E}_2 \rangle$ *deux graphes de simulation, obtenus respectivement à partir de deux automates temporisés* A_1 *et* A_2. *La τ-simulation temporisée DS symbolique* $\mathcal{Z}_{ds}$ *est la plus grande relation binaire incluse dans* $Z_2 \times Z_1$, *telle que* $z_2 \mathcal{Z}_{ds} z_1$ *si les conditions suivantes sont vérifiées :*

1. *Simulation stricte :*
   $z_2 \xrightarrow{e_2} z_2' \wedge \texttt{label}(e_2) \in \mathit{Labels}_1 \Rightarrow$
   $\exists z_1' \cdot (z_1 \xrightarrow{e_1} z_1' \wedge \texttt{label}(e_1) = \texttt{label}(e_2) \wedge$
   $\texttt{src\_val}(z_2, e_2, z_2') \rfloor_{X_1} \subseteq \texttt{src\_val}(z_1, e_1, z_1') \ \wedge \ z_2' \ \mathcal{Z}_{ds} \ z_1').$
2. *Egalité des délais et respect des valuations des horloges communes :*
   $\texttt{poly}(z_2) \rfloor_{X_1} \subseteq \texttt{poly}(z_1)$.
3. *Bégaiement des τ-transitions :*
   $z_2 \xrightarrow{e_2} z_2' \wedge \texttt{label}(e_2) = \tau \Rightarrow z_2' \ \mathcal{Z}_{ds} \ z_1$.
4. *Respect du décor des états :*
   $(\texttt{disc}(z_2), \texttt{disc}(z_1)) \models_c P_c$.
5. *Sensibilité à la divergence :*
   SG_2 *ne contient pas de τ-cycles non zénon.*
6. *Respect de la stabilité :*
   $(\texttt{poly}(z_2) \backslash \texttt{free}(\texttt{disc}(z_2)) \rfloor_{X_1} \subseteq \texttt{poly}(z_1) \backslash \texttt{free}(\texttt{disc}(z_1))$.

Comme pour les définitions sémantiques, nous étendons cette relation aux graphes de simulation.

DÉFINITION 14 (EXTENSION DE $\mathcal{Z}_{ds}$ AUX GRAPHES DE SIMULATION) *Soient SG_1 et SG_2 deux graphes de simulation, dont les états initiaux sont respectivement z_{0_1} et z_{0_2}. On dit que SG_1 DS-τ-simule SG_2, noté $SG_2 \preceq_{\mathcal{Z}_{ds}} SG_1$, si $z_{0_2} \mathcal{Z}_{ds} z_{0_1}$.*

7.2 Implication des relations

Nous avons prouvé précédemment que la τ-simulation temporisée DS $\mathcal{S}_{ds}$ préserve toutes les propriétés MITL. Pour bénéficier de cette préservation au niveau des graphes de simulation, nous allons prouver que cette relation est impliquée par sa version symbolique $\mathcal{Z}_{ds}$ (lemmes 7 et 8). En d'autres termes, nous allons prouver que si un graphe de simulation DS-τ-simule un autre graphe de simulation, alors l'automate temporisé duquel est issu le premier DS-τ-simule également l'automate temporisé duquel est issu le second (théorème 5).

LEMME 7 *Soient A un automate temporisé et c la plus grande constante apparaissant dans une contrainte de A. Soit $SG(A,c)$ le graphe de simulation construit à partir de A. Si A ne contient pas de τ-cycles non-zénon, alors $SG(A,c)$ ne contient pas de τ-cycles non-zénon, et inversement.*

PREUVE. La preuve découle directement du lemme 3. □

LEMME 8 *Soient SG_1 et SG_2 deux graphes de simulation. On considère (q_1,ζ_1) et (q_2,ζ_2) deux états respectifs de SG_1 et SG_2 tels que $(q_2,\zeta_2)\mathcal{Z}_{ds}(q_1,\zeta_1)$. On a : pour chaque configuration (q_2,v_2) de (q_2,ζ_2), il existe une unique configuration (q_1,v_1) de (q_1,ζ_1) telle que $(q_2,v_2)\mathcal{S}_{ds}(q_1,v_1)$. Formellement :*

$$(q_2,\zeta_2)\ \mathcal{Z}_{ds}\ (q_1,\zeta_1) \Rightarrow \forall v_2 \cdot (v_2 \in \zeta_2 \Rightarrow \exists v_1 \cdot (v_1 \in \zeta_1 \wedge (q_2,v_2)\ \mathcal{S}_{ds}\ (q_1,v_1)))(\star)\ .$$

PREUVE. La preuve est effectuée par induction point-fixe.

Nous définissons une fonction $\mathcal{F} : Z_2 \times Z_1 \rightarrow Z_2 \times Z_1$ de la manière suivante. Si $\mathcal{Z}_{ds} \subseteq Z_2 \times Z_1$, alors $((q_2,\zeta_2),(q_1,\zeta_1)) \in \mathcal{F}(\mathcal{Z}_{ds})$ ssi les clauses de la définition 13 sont vérifiées, c'est à dire :

1. *Simulation stricte :*
   $z_2 \xrightarrow{e_2} z_2' \wedge \texttt{label}(e_2) \in Labels_1 \Rightarrow$
   $\exists z_1' \cdot (z_1 \xrightarrow{e_1} z_1' \wedge \texttt{label}(e_1) = \texttt{label}(e_2) \wedge$
   $\texttt{src\_val}(z_2,e_2,z_2')\rfloor_{X_1} \subseteq \texttt{src\_val}(z_1,e_1,z_1') \wedge\ z_2'\ \mathcal{Z}_{ds}\ z_1')$.
2. *Bégaiement des τ-transitions :*
   $z_2 \xrightarrow{e_2} z_2' \wedge \texttt{label}(e_2) = \tau \Rightarrow z_2'\ \mathcal{Z}_{ds}\ z_1$.
3. *Respect du décor des états :*
   $(\texttt{disc}(z_2),\texttt{disc}(z_1)) \models_c P_c$.

4. *Respect des valuations des horloges communes :*
   $\texttt{poly}(z_2)\rfloor_{X_1} \subseteq \texttt{poly}(z_1)$.

5. *Sensibilité à la divergence :*
   SG_2 ne contient pas de τ-cycles non zénon.
6. *Respect de la stabilité :*
   $(\texttt{poly}(z_2)\backslash\texttt{free}(\texttt{disc}(z_2))\rfloor_{X_1} \subseteq \texttt{poly}(z_1)\backslash\texttt{free}(\texttt{disc}(q_1))$.

Dans la définition 14, nous nous intéressons donc au plus grand point fixe de $\mathcal{F}$. La fonction $\mathcal{F}$ est trivialement monotone (et les ensembles Z_2 et Z_1 finis), ainsi nous pouvons raisonner inductivement sur les pré-points fixes de cette fonction.

Le principe de cette induction est le suivant : étant donné un prédicat P, cette induction permet de prouver $P(\mathcal{Z}_{ds})$, en prouvant $P(\mathcal{F}(\mathcal{Z}_{ds}))$, et en prenant comme hypothèse d'induction que $P(\mathcal{Z}_{ds})$ est vrai.
Par conséquent, nous supposons que $(\star)$ est vrai pour $\mathcal{Z}_{ds}$, et prouvons que $(\star)$ est également vrai pour $\mathcal{F}(\mathcal{Z}_{ds})$. En d'autres termes, nous voulons prouver que :

$$((q_2,\zeta_2),(q_1,\zeta_1)) \in \mathcal{F}(\mathcal{Z}_{ds}) \Rightarrow \forall v_2 \cdot (v_2 \in \zeta_2 \Rightarrow \exists v_1 \cdot (v_1 \in \zeta_1 \wedge (q_2,v_2)\mathcal{S}_{ds}(q_1,v_1)))$$

La relation $\mathcal{S}_{ds}$ est la plus grande τ-simulation temporisée sur $S_2 \times S_1$. Nous allons considérer chacune des clauses de la définition de cette simulation, en rappelant à chaque fois, pour plus de lisibilité, ce que nous cherchons à prouver. Nous commençons par la clause *Respect des valuations des horloges communes* qui sera utilisée dans la preuve des autres clauses.

1. *Respect des valuations des horloges communes* : Nous voulons ici prouver que

   $$((q_2,\zeta_2),(q_1,\zeta_1)) \in \mathcal{F}(\mathcal{Z}_{ds}) \Rightarrow \forall v_2 \cdot (v_2 \in \zeta_2 \Rightarrow \exists v_1 \cdot (v_1 \in \zeta_1 \wedge v_2\rfloor_{X_1} = v_1)).$$

   Par définition de $\mathcal{F}$, si $((q_2,\zeta_2),(q_1,\zeta_1)) \in \mathcal{F}(\mathcal{Z}_{ds})$, alors $\texttt{poly}(\zeta_2)\rfloor_{X_1} \subseteq \zeta_1$. La preuve est ici immédiate par définition de l'opérateur $\rfloor$ de projection avec restriction de dimension.

2. *Simulation stricte* : nous voulons ici prouver
   $((q_2,\zeta_2),(q_1,\zeta_1)) \in \mathcal{F}(\mathcal{Z}_{ds}) \Rightarrow \forall v_2 \cdot (v_2 \in \zeta_2 \wedge (q_2,v_2) \xrightarrow{e_2} (q_2',v_2') \wedge \texttt{label}(e_2) \in Labels_1 \Rightarrow$

   $$\exists v_1 \cdot (v_1 \in \zeta_1 \wedge (q_1,v_1) \xrightarrow{e_1} (q_1',v_1') \wedge \texttt{label}(e_1) \in Labels_1 \wedge (q_2',v_2')\mathcal{S}_{ds}(q_1',v_1'))).$$

   Considérons une transition $(q_2,v_2) \xrightarrow{e_2} (q_2',v_2')$ telle que $\texttt{label}(e_2) \in Labels_1$. Par construction du graphe de simulation, si une telle transition existe, alors une transition $(q_2,\zeta_2) \xrightarrow{e_2} (q_2',\zeta_2')$ existe dans $SG(A,c)$, telle que $v_2 \in \zeta_2$ et $v_2' \in \zeta_2'$. Comme $((q_2,\zeta_2),(q_1,\zeta_1)) \in \mathcal{F}(\mathcal{Z}_{ds})$, il existe une transition $(q_1,\zeta_1) \xrightarrow{e_1} (q_1',\zeta_1')$ telle que $\texttt{label}(e_1) = \texttt{label}(e_2)$ satisfaisant la clause 1 de la définition de $\mathcal{F}$. En particulier, nous avons donc $(q_2',\zeta_2')\mathcal{Z}_{ds}(q_1',\zeta_1')$. Comme cette transition existe, alors, par construction du graphe de simulation, il existe au moins une transition discrète (sémantique) inscrite dans cette transition.

D'autre part, nous avons déjà prouvé qu'il existe $v_1 \in \zeta_1$ tel que $v_2\rfloor_{X_1} = v_1$. Comme $\texttt{src\_val}((q_2,\zeta_2),e_2,(q'_2,\zeta'_2))\rfloor_{X_1} \subseteq \texttt{src\_val}((q_1,\zeta_1),e_1,(q'_1,\zeta'_1))$, alors il existe une transition discrète $(q_1,v_1) \xrightarrow{e_1} (q'_1,v'_1)$ telle que $v'_1 \in \zeta'_1$.
Il reste alors à prouver que $(q'_2,v'_2)\mathcal{S}_{ds}(q'_1,v'_1)$. L'action étiquetant e_2 est une action observable de A_2. On sait ainsi que la transition remet à zéro les mêmes horloges de X_1 que e_1, c'est à dire que $\texttt{reset}(e_2)\cap X_1 = \texttt{reset}(e_1)$. De plus, comme $v_2\rfloor_{X_1} = v_1$, et que par définition

$$v'_2 = [\texttt{reset}(e_2) := 0]v_2 \text{ et } v'_1 = [\texttt{reset}(e_1) := 0]v_1$$

alors on a $v'_2\rfloor_{X_1} = v'_1$. Comme $(q'_2,\zeta'_2)\mathcal{Z}_{ds}(q'_1,\zeta'_1)$, et que $(\star)$ est vrai pour $\mathcal{Z}_{ds}$, alors on a bien $(q'_2,v'_2)\mathcal{S}_{ds}(q'_1,v'_1)$.

3. *Egalité des délais* : on veut prouver que
$((q_2,\zeta_2),(q_1,\zeta_1)) \in \mathcal{F}(\mathcal{Z}_{ds}) \Rightarrow$
$\forall v_2 \cdot (v_2 \in \zeta_2 \wedge (q_2,v_2) \xrightarrow{t} (q_2,v'_2) \Rightarrow \exists v_1 \cdot (v_1 \in \zeta_1 \wedge (q_1,v_1) \xrightarrow{t} (q_1,v'_1) \wedge (q_2,v'_2)\mathcal{S}_{ds}(q_1,v'_1))).$

   Les transitions de temps qui apparaissent dans un graphe sémantique n'apparaissent plus explicitement en tant que transition dans le graphe de simulation associé. Intuitivement, on peut dire que le temps s'écoule à l'intérieur des zones. Considérons une transition $(q_2,v_2) \xrightarrow{t} (q_2,v'_2)$ dans la zone (q_2,ζ_2) (c'est à dire $v_2 \in \zeta_2$ et $v'_2 \in \zeta_2$). Considérons également la valuation $v_1 \in \zeta_1$ telle que $v_2\rfloor_{X_1} = v_1$ (cette valuation existe par la clause *respect des valuations des horloges communes*). Comme $\zeta_2\rfloor_{X_1} \subseteq \zeta_1$, il existe une transition $(q_1,v_1) \xrightarrow{t} (q_1,v'_1)$ telle que $v'_1 \in \zeta_1$ et $v'_1 = v'_2\rfloor_{X_1}$. L'hypothèse d'induction permet de dire qu'il existe une valuation v''_1 telle que $(q_2,v'_2)\mathcal{S}_{ds}(q_1,v''_1)$ telle que $v''_1 = v'_2\rfloor_{X_1}$ et donc $v''_1 = v'_1$. Il suit donc qu'on a bien $(q_2,v'_2)\mathcal{S}_{ds}(q_1,v'_1)$.

4. *Bégaiement des τ-transitions* : nous voulons ici prouver
$((q_2,\zeta_2),(q_1,\zeta_1)) \in \mathcal{F}(\mathcal{Z}_{ds}) \Rightarrow$

   $\forall v_2 \cdot (v_2 \in \zeta_2 \wedge (q_2,v_2) \xrightarrow{e_2} (q'_2,v'_2) \wedge \texttt{label}(e_2) = \tau \Rightarrow \exists v_1 \cdot (v_1 \in \zeta_1 \wedge (q'_2,v'_2)\mathcal{S}_{ds}(q_1,v_1))).$

   Considérons une transition $(q_2,v_2) \xrightarrow{e_2} (q'_2,v'_2)$ telle que $\texttt{label}(e_2) = \tau$. Par construction du graphe de simulation, une transition $(q_2,\zeta_2) \xrightarrow{e_2} (q'_2,\zeta'_2)$ existe dans $SG(A,c)$, telle que $v_2 \in \zeta_2$ et $v'_2 \in \zeta'_2$. Comme $((q_2,\zeta_2),(q_1,\zeta_1)) \in \mathcal{F}(\mathcal{Z}_{ds})$, alors par définition de $\mathcal{F}$, on a $(q'_2,\zeta'_2)\mathcal{Z}_{ds}(q_1,\zeta_1)$. On a déjà prouvé qu'il existe une valuation $v_1 \in \zeta_1$ telle que $v_1 = v_2\rfloor_{X_1}$. Il reste à prouver que $v'_2\rfloor_{X_1} = v_1$. Comme les τ-transitions ne remettent à zéro que les horloges de $X_2\backslash X_1$, seules les valuations des horloges de X_2 sont modifiées dans v'_2 par rapport à v_2. On a donc bien $v'_2\rfloor_{X_1} = v_1$.
   Comme $(q'_2,\zeta'_2)\mathcal{Z}_{ds}(q_1,\zeta_1)$, et que $(\star)$ est vrai pour $\mathcal{Z}_{ds}$, alors on a bien $(q'_2,v'_2)\mathcal{S}_{ds}(q_1,v_1)$.

5. *Respect du décor des états* : immédiat par définition de $\mathcal{F}$.
6. *Sensibilité à la divergence* : immédiat par le lemme 7.

7. *Respect de la stabilité* : on veut ici prouver que

$$\begin{array}{c}\text{si } (\zeta_2 \backslash \texttt{free}(q_2))\rfloor_{X_1} \subseteq \zeta_1 \backslash \texttt{free}(q_1) \\ \text{alors } \forall v_2 \cdot (v_2 \in \zeta_2 \wedge v_2 \notin \texttt{free}(q_2) \Rightarrow \exists v_1 \cdot (v_1 \in \zeta_1 \wedge v_1 \notin \texttt{free}(q_1))).\end{array}$$

De manière équivalente, on a :

$$\begin{array}{c}\text{si } \forall v \cdot (v \in (\zeta_2 \backslash \texttt{free}(q_2))\rfloor_{X_1} \Rightarrow \exists v' \cdot (v' \in \zeta_1 \backslash \texttt{free}(q_1) \wedge v' = v)) \\ \text{alors } \forall v_2 \cdot (v_2 \notin \texttt{free}(q_2) \Rightarrow \exists v_1 \cdot (v_1 \notin \texttt{free}(q_1) \wedge v_2\rfloor_{X_1} = v_1)).\end{array}$$

Par conséquent, on a immédiatement $(\star)$ est vrai pour $\mathcal{F}(\mathcal{Z}_{ds})$.

□

THÉORÈME 5 *Considérons deux automates temporisés A_1 et A_2, et leurs graphes de simulation respectifs SG_1 et SG_2. Si $SG_2 \preceq_{\mathcal{Z}_{ds}} SG_1$ alors $A_2 \preceq_{\mathcal{S}_{ds}} A_1$.*

PREUVE. Comme $SG_2 \preceq_{\mathcal{Z}_{ds}} SG_1$ alors $z_{0_2}\ \mathcal{Z}_{ds}\ z_{0_1}$. Considérons les configurations initiales s_{0_2} et s_{0_1} de A_2 et A_1. Par définition, $s_{0_2} \in z_{0_2}$ et $s_{0_1} \in z_{0_1}$. Par le lemme 8, on sait que s_{0_2} est en relation (w.r.t. $\mathcal{S}_{ds}$) avec une unique configuration s dans z_{0_1}. Par définition de $\mathcal{S}_{ds}$, les valuations sur X_1 de s et s_{0_2} sont égales. Il suit donc $s = s_{0_1}$ et que $s_{0_2}\mathcal{S}_{ds}s_{0_1}$. □

7.3 Vers l'algorithme

Nous détaillons dans cette section l'algorithme permettant de vérifier la τ-simulation temporisée DS symbolique. On considère pour cela deux graphes de simulation SG_2 et SG_1 et on vérifie que $SG_2 \preceq_{\mathcal{Z}_{ds}} SG_1$. Cette vérification est effectuée en deux temps :

1. la première phase consiste à détecter les τ-cycles non zénon,
2. la seconde phase consiste à vérifier la τ-simulation temporisée symbolique et le respect de la stabilité.

7.3.1 Détecter les τ-cycles non zénon

Pour cette détection, nous utilisons l'algorithme *full DFS* (*full Depth First Search*) défini dans [Tri98, TYB05]. Cet algorithme a été initialement mis en place pour tester le vide d'un automate de Büchi temporisé, dans le cas d'une condition d'acceptation *persistante*, où à partir d'un point, l'automate ne visite plus que des états d'acceptation. L'algorithme consiste à détecter les cycles non zénon d'un automate ne contenant que des états d'acceptation. Pour cela, il explore tous les chemins d'un graphe de simulation et les stocke dans une pile. L'exploration d'un chemin s'arrête dès lors que l'on visite un état déjà présent dans la pile, signifiant qu'un cycle élémentaire a été détecté. Il ne reste alors plus qu'à tester si ce cycle est non zénon et ne contient que des états d'acceptation.

L'algorithme 7.1 présente l'adaptation que nous en avons faite pour détecter les τ-cycles non zénon. Dès qu'un cycle est détecté, nous testons s'il est non zénon et si toutes les transitions de ce cycle sont étiquetées par τ. Les procédures `Sommet`, `Empiler` et `Depiler` représentent les opérations classiques sur les piles, permettant de récupérer le

sommet d'une pile, d'ajouter ou de supprimer un élément dans la pile. La procédure `Partie(Pile, e)` permet de récupérer tous les éléments de la pile `Pile` empilés à partir de l'élément `e`. La procédure `Suivant(Pile, e)` permet de récupérer l'élément qui suit `e` dans la pile `Pile` (c'est-à-dire l'élément empilé après `e`). La procédure `non_zenon` est définie comme dans [Tri98] et effectue un test syntaxique pour tester le non-zénonisme d'un chemin. Ce test consiste à vérifier que, dans le cycle, il existe une horloge x qui est remise à zéro à un point i du cycle, et que x possède une borne inférieure à un point j du cycle. Cela permet d'assurer qu'au moins une unité de temps s'écoule à chaque passage dans le cycle.

ALGORITHME 7.1. UNE *full-DFS* POUR DÉTECTER LES τ-CYCLES NON ZÉNON

```
sensibilite_divergence(SG){
   Pile := {z0}
   retourne τ_cycles_non_zenon()
}

τ_cycles_non_zenon(){
   z := Sommet(Pile)
   cycle := faux
   tant que ∃ z -e-> z' et cycle = faux
      si z' ∉ Pile alors
         Empiler(z', Pile)
         cycle := τ_cycles_non_zenon()
         Depiler(Pile)
      sinon
         si ∀z1 ∈ Partie(Pile, z'), ∃z1 -τ-> Suivant(Pile, z1)
                              et non_zenon(Partie(Pile, z')) alors
            retourne vrai
   fin tant que

   retourne cycle
}
```

7.3.2 Vérifier la τ-simulation temporisée DS symbolique

L'algorithme 7.2 permet de vérifier la τ-simulation temporisée DS symbolique entre deux graphes de simulation SG_1 et SG_2, étant donné un prédicat de collage P_c. Formellement, il vérifie $SG_2 \preceq_{\mathcal{Z}_{ds}} SG_1$. Cette vérification est en $\mathcal{O}((|Z_1| + |\mathcal{E}_1|) \times (|Z_2| + |\mathcal{E}_2|))$, où Z_i et $\mathcal{E}_i$, pour $i = 1, 2$, sont les ensembles d'états et de transitions de chaque SG_i. L'algorithme est découpé en quatre parties, dont la principale est `verification_`$\mathcal{Z}_{\mathtt{ds}}$.

Pour cela, une procédure `verif_`$\mathcal{Z}$`_et_respect_stabilite` effectue un parcours en profondeur d'abord de SG_2 et SG_1 de manière conjointe, et à chaque étape, vérifie les clauses de la définition 13. Un ensemble `Visites` permet de stocker les couples d'états en relation déjà parcourus, et une pile `Pile` permet de stocker les couples d'états qui sont

en cours de vérification. Cette pile permet également de retourner des diagnostics dans le cas où la relation n'est pas vérifiée.

Les algorithmes `condition_simulation_stricte` et `respect_stab` ne sont en réalité présents qu'en tant que raccourci pour alléger l'écriture de l'algorithme précédent `verif_`$\mathcal{Z}$`_et_respect_stabilite`. Ils ne sont respectivement qu'une réécriture algorithmique de la condition de la clause 1 (simulation stricte) et de la clause 6 (respect de la stabilité) de la définition 13 de la τ-simulation temporisée DS symbolique.

7.4 Conclusion

Dans ce chapitre, nous avons présenté une version symbolique des τ-simulations temporisées, jusqu'alors définies à un niveau sémantique. Nous avons montré que, lorsque ces relations symboliques sont établies, elles bénéficient des propriétés des relations sémantiques correspondantes, que ce soit au niveau de la préservation de propriétés (MITL, absence de blocage et non-zénonisme), ou de la compatibilité et de la compositionnalité avec les opérateurs de composition $||$ et $|$.

Dans la partie suivante, nous mettons en œuvre l'utilisation de la τ-simulation temporisée DS symbolique, et de ses propriétés, pour étudier son intérêt en pratique. Nous commençons dans le chapitre suivant par présenter l'outil que nous avons développé et qui implante notamment la vérification de cette relation, dans le cadre d'une modélisation incrémentale par intégration de composants.

ALGORITHME 7.2. VÉRIFICATION DE LA τ-SIMULATION TEMPORISÉE DS SYMBOLIQUE

```
verification_Z_ds(SG2, SG1, Pc){
   si (sensibilite_divergence(SG2)) alors
      retourner faux
   sinon
      Pile := {(z0_2, z0_1)}
      Visites := ∅
      retourne verif_Z_et_respect_stabilite()
}

verif_Z_et_respect_stabilite(){
   simul_ok := vrai
   (z2, z1) := sommet(Pile)
   si poly(z2)⌋X1 ⊆ poly(z1) ∧ (disc(z2), disc(z1)) ⊨ Pc ∧ respect_stab(z2, z1) alors
      tant que ∃ une transition z2 -e2-> z2' de SG2 et que simul_ok = vrai
         si label(e2) ∈ Labels1 alors
            si ∃z1 -e1-> z1' t.q. label(e1) = label(e2) ∧
                  condition_simulation_stricte(z1, e1, z1', z2, e2, z2') = vrai alors
               si (z2', z1') ∉ Visites et (z2', z1') ∉ Pile
                  Empiler((z2', z1'), Pile)
                  simul_ok := verif_Z_et_respect_stabilite()
                  Depiler(Pile)
            sinon
               retourner faux
         sinon
            si (z2', z1') ∉ Visites et (z2', z1') ∉ Pile
               Empiler((z2', z1), Pile)
               simul_ok := verif_Z_et_respect_stabilite()
               Depiler(Pile)
      fin tant que
   sinon
      retourner faux

   si simul_ok = vrai alors Visites := Visites ∪ {(z2, z1)}
   retourner simul_ok
}

condition_simulation_stricte (z1, e1, z1', z2, e2, z2'){
   si (src_val(z2, e2, z2')⌋X1 ⊆ src_val(z1, e1, z1')) alors
      retourner vrai
   sinon
      retourner faux
}

respect_stab (z2, z1){
   si (poly(z2)\free(disc(z2)))⌋X1 ⊆ poly(z1)\free(disc(z1)) alors
      retourner vrai
   sinon
      retourner faux
}
```

8

L'outil VeSTA

Sommaire

Dans ce document, nous avons proposé d'utiliser une méthode alternative pour vérifier les propriétés de systèmes temporisés à composants. Cette méthode est basée sur la préservation des propriétés durant une modélisation incrémentale de ces systèmes, plutôt que sur une vérification directe sur le modèle complet.
Pour les propriétés locales des composants, cette méthode permet de ne les vérifier que sur ces composants, plutôt que sur le modèle complet, puis de garantir qu'une intégration de ces composants dans un système préserve ces propriétés. Pour les propriétés globales, une possibilité est d'utiliser une modélisation par raffinement. Il s'agit de considérer un modèle abstrait pour chaque composant. Une vue complète du modèle peut ensuite être obtenue par composition de ces composants abstraits, et les propriétés globales du système peuvent être vérifiées sur cette abstraction. Chaque composant est alors raffiné, et le modèle concret (raffiné) complet du système est obtenu par composition des versions raffinées de chaque composant. La préservation des propriétés est alors garantie par le raffinement du modèle complet, qui lui-même doit être établi comme une conséquence du raffinement de chaque composant (propriété de compositionnalité du raffinement).
Dans le chapitre 5, nous avons montré que des relations de simulation temporisées peuvent être utilisées afin de garantir la préservation de propriétés linéaires, exprimées en MITL. Puis, dans le chapitre 6, nous avons établi le lien entre ces simulations et les méthodes de développement incrémental, en établissant comme bilan :

1. L'opérateur de composition | est adapté au développement incrémental formalisé par les τ-simulations définies, pour la préservation de toute propriété exprimée en MITL, mais également pour la préservation du non-zénonisme et des propriétés d'absence de blocage (sous les conditions données dans le chapitre 6),

2. L'opérateur || n'est adapté que lorsque seules les propriétés de sûreté doivent être préservées.

Pour montrer l'intérêt de cette méthode par préservation en pratique, nous souhaitons la comparer avec les méthodes classiques de vérification, s'appliquant directement sur le modèle global. En particulier, nous nous concentrons dès à présent sur un développement incrémental par intégration de composants. Nous nous intéressons à la préservation des propriétés locales des composants par intégration de composants. Dans le premier cas, avec l'opérateur *à la CCS* |, le gain en pratique est immédiat, car toutes les propriétés linéaires locales aux composants sont préservées automatiquement. Ainsi, les propriétés locales des composants peuvent être vérifiées sur ces composants, de taille plus petite que le modèle complet, où la vérification par model-checking souffrira moins de l'explosion combinatoire induite par le produit des composants. Dans le second cas, seules les propriétés de sûreté sont préservées automatiquement par l'opérateur *à la CSP* ||. Pour garantir la préservation d'autres types de propriétés, telles que les propriétés de vivacité et de réponse bornée, la τ-simulation temporisée DS doit être vérifiée.

Pour cela, nous avons développé l'outil VeSTA (**Ve**rification of **S**imulations for **T**imed **A**utomata). L'objet de ce chapitre est de présenter cet outil. Nous commencerons par une présentation générale dans la section 8.1. Puis, nous décrirons son architecture dans la section 8.2. Enfin, la section 8.3 expliquera comment utiliser l'outil.

8.1 Présentation générale

VeSTA considère des systèmes temporisés modélisés à base de composants, et se focalise sur une modélisation incrémentale de ces systèmes effectuée par intégration de composants. La fonctionnalité principale de VeSTA est de garantir que l'intégration d'un composant C dans un système S (par l'opérateur de composition *à la CSP* ||), préserve les propriétés MITL établies du composant C. Pour assurer cette préservation, VeSTA doit vérifier que le composant C simule son intégration dans S, notée $C||S$, par rapport à la τ-simulation temporisée DS $\mathcal{Z}_{ds}$. Formellement, VeSTA vérifie $C||S \preceq_{\mathcal{Z}_{ds}} C$.

VeSTA offre deux fonctionnalités additionnelles. Tout d'abord, il donne la possibilité de connecter le modèle (ou des assemblages de composants du modèle) à la plate-forme de vérification OPEN-CAESAR [Gar98], afin de pouvoir utiliser ses modules de vérification. Une autre possibilité de VeSTA est de spécifier les propriétés de C qui doivent être préservées, et de n'effectuer qu'une vérification *partielle* de la simulation. Ceci est expliqué en détail dans la remarque suivante.

REMARQUE 8.1. VÉRIFICATION PARTIELLE DE LA SIMULATION. L'objectif d'une telle vérification est de ne garantir la préservation que des propriétés spécifiées, plutôt que la préservation de toutes les propriétés pouvant être exprimées. Cette fonctionnalité n'est pour l'instant disponible que pour des propriétés de réponse du type $\Box(p \Rightarrow \Diamond q)$.

Les motivations de cette vérification partielle sont les suivantes. Il est possible que la vérification de la simulation échoue et ceci est dû, en général, à l'introduction d'un nouveau blocage (non-satisfaction de la clause *respect de la stabilité*). Il est donc impossible dans ce cas de conclure que les propriétés de C, quelles qu'elles soient, sont préservées. Cependant, si la préservation n'est pas garantie de manière générale, il se peut que les propriétés spécifiques de C soient quant à elles préservées. En effet, un chemin π de $C||S$, dans lequel un blocage est introduit par rapport au chemin de C qui le simule, fait échouer la vérification de la préservation. Mais, si ce chemin ne concerne pas les propriétés de C à préserver, alors cette préservation devrait pouvoir être garantie. C'est pour cette raison que la vérification partielle a été mise en oeuvre.
Détaillons à présent en quoi une telle vérification consiste pour une propriété de réponse de type $\Box(p \Rightarrow \Diamond q)$. Pour s'assurer de la préservation d'une telle propriété, il faut garantir que, dès lors que p est rencontré dans un chemin π, ce chemin n'est pas *coupé* avant que q soit rencontré, par l'introduction d'un blocage. Ainsi, une vérification partielle consiste à ne vérifier cette non-introduction de blocage (c'est-à-dire le respect de la stabilité) que pour les états de π entre l'état satisfaisant p et celui satisfaisant q. La Fig. 8.1 illustre le principe de cette vérification partielle pour ce type de propriété de réponse. Un chemin est représenté et les états sur lesquels est vérifié le respect de la stabilité sont grisés. Notons que la sensibilité à la divergence doit également être vérifiée, pour assurer qu'une coupure dans le chemin n'aura pas lieu par le biais d'une "introduction infinie" d'actions non observables.

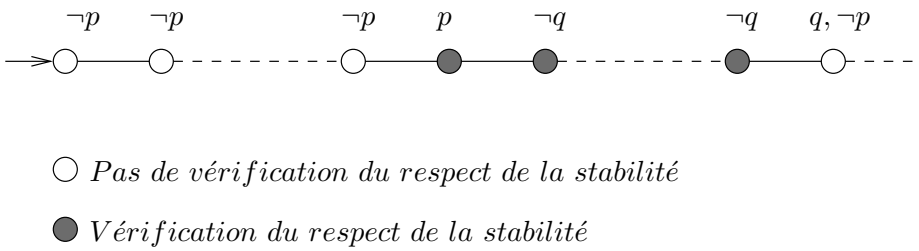


FIG. 8.1 – Vérification partielle pour une propriété de type $\Box(p \Rightarrow \Diamond q)$

Ainsi, VESTA offre la possibilité de spécifier les propriétés de C, de type $\Box(p \Rightarrow \Diamond q)$, qui doivent être préservées, et permet de vérifier la simulation uniquement pour garantir la préservation de ces propriétés. Remarquons que le non-zénonisme est toujours préservé par ce type de vérification, mais que ce n'est trivialement plus le cas des propriétés d'absence de blocages.

8.2 Architecture

VESTA a été développé à la fois en Java (pour la partie interface homme-machine) et en C (pour les modules concernant la vérification de la simulation). La figure 8.2 présente l'architecture de VESTA. Notons tout d'abord que nous avons adopté une architecture sur

le modèle de celle de l'outil OPEN-KRONOS[15] [Tri98]. OPEN-KRONOS permet notamment de tester si un état est atteignable dans un automate temporisé, ou encore de tester le vide d'un automate de Büchi temporisé, grâce au module PROFOUNDER [TYB05]. OPEN-KRONOS utilise la bibliothèque SMI[16] (Symbolic Model Interface), qui fournit une représentation symbolique basée sur les diagrammes de décision pour des modèles à nombre d'états fini décrits par des réseaux d'automates étendus. Un automate temporisé étendu peut utiliser, en plus des horloges, des variables entières, booléennes ou encore de type énuméré. OPEN-KRONOS considère donc des réseaux d'automates temporisés étendus et crée un fichier permettant de générer à la volée le graphe de simulation correspondant à l'automate temporisé étendu du modèle complet. Ce fichier est créé de telle manière qu'il peut être connecté à la plate-forme OPEN-CAESAR afin d'utiliser les différents modules de vérification qu'elle propose.

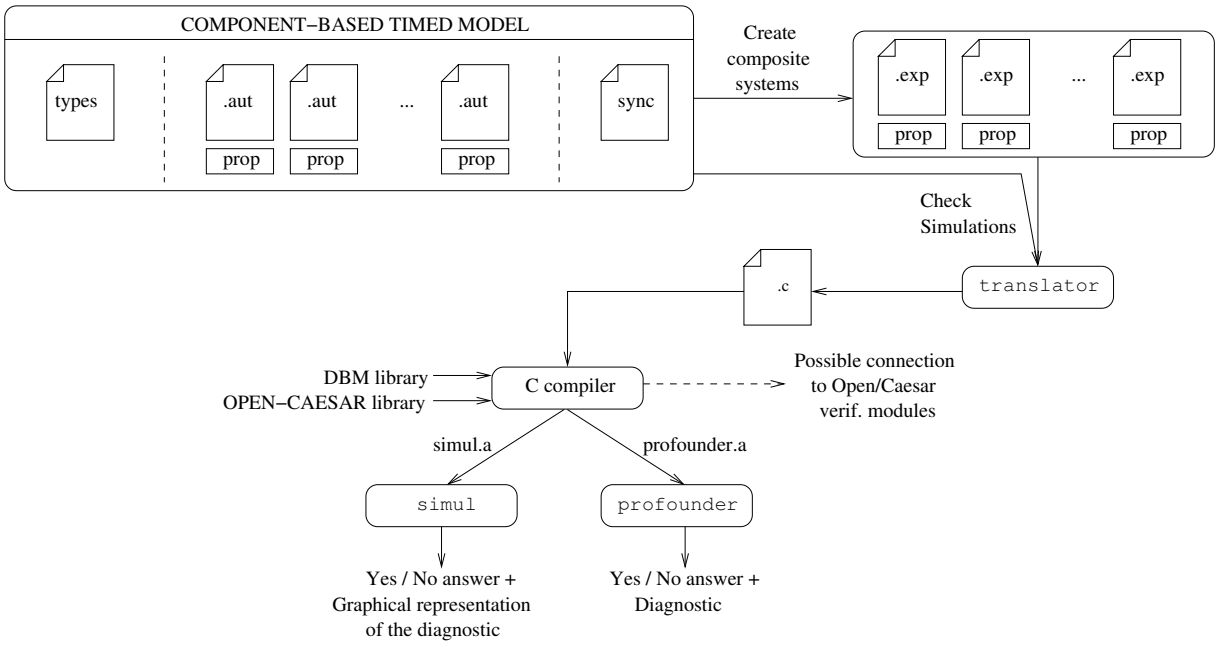


FIG. 8.2 – Architecture de VeSTA

Nous avons adopté la même architecture pour développer VeSTA. VeSTA considère des modèles temporisés à base de composants, où chaque composant est décrit par un automate temporisé étendu. Les modèles pris en compte comportent les éléments suivants : l'ensemble des composants (*.aut*), leurs synchronisations (*sync*) et les types de variables utilisés (*types*). A partir de ce modèle, les assemblages de composants peuvent être automatiquement créés (*.exp*), en utilisant l'opérateur de composition parallèle classique ||, qui est celui dont nous disposons grâce à la bibliothèque SMI et sur lequel nous nous focalisons particulièrement. Dans l'objectif d'une vérification partielle, il est également

[15] `http://www-verimag.imag.fr/~tripakis/openkronos.html`
[16] `http://www-verimag.imag.fr/~async/SMI/`

possible de spécifier les propriétés locales aux composants ou aux assemblages de composants qui doivent être préservées par intégration (*prop*).

La partie centrale de l'outil est celle qui permet de vérifier qu'un composant simule son intégration dans un système, en appliquant l'algorithme 7.2. Cette partie prend en entrée deux automates temporisés étendus[17], l'un correspondant au composant[18] et l'autre au modèle obtenu après intégration du composant dans le système. Cette partie principale est constituée de trois modules : `translator`, `simul` et PROFOUNDER :

- `translator` traduit les deux automates temporisés étendus en un fichier .c qui contient toutes les structures de données et fonctions permettant de générer le graphe de simulation correspondant à chacun des automates. De plus, les fonctions et structures de données sont nommées de façon à pouvoir connecter ce fichier à la plate-forme OPEN-CAESAR (plus exactement, c'est le modèle correspondant au composant plongé dans le système qui pourra être connecté aux modules d'OPEN-CAESAR). Le fichier créé est ensuite compilé et lié à la bibliothèque DBM et aux bibliothèques d'OPEN-CAESAR. Les DBM (*Difference-Bound Matrix*) sont une structure de données permettant de représenter des polyhèdres convexes. La bibliothèque DBM utilisée fournit toutes les fonctions nécessaires à la manipulation des polyhèdres, qui seront utilisés pour construire le graphe de simulation. Notons que nous avons également ajouté quelques fonctionnalités à cette bibliothèque, en particulier des fonctions permettant de manipuler des polyhèdres non convexes, en utilisant des listes de DBM.

- `simul` est l'exécutable permettant de vérifier la *τ-simulation temporisée respectant la stabilité* entre les graphes de simulation représentés par le fichier créé par `translator`. Une option lors de la création de cet exécutable permet de vérifier la simulation de manière générale, c'est à dire pour assurer la préservation de toutes les propriétés linéaires d'un composant, ou de ne la vérifier que partiellement, pour les propriétés locales qui ont été spécifiées pour le composant. Cette seconde possibilité correspond à la vérification partielle de la simulation. Rappelons que seules les propriétés de réponse de type $\Box(p \Rightarrow \Diamond q)$ sont pour l'instant concernées par cette possibilité.

- PROFOUNDER est l'exécutable permettant de vérifier la partie *sensibilité à la divergence* de la τ-simulation. Le module `profounder.a` fait partie d'OPEN-KRONOS et permet de tester si un état est atteignable dans un automate temporisé ou de tester le vide d'un automate de Büchi temporisé. Nous en avons adapté une partie pour

[17] Des variables entières et booléennes peuvent être utilisées dans ces automates temporisés étendus. Lorsque ces variables sont utilisées comme des variables partagées, les résultats démontrés dans le chapitre 6 sur l'intégration, la compatibilité et la composabilité, ne sont plus nécessairement vrais. VESTA impose une restriction sur ces automates étendus, en interdisant l'utilisation de variables partagées. En revanche, des variables entières et booléennes locales aux automates, peuvent être utilisées sans problèmes.

[18] Notons que ce composant dont on souhaite vérifier l'intégration peut également être un assemblage de composants.

détecter les τ-cycles non zénon dans l'automate représentant l'intégration.

8.3 Utilisation de VeSTA

Nous présentons à présent comment utiliser VeSTA, pour modéliser des composants temporisés, puis pour s'assurer qu'un composant s'intègre correctement dans un système. Nous reprenons dans cette partie l'exemple du passage à niveau, introduit dans le chapitre 2 et considérons les propriétés P_2 et P_5 concernant les composants *barrière* et *contrôleur*. Nous montrons comment modéliser ce système avec VeSTA, et comment vérifier que P_2 et P_5 sont préservées par l'intégration des composants *train*.

8.3.1 Spécification des systèmes temporisés à composants

Comme nous l'avons dit dans la section précédente, VeSTA utilise la bibliothèque SMI pour bénéficier d'une représentation symbolique efficace des modèles traités. La syntaxe utilisée pour modéliser les composants, ainsi que les assemblages de composants, est celle définie par cette bibliothèque. Tout d'abord, rappelons que les composants sont décrits par des automates temporisés étendus. La figure 8.3 présente les automates temporisés étendus des composants *train*, *contrôleur* et *barrière* du passage à niveau.

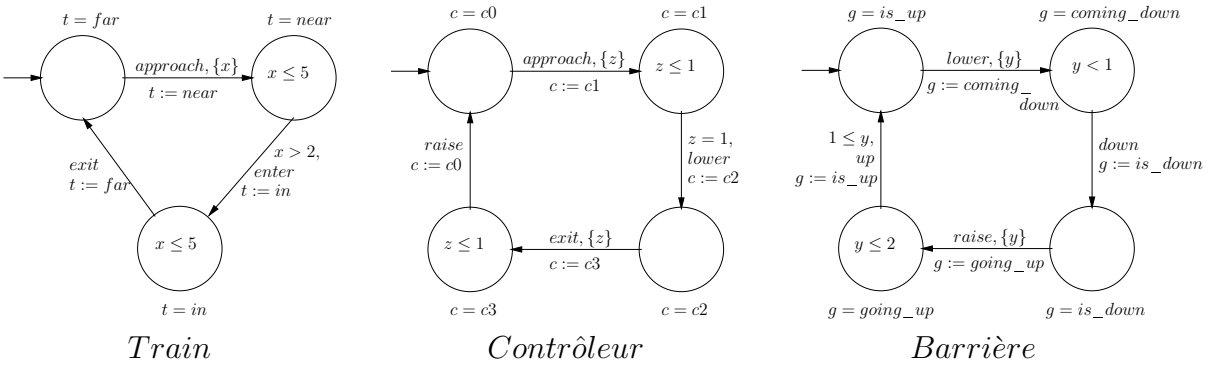


FIG. 8.3 – Automates temporisés étendus modélisant le train, le contrôleur et la barrière

La figure 8.4 présente la syntaxe utilisée pour décrire des automates dans VeSTA, sur l'exemple des trois automates *train*, *contrôleur* et *barrière*. Détaillons-la à présent sur l'exemple de l'automate du *train*. Avant de décrire l'automate lui-même, on commence par déclarer les variables qu'il utilise (`x` et `t`). La notation `des(0,3,3)` donne l'état initial de l'automate (0), son nombre de transitions (3) puis son nombre d'états (3). Chaque transition de l'automate est alors spécifiée par un état source, éventuellement une garde, un nom, éventuellement les variables qu'elle modifie, et enfin un état cible. Enfin, on modélise les invariants de chaque état de l'automate. Notons que les noms placés entre `(*` et `*)` représentent des commentaires dans la description de l'automate. Chaque automate

est sauvegardé dans un fichier d'extension `.aut`.

```
(* train.aut *)

x : clock
t : trainState

des(0, 3, 3)

(* far *)
(0, approach x := 0 t := near, 1)

(* near *)
(1, [x > 2] enter t := in, 2)

(* in *)
(2, exit t := far, 0)

(* Time − progress conditions *)
[0, t = far]
[1, t = near /\ x ≤ 5]
[2, t = in /\ x ≤ 5]
```

```
(* Controller.aut *)

z : clock
c : controllerState

des(0, 4, 4)

(* c0 *)
(0, approach z := 0 c := c1, 1)

(* c1 *)
(1, [z = 1] lower c := c2, 2)

(* c2 *)
(2, exit z := 0 c := c3, 3)

(* c3 *)
(3, raise c := c0 , 0)

(* Time − progress conditions *)
[0, c = c0]
[1, z ≤ 1 /\ c = c1]
[2, c = c2]
[3, z ≤ 1 /\ c = c3]
```

```
(* gate.aut *)

y : clock
g : gateState

des(0, 4, 4)

(* up *)
(0, lower y := 0 g := coming_down, 1)

(* comingDown *)
(1, down g := down, 2)

(* down *)
(2, raise y := 0 g := goingUp, 3)

(* goingUp *)
(3, [1 ≤ y] up g := up, 0)

(* Time − progress conditions *)
[0, g = up]
[1, g = comingDown /\ y < 1]
[2, g = down]
[3, g = goingUp /\ y ≤ 2]
```

FIG. 8.4 – Automates temporisés étendus dans VeSTA

Les types énumérés utilisés par les variables des automates sont définis à part et sauvegardés dans un fichier `.types`. Pour l'exemple du passage à niveau, les types suivants sont utilisés :

```
nat {5}
enum trainState {far,near,in}
enum controllerState {c0,c1,c2,c3}
enum gateState {up,comingDown,down,goingUp}
```

Le type `nat{5}` est déclaré dès lors que des horloges ou des variables entières sont utilisées dans un des automates. Le nombre donné en argument, ici 5, correspond à la plus grande valeur utilisée dans les contraintes ou affectations portant sur les variables de ce type.

La dernière étape dans la modélisation du système consiste à donner les synchronisations entre composants. VeSTA utilise l'opérateur de composition ||. Les composants se synchronisent donc sur les actions de même nom. Ces synchronisations sont spécifiées dans VeSTA de manière graphique, en quelque sorte sous la forme d'un graphe non orienté dont les états sont les composants, et les transitions représentent les synchronisations. Le graphe représentant les synchronisations du passage à niveau est donné par la figure 8.5 tel qu'il peut être visualisé dans VeSTA.

Les assemblages de composants sont décrits toujours en utilisant la syntaxe SMI, et

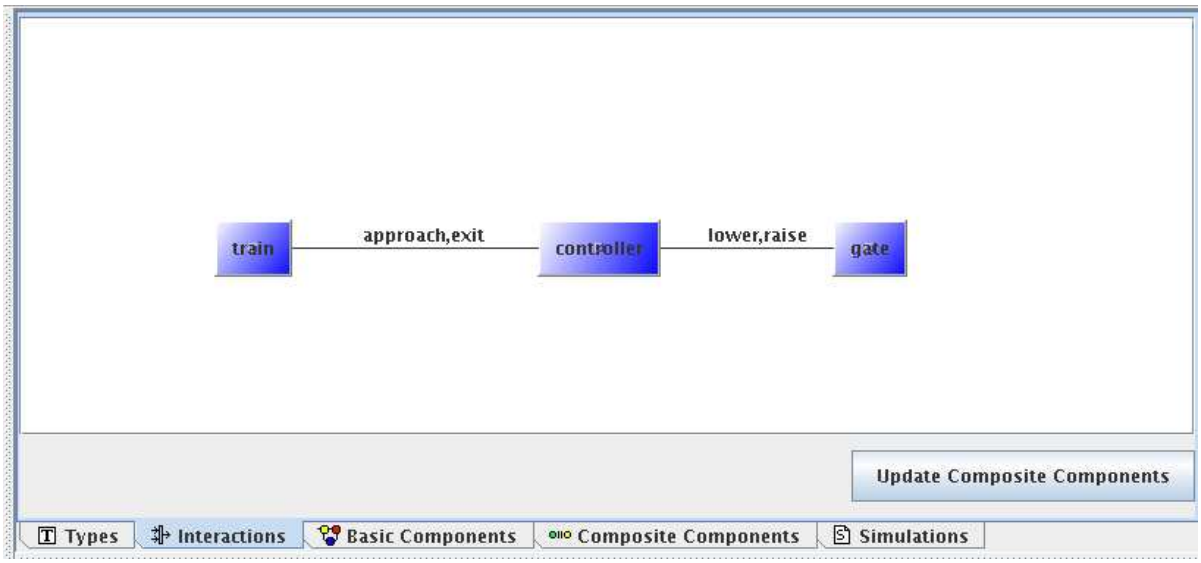


FIG. 8.5 – Synchronisations entre les composants du *passage à niveau*

sauvegardés dans des fichiers d'extension `.exp`. Pour simplifier la tâche d'un utilisateur, ces fichiers sont créés automatiquement par VeSTA à partir de composants choisis pour l'assemblage et des synchronisations spécifiées entre ces composants. Le modèle complet (avec un seul train), obtenu par assemblage des composants *train*, *contrôleur* et *barrière*, est décrit de la manière suivante :

```
Lotos-Behavior

train
|[approach,exit]|
controller
|[lower,raise]|
gate
```

Les noms `train`, `controller` et `gate` représentent des composants du système. Les notations `|[approach,exit]|` et `|[lower,raise]|` représentent les actions sur lesquelles se synchronisent les composants utilisés dans l'assemblage.

8.3.2 Vérification de l'intégration d'un composant dans un système

La fonctionnalité principale de VeSTA est de vérifier si un composant C simule son intégration dans un système S, notée $C||S$. Rappelons que C et $C||S$ sont décrits par des automates temporisés étendus. Le module `translator` crée un fichier contenant la description des graphes de simulation de C et $C||S$, puis le module `simul` vérifie la τ-simulation temporisée respectant la stabilité. L'algorithme utilisé est l'algorithme 7.2 présenté dans

le chapitre précédent et qui implante directement la définition symbolique 14 de la simulation. Il examine si les graphes de simulation de $C||S$ et C sont en relation, avec un prédicat de collage P_c défini par *true*[19]. Le module PROFOUNDER est utilisé pour vérifier la partie *sensibilité à la divergence* de la simulation. Le résultat de cette vérification est affiché dans l'interface de VeSTA. Si la vérification échoue, VeSTA fournit un diagnostic mettant en évidence une trace de $C||S$ qui n'est simulée par aucune trace de C (plus précisément, VeSTA affiche également la trace de C qui devait simuler cette trace de $C||S$).

EXEMPLE 8.1. Reprenons l'exemple du passage à niveau. Nous cherchons à nous assurer que les propriétés de l'assemblage `controller || gate` sont préservées par l'intégration d'un composant `train`. Pour cela, nous voulons vérifier que :

$$SG(\texttt{controller||gate||train}) \preceq_{\mathcal{Z}_{ds}} SG(\texttt{controller||gate})^{20}.$$

ce qui est effectivement le cas. Le résultat de cette vérification est affiché comme indiqué par la figure 8.6.

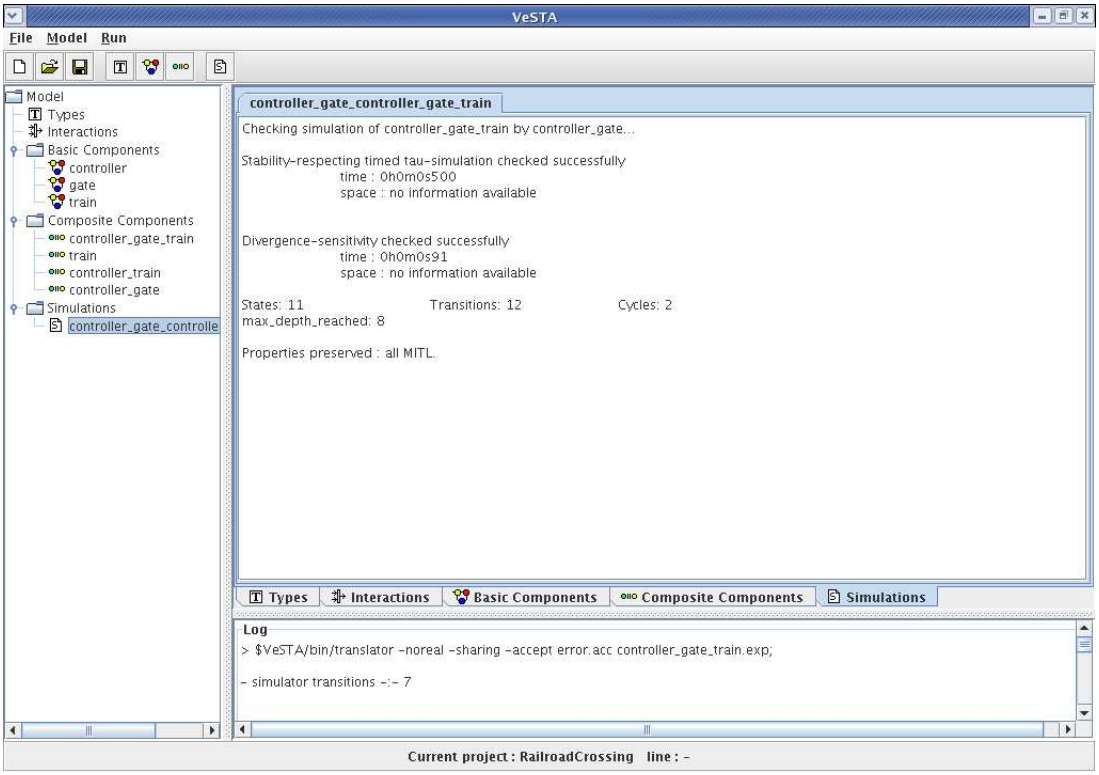


FIG. 8.6 – Résultat affiché par VeSTA lors d'une vérification réussie de la simulation

Lorsque la vérification échoue, un diagnostic apparait comme le montre la figure 8.7. Reprenons l'exemple du passage à niveau, et modifions l'invariant de l'état 2

[19]Rappelons que, dans le cadre d'une intégration de composants, le prédicat de collage entre deux automates temporisés est *true*.

[20]Jusqu'à présent, nous utilisions la notation $SG(A, c)$ pour dénoter le graphe de simulation d'un automate temporisé A, par rapport à un constante c. Pour plus de lisibilité, nous écrivons ici simplement $SG(A)$ et omettons la constante c, que nous supposons égale à la plus grande constante apparaissant dans une contrainte de A.

du train (t = in) de $x \leq 5$ à $x \leq 3$. Avec cette modification, l'intégration du composant *train* introduit un blocage et fait échouer la vérification de la simulation. En effet, l'état $(c2, is\_down, near)$ atteint dans le graphe de simulation de l'automate `controller || gate || train` contient des blocages pour $3 < z \leq 5$. La seule transition disponible depuis cet état est la transition *enter* du train, activable pour $2 < x \leq 3$ dans le composant *train* (puisque l'invariant de l'état cible est $x \leq 3$). Or, l'état source de la transition permet au temps de s'écouler tant que $x \leq 5$, les configurations de valuations $3 < x \leq 5$ sont des blocages. Comme $x = z$ dans l'état $(c2, is\_down, near)$, des blocages apparaissent pour les mêmes valeurs de z, c'est à dire pour $3 < z \leq 5$. Or, dans l'assemblage `controller || gate`, cet état ne possédait pas de blocages. En effet, son invariant est `true` et la transition disponible était la transition `exit` dont la garde est `true` et où l'horloge z est remise à zéro. Ainsi, le temps pouvait s'écouler sans borne supérieure et la transition était toujours activable. VeSTA affiche alors la trace de l'assemblage `controller || gate || train` dans laquelle apparaît un blocage qui n'existait pas avant l'intégration du composant `train`.

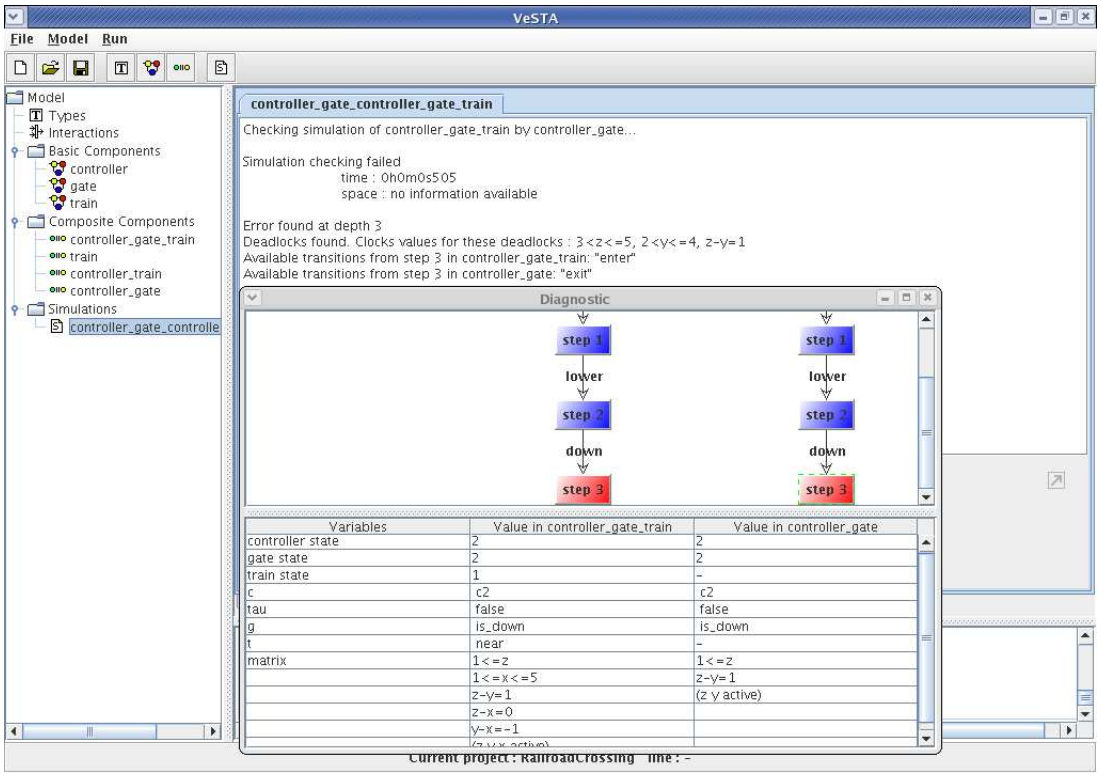


FIG. 8.7 – Diagnostic affiché par VeSTA lorsque la vérification de la simulation échoue

8.4 Conclusion

Dans ce chapitre, nous avons présenté le prototype VeSTA qui implante la vérification de la τ-simulation temporisée DS symbolique présentée au chapitre 7. A l'origine, VeSTA a été réalisé pour des systèmes temporisés modélisés incrémentalement par intégration de composants. Sa fonction première est de garantir qu'un composant s'intègre correctement dans un système, c'est-à-dire que les propriétés du composant sont préservées lors de cette intégration. Il vérifie pour cela la τ-simulation temporisée DS entre le composant intégré dans son système et le composant seul. Notons toutefois qu'il est possible de vérifier la τ-simulation temporisée DS entre des automates temporisés "quelconques" (c'est-à-dire dont l'un n'est pas un composant faisant partie de l'autre), en utilisant uniquement les modules `translator`, `simul` et PROFOUNDER sans passer par l'interface graphique de VeSTA.

VeSTA nous a permis d'appliquer la méthode de développement incrémental par intégration de composants à quelques systèmes, afin de se rendre compte de l'apport d'une telle démarche en pratique. Les résultats observés sont présentés dans le chapitre suivant.

9

Etudes de cas

Sommaire

Nous présentons dans ce chapitre les exemples sur lesquels nous avons appliqué la méthode proposée, dans le cadre de l'intégration de composants. Nous considérons des systèmes temporisés modélisés à base de composants, où les composants possèdent des propriétés locales à vérifier. Nous vérifions ces propriétés sur les composants concernés, puis vérifions qu'ils s'intègrent correctement dans leur environnement par le biais des relations de simulation définies dans la partie précédente. Nous comparons également notre approche avec la méthode classique de vérification, consistant à vérifier les propriétés sur le modèle global.
Rappelons que nous considérons que ces propriétés sont exprimées en MITL. A notre connaissance, il n'existe pas d'outil implantant la méthode de model-checking pour MITL. Nous considérons donc des propriétés MITL pouvant également être exprimées en TCTL, et utilisons l'outil KRONOS pour effectuer la vérification. Sauf indication contraire, la vérification des simulations est quant à elle effectuée à l'aide de VESTA.

9.1 La cellule de production

Cette cellule de production a été développée par FZI (le Centre de Recherche pour les Technologies de l'Information, à Karlsruhe), dans le cadre du projet Korso. L'objectif de cette étude de cas était d'étudier l'impact des méthodes formelles pour la modélisation et la vérification de processus industriels. Cette étude de cas a été traitée dans une trentaine de formalismes. Nous la modélisons ici comme elle l'a été dans [Bur03].

9.1.1 Description de la cellule

La cellule est composée de six éléments : deux tapis convoyeurs sur lesquels les pièces devant être traitées par la cellule arrivent et sont évacuées, un table élevatrice et rotative, un robot à deux bras, une presse, et un détecteur permettant de repérer l'arrivée des pièces. Un schéma représentant ce système est donné par la Fig. 9.1.

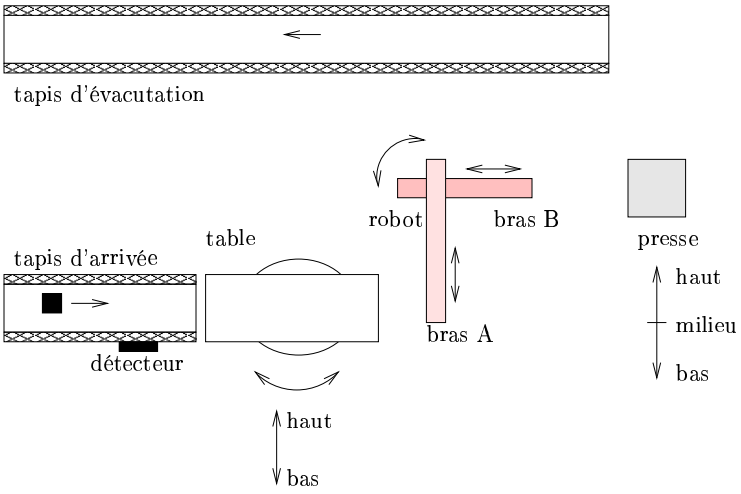


FIG. 9.1 – La cellule de production

Le fonctionnement global de la cellule de production est le suivant. Les pièces à traiter sont déposées sur le tapis d'arrivée. Le détecteur, placé sur ce tapis, détecte quand une pièce est introduite dans le dispositif, et envoie un signal au robot pour l'informer qu'une pièce est disponible. Lorsqu'elle arrive au bout du tapis, la pièce est transférée sur la table, qui monte et tourne pour se placer dans une position dans laquelle la pièce peut être récupérée par le robot. Pendant ce temps, le robot s'est placé en position pour que son bras A puisse prendre cette pièce. Le bras A prend donc la pièce et la transporte jusqu'à la presse. La presse attend que le robot ait fini de décharger cette pièce et retiré son bras. Elle peut ensuite traiter la pièce. Le robot, qui s'est déplacé pendant ce temps de façon à ce que le bras B soit en face de la presse, prend la pièce et la transporte jusqu'au tapis d'évacuation. La pièce est alors évacuée.
Cette cellule de production est soumise à des contraintes de temps données dans le tableau

9.1. Ces informations sont reprises de [Bur03].

Elément	Description de l'action	Durée (en u.t.)
Robot	tourne de 90°	15
Robot	déplacement de la table à la presse	5
Robot	déplacement de la presse au tapis d'évacuation	5
Robot	déplacement du tapis d'évacuation à la table	25
Robot	déplacement du tapis d'évacuation à la position d'attente	22
Robot	déplacement de la presse à la position d'attente	17
Robot	déplacement de la position d'attente à la table	3
Robot	déplacement de la position d'attente à la presse	2
Robot	en position d'attente	2
Tapis d'arrivée	déplacement de la pièce jusqu'au détecteur	3
Tapis d'arrivée	déplacement de la pièce du détecteur à la table	1
Table	élévation et rotation	2
Table	abaissement et rotation	2
Presse	traitement d'une pièce	22-25
Presse	disponible pour une nouvelle pièce	18-20
Tapis d'évacuation	évacuation d'une pièce	4

TAB. 9.1 – Contraintes de temps de la cellule de production

9.1.2 Modélisation de la cellule par des automates temporisés

Ce système est modélisé par au moins sept composants : un pour chaque élément de la cellule, ainsi qu'une ou plusieurs pièces. Chaque composant est modélisé par un automate temporisé. Le modèle complet est obtenu par composition parallèle (en utilisant l'opérateur ||) de tous les composants. Décrivons à présent chaque composant de manière détaillée.

Tapis d'arrivée (Fig. 9.2). Le tapis d'arrivée permet d'introduire les pièces dans la cellule de production. Initialement, il attend qu'une pièce arrive dans le dispositif. Quand il détecte l'arrivée d'une nouvelle pièce (`new_plate`), il la déplace jusqu'au détecteur en 3 u.t. (`arrive_to_sensor`), qui la détecte dès qu'elle passe devant lui (`sensor_detection`). La pièce continue à être acheminée vers la fin du tapis (`arrive_end_belt`). Ce déplacement prend une u.t. La pièce se trouve alors au bout du tapis (`plt`), et est transférée sur la table (`feedbelt_to_table`). Le tapis détecte alors que la pièce a bien été transférée ($\texttt{plt}_0$), et est de nouveau prêt pour recevoir une nouvelle pièce. Notons que nous avons repris ici exactement la description donnée dans [Bur03], où il est considéré qu'une seule pièce à la fois peut se trouver sur le tapis d'arrivée.

Tapis d'évacuation (Fig. 9.3). Le tapis d'évacuation permet de faire sortir les pièces qui ont été traitées par la presse de la cellule. Dès qu'une pièce est déchargée sur ce tapis par le robot (`robot_to_db`), le tapis l'achemine vers la sortie du dispositif en quatre u.t. (`move`). Dès qu'il a détecté qu'il avait évacué la pièce (`final`), il est de nouveau prêt à

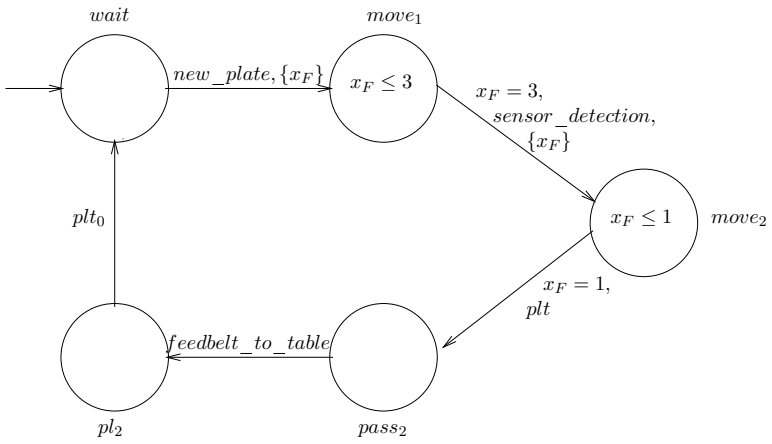


FIG. 9.2 – Automate temporisé du tapis d'arrivée

recevoir une nouvelle pièce. Tout comme pour le tapis d'arrivée, on considère que le tapis d'évacuation ne peut transporter qu'une seule pièce à la fois.

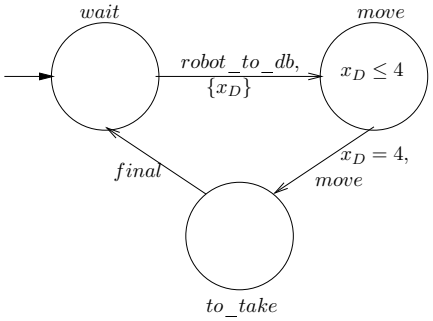


FIG. 9.3 – Automate temporisé du tapis d'évacuation

Table (Fig. 9.4). Lorsqu'une pièce est déposée sur la table (`feedbelt_to_table`), celle-ci monte et tourne en deux u.t. pour être dans une position dans laquelle le robot pourra récupérer la pièce (`move_twist`). La pièce est ensuite saisie par le robot (`table_to_robot`) et la table détecte qu'elle ne contient plus de pièce ($\texttt{plt}_1$). Elle met alors deux u.t. pour revenir à sa position initiale et est disponible pour transporter une nouvelle pièce (`table_move_back`).

Presse (Fig. 9.5). Lorsqu'une pièce est déposée sur la presse par le robot (`press_get_plate`), la presse la détecte ($\texttt{plt}_0$) et met entre 22 et 25 u.t. pour la traiter (`process_plate`). La pièce est ensuite disponible pour être récupérée par le robot (`plate_taken_by_robot`). La presse détecte qu'elle ne contient plus de pièce ($\texttt{plt}_3$) et met alors entre 12 et 20 u.t. pour revenir à sa position initiale pour accueillir une nouvelle pièce (`press_move_back`).

Détecteur (Fig. 9.6). Le détecteur consiste uniquement à détecter le nombre de pièces

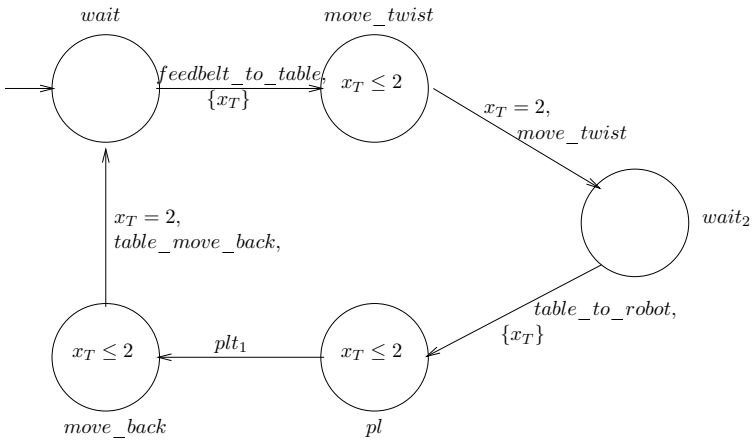


FIG. 9.4 – Automate temporisé de la table

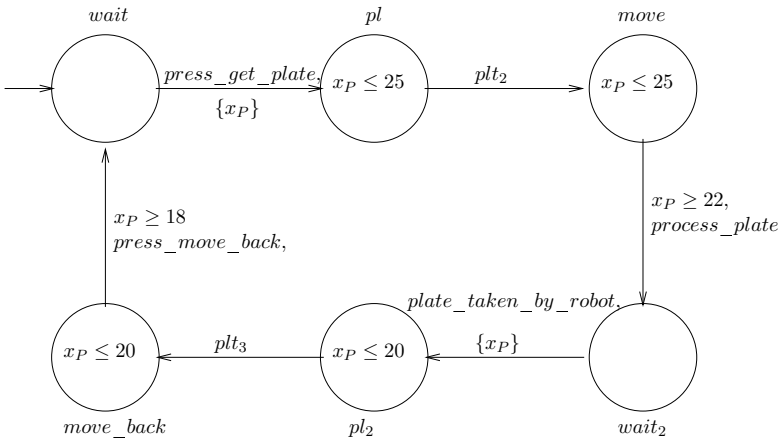


FIG. 9.5 – Automate temporisé de la presse

présentes dans la cellule. Quand une pièce est détectée sur le tapis (`sensor_detection`), le nombre de pièces est incrémenté, tandis qu'il est décrémenté lorsqu'une pièce est prise par le robot (`read`).

Robot (Fig. 9.7). Rappelons que le robot possède deux bras : le bras A permet de transporter les pièces de la table vers la presse, tandis que le bras B permet de les déplacer de la presse au tapis d'évacuation. Détaillons à présent le fonctionnement de ce composant. Lorsqu'une pièce est introduite dans la cellule, le robot reçoit un signal du détecteur. Dès que la pièce est disponible sur la table, le robot la prend avec le bras A et tourne pour placer le bras B en face de la presse. Si une pièce est déjà présente sur la presse, le bras B la prend, sinon le robot continue à tourner pour placer le bras A en face de la presse et déposer la pièce qu'il vient de saisir sur la table. Puis, si le bras B est chargé, le robot tourne jusqu'au tapis d'évacuation pour décharger la pièce que le bras B contient (puis revient vers la table), sinon il se place dans une position intermédiaire, appelée position

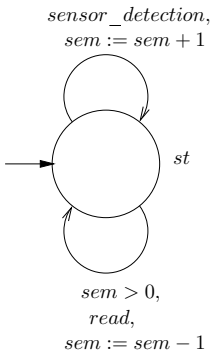


FIG. 9.6 – Automate temporisé du détecteur

d'attente. De cette position, il pourra alors soit retourner vers la table si une nouvelle pièce arrive, soit aller à la presse pour évacuer une pièce qui a été traitée.

Pièce (Fig. 9.8). Les actions apparaissant dans l'automate temporisé de la pièce correspondent à la progression de la pièce dans la cellule. Toutes ces actions ont été commentées lors de la description des composants de la cellule. Comme dans [Bur03], nous considérons qu'une pièce est un composant *passif* du système, qui est transportée d'un composant de la cellule à un autre, et dont le déplacement dépend de ces autres composants. Pour cette raison, l'automate la représentant ne possède aucune contrainte de temps.

9.1.3 Propriétés à vérifier

Rappelons que nous nous intéressons aux propriétés locales des composants ou des groupes de composants. Nous souhaitons vérifier ce type de propriétés localement puis assurer leur préservation au moyen des τ-simulations temporisées, plutôt que de les vérifier directement sur le modèle global. Nous avons identifié quelques propriétés locales portant sur le robot. Les propriétés P_1 et P_2 sont des propriétés de sûreté, P_3 et P_4 sont des propriétés de vivacité, et P_5 à P_7 sont des propriétés de réponse bornée :

(P_1) Quand le robot est en position d'attente, ses deux bras sont vides.

$$\Box(\texttt{at\_wait} \Rightarrow \neg\texttt{PA} \wedge \neg\texttt{PB}).$$

(P_2) Le robot n'est pas en train d'attendre une pièce devant la table s'il en possède déjà une dans le bras A.

$$\Box(\texttt{PA} \Rightarrow \neg\texttt{at\_table} \wedge \neg\texttt{wait\_table})$$

(P_3) S'il y a une pièce dans le bras B, le robot se déplacera fatalement vers le tapis d'évacuation.

$$\Box(\texttt{PB} \Rightarrow \Diamond\texttt{to\_dep\_belt})$$

(P_4) S'il y a une pièce dans le bras A, le robot se déplacera fatalement vers la presse.

$$\Box(\texttt{PA} \Rightarrow \Diamond\texttt{at\_press}_2)$$

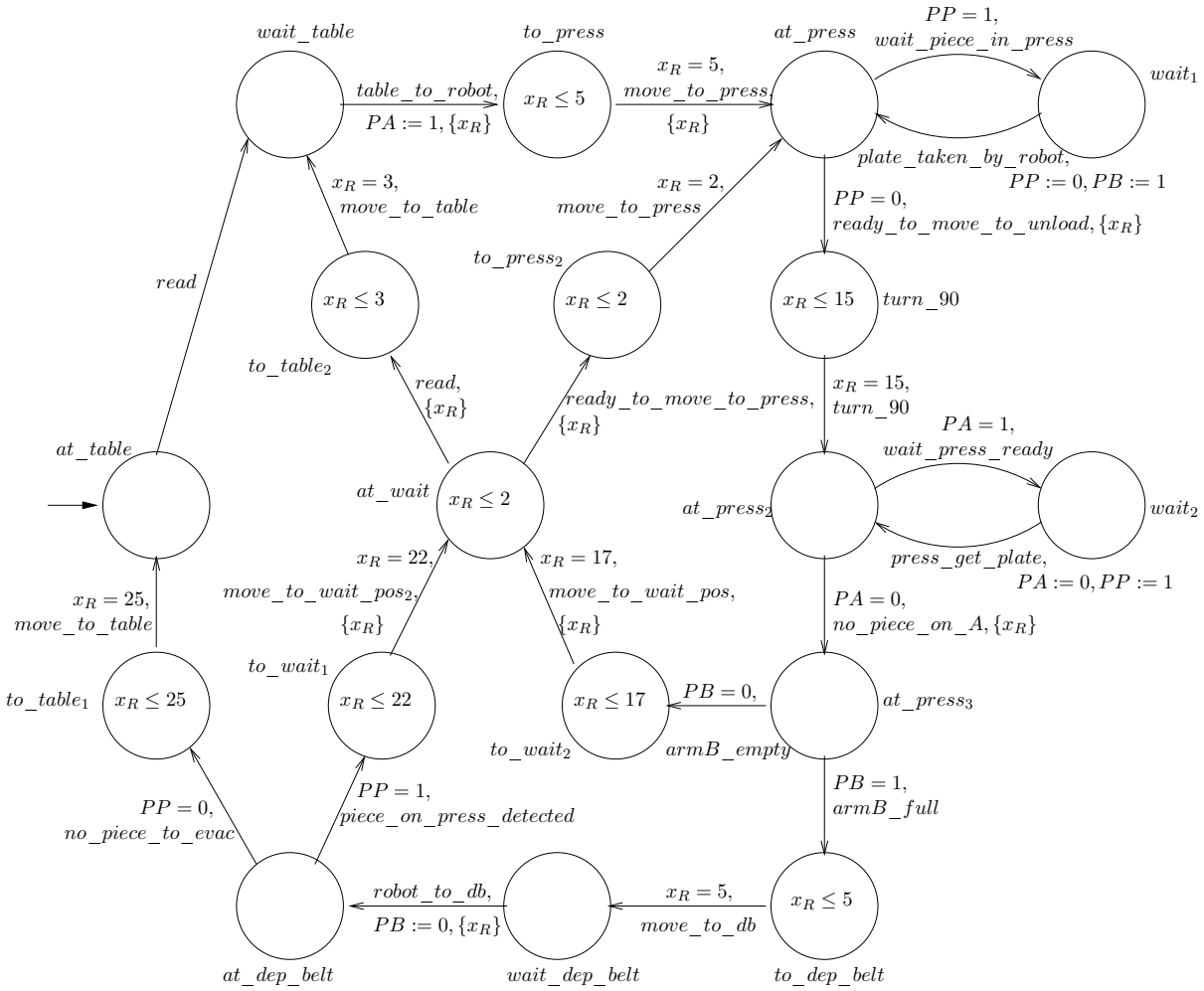


FIG. 9.7 – Automate temporisé du robot

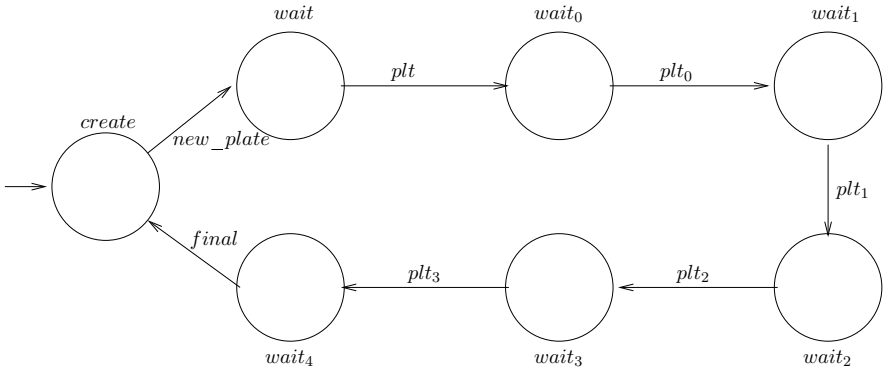


FIG. 9.8 – Automate (non temporisé) d'une pièce

(P_5) Après avoir déposé une pièce sur le tapis d'évacuation, le robot retourne vers la table en au plus 25 u.t. s'il n'y a pas de pièce dans la presse.

$$\Box(\texttt{at\_dep\_belt} \wedge \neg \texttt{PP} \Rightarrow \Diamond_{\leq 25} \texttt{at\_table})$$

(P_6) Après avoir déposé une pièce sur le tapis d'évacuation, le robot se met en position d'attente s'il y a une pièce dans la presse.

$$\Box(\texttt{at\_dep\_belt} \wedge \texttt{PP} \Rightarrow \Diamond_{\leq 22} \texttt{at\_wait})$$

(P_7) Quand le robot est en position d'attente, soit il va à la presse dans les deux u.t. pour prendre la pièce qui vient d'être traitée, soit il retourne à la table dans les 3 u.t. pour prendre une nouvelle pièce.

$$\Box(\texttt{at\_wait} \Rightarrow \Diamond_{\leq 2} \texttt{at\_press} \vee \Diamond_{\leq 3} \texttt{wait\_table})$$

La propriété suivante est une propriété de vivacité portant sur l'assemblage robot || presse :

(P_8) Si le bras A contient une pièce, alors la presse sera fatalement disponible.

$$\Box(\texttt{PA} \Rightarrow \Diamond \texttt{pl}_2)$$

9.1.4 Vérification locale et préservation vs Vérification classique

La vérification par model-checking utilise la représentation symbolique des automates temporisés décrits précédemment. Le tableau 9.2 donne la taille de chaque graphe de simulation, en termes de nombre d'états et de transitions.

Composant	Tapis d'arrivée	Capteur	Table	Robot
Etats / Transitions	39/40	7/7	6/6	4/4
Composant	Presse	Tapis d'évacuation	Pièce	Modèle complet
Etats / Transitions	6/6	2/2	7/7	1655/2395

TAB. 9.2 – Taille des graphes de simulation de chaque composant de la cellule de production

Nous avons comparé les deux approches de vérification pour ces propriétés. Dans un premier temps, nous avons considéré un modèle complet ne contenant qu'une seule pièce. Dans le cas d'une vérification directe sur ce modèle complet, nous avons obtenu que toutes les propriétés étaient vérifiées. Dans le cas d'une vérification locale, nous avons vérifié les propriétés P_1 à P_7 sur le composant du robot, et la propriété P_8 sur l'assemblage robot || presse. Ici également, les propriétés sont vérifiées. Il s'agissait ensuite de vérifier la préservation des propriétés P_1 à P_7 lors de l'intégration du robot avec la presse, puis lors de l'intégration de l'assemblage robot || presse avec les autres composants du système. Nous avons utilisé la τ-simulation temporisée DS, puisque des propriétés de vivacité devaient être préservées. La simulation ayant été vérifiée pour les deux intégrations, on peut ainsi conclure que les propriétés vérifiées localement sont préservées sur le modèle global.

Le tableau 9.3 donne les résultats de la comparaison des deux approches en termes de temps de vérification. Nous pouvons ainsi voir que, même sur un exemple de petite taille, notre approche basée sur la préservation met moins d'une seconde pour vérifier localement

les propriétés et leur préservation, alors que l'approche *classique* met presque 20 secondes.

Propriété	Vérification globale	Vérification locale	Vérification de la préservation
P_1	0.01	< 0.001	
P_2	0.01	< 0.001	
P_3	0.98	$< 0,.001$	
P_4	15.79	0.04	0.05
P_5	0.68	< 0.001	
P_6	0.48	< 0.001	
P_7	0.7	< 0.001	
P_8	0.93	0.02	0.46
Total	19.58	0.06	0.51

TAB. 9.3 – Temps de vérification des propriétés de la cellule de production (en secondes)

Dans tout ce qui précède, nous avons considéré qu'une seule pièce à la fois était présente sur le dispositif. En effet, dans la seconde approche (vérification locale et préservation), ajouter d'autres pièces au système n'affecte pas les résultats de la préservation. Il s'agit uniquement de vérifier qu'un tel ajout permet de garantir la sensibilité à la divergence et le respect de la stabilité. La sensibilité à la divergence est trivialement vérifiée, car le composant pièce ne contient pas d'actions internes (toutes ses actions sont synchronisées avec les autres composants du système). Etudions à présent le respect de la stabilité. Comme un composant *Pièce* existe déjà dans le système global, et qu'il n'existe aucune synchronisation entre les différentes pièces, aucun nouveau blocage ne peut apparaître en ajoutant un nouveau composant *Pièce*. En effet, le système pourra se comporter comme il le faisait avec une seule pièce, ou se synchroniser avec la nouvelle. Dans ce dernier cas, comme le système pouvait se synchroniser avec une pièce sans introduire de nouveaux blocages par rapport à l'assemblage *Robot* // *Pièce*, alors il pourra également se synchroniser avec les nouvelles pièces sans en introduire non plus.
Ainsi, les propriétés P_1 à P_8 restent préservées, quel que soit le nombre de pièces. En comparaison avec l'approche classique, ajouter de nouvelles pièces (et de manière générale, ajouter de nouveaux composants) augmente de manière considérable le temps et la mémoire nécessaires à la vérification des propriétés de vivacité ou de réponse bornée.

Le nombre de pièces dans le système peut être vue comme un paramètre de ce système. Cette étude de cas nous permet d'entrevoir que l'approche par préservation peut être intéressante dans les cas de figure où un même composant du système peut être répété plusieurs fois. L'idée est alors d'effectuer la vérification avec le plus petit nombre possible de composants identiques, et de garantir la préservation des propriétés pour un nombre quelconque de tels composants. Dans cette optique, nous présentons une deuxième expérimentation que nous avons réalisée, sur un système paramétré connu qui est le protocole CSMA/CD.

9.2 Le protocole CSMA/CD

Le protocole CSMA/CD (Carrier Sense, Multiple Access with Collision Detection) [IEE85, Tan89] permet de résoudre le problème de l'accès à un canal unique dans un réseau lorsque plusieurs stations essaient d'y accéder. Il permet en particulier de détecter les collisions lorsque plusieurs stations essaient d'envoyer une trame en même temps sur le canal. C'est un protocole largement utilisé pour les LANs et localisé dans la sous-couche MAC.

9.2.1 Description et modélisation du protocole

Le fonctionnement global du protocole est le suivant : une station essaie d'envoyer une trame. Si le canal n'est pas occupé (aucune autre station n'a essayé d'émettre), elle envoie sa trame. Dans le cas contraire, elle attend un certain délai, puis essaie de renvoyer cette trame. L'accès multiple au canal implique que plusieurs stations peuvent essayer d'émettre en même temps, ce qui provoque une collision. Dans ce cas, la collision doit être détectée par les stations concernées qui stoppent alors la transmission de leur trame. Celle-ci sera envoyée ultérieurement.

Nous reprenons ici la modélisation simplifiée donnée dans [Yov97], qui se concentre sur les contraintes de temps. Deux automates temporisés sont utilisés pour modéliser ce protocole : le premier modélisant une station, et le second modélisant un médium. La figure 9.9 présente ces automates. Le modèle complet du protocole est ensuite obtenu par composition parallèle (en utilisant l'opérateur $||$) du médium et d'au moins deux stations. Pour n stations, les synchronisations sont les suivantes : pour $a \in \{begin, busy, end\}$, il existe une transition synchronisée $a_i||a_M$, pour $i \in [1..n]$. De plus, les transitions cd de chaque automate sont synchronisées, c'est à dire qu'il existe une transition synchronisée $cd_M||cd_1||\cdots||cd_n$.

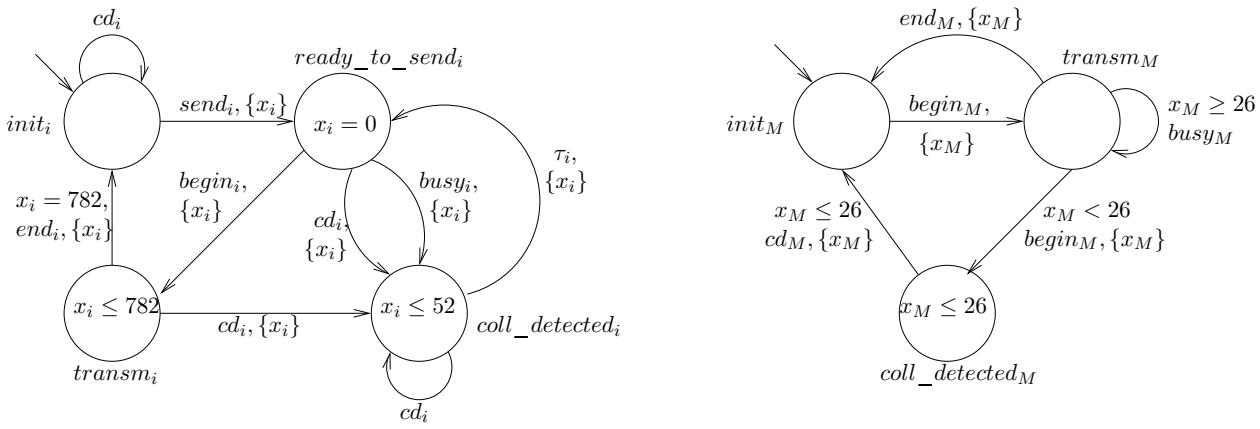


FIG. 9.9 – Automates temporisés de la station i et du médium pour le protocole CSMA/CD

9.2.2 La propriété de détection de collision

Une propriété principale de ce protocole est la suivante : *Quel que soit le nombre de stations, si une collision arrive entre deux stations i et j, $i \neq j$, les deux stations le détectent durant les 26 u.t. qui suivent.* C'est une propriété de réponse bornée, que l'on écrit de la manière suivante en MITL :

$$\Box(transm_i \wedge transm_j \Rightarrow \Diamond_{\leq 26}(coll\_detected_i \wedge coll\_detected_j)).$$

9.2.3 Vérification locale et préservation vs Vérification classique

La propriété précédente doit être vérifiée quel que soit le nombre de stations considérées. Etant donné que la propriété concerne deux stations, nous la vérifions localement sur un modèle uniquement constitué de deux stations et du médium. Pour garantir qu'elle est vraie pour un nombre indeterminé de stations, il suffit ensuite de montrer que l'intégration des autres stations préserve la propriété. Etant donné que les automates modélisant chaque station sont identiques, modulo le renommage, la propriété sera vraie quelles que soit les stations i et j pour lesquelles elle aura été vérifiée localement. Comme précédemment, nous avons comparé cette approche de vérification avec la vérification classique.

Vérification locale et préservation

Nous avons mené la vérification de la propriété pour un modèle constitué de deux stations S_1 et S_2 et du médium. Cette vérification, effectuée avec KRONOS, a pris moins de 0.001 secondes pour être effectuée. L'objectif est donc de garantir que l'ajout d'autres stations n'altérera pas le résultat de la vérification. En d'autres termes, avec un nombre $n > 2$ de stations, si S_1 et S_2 transmettent leur trame simultanément, elles détectent toujours la collision. Cette propriété étant une propriété de vivacité bornée, il faut utiliser la τ-simulation temporisée DS pour assurer sa préservation. Il faut vérifier que, pour n quelconque :

$$S_1||S_2||S_3||\cdots||S_n||Medium \preceq_{\mathcal{S}_{ds}} S_1||S_2||Medium.$$

Nous avons tout d'abord utilisé VESTA pour vérifier cette simulation pour n fixé. Cependant, la taille des graphes de simulation pour des valeurs de n supérieures à 2 est trop grande pour pouvoir être prise en compte par VESTA. Notons également que nous avons essayé d'utiliser d'autres outils, KRONOS et PROFOUNDER, pour construire ces graphes et que nous nous sommes également heurté à cette taille trop importante.
Malgré cela, il est possible d'étudier cet exemple et de montrer que la préservation est garantie par les arguments suivants. Ces arguments sont basés sur la définition de la τ-simulation temporisée DS. Tout d'abord, on voit clairement sur l'automate temporisé de la Fig. 9.9 que les stations S_i ne possèdent pas de cycles non zénon d'activité interne (leurs seules actions internes sont les actions $send_i$ et τ_i). Grâce aux propositions 3 et 6, il suffit de montrer que l'intégration des S_3, $\cdots$, S_n n'ajoute pas de blocages à $S_1||S_2||Medium$. Supposons à présent que l'on veuille ajouter une station S_3 au modèle à deux stations $S_1||S_2||Medium$. Comme pour l'ajout de nouveaux composants *pièce* dans l'exemple précédent de la cellule de production, nous allons tenir compte du fait qu'il existe déjà deux

stations dans ce modèle. Ainsi, en ajoutant la nouvelle station S_3, les synchronisations de cette station avec le médium (actions *begin*, *end* et *busy*) peuvent être effectuées de la même façon qu'elles pouvaient l'être avec les stations S_1 et S_2. Ces synchronisations ne peuvent pas introduire de blocages.
En revanche, à la différence des pièces la cellule de production, il existe des synchronisations entre stations : les actions *cd* de chaque station se synchronisent entre elles, et également avec l'action *cd* du médium. Quand une collision se produit pour deux stations, ces deux stations doivent effectuer l'action *cd*. Dans ce cas, toutes les autres stations doivent leur permettre de détecter cette collision, et doivent donc avoir la possibilité d'effectuer également l'action *cd*. Si ce n'est pas le cas, un blocage est introduit, car la synchronisation entre toutes les actions *cd* ne peut pas avoir lieu alors qu'elle le pouvait lorsque seules deux stations étaient considérées. Pour montrer qu'aucun blocage n'est introduit par l'ajout de la station S_3, nous devons assurer qu'une collision se produit entre deux stations et que celles-ci doivent effectuer l'action *cd*, toutes les autres stations sont prêtes à déclencher également cette action *cd*. Sur l'automate temporisé modélisant une station, nous voyons que chaque état est source d'une transition étiquetée par *cd*, et qu'aucune de ces transitions n'est munie d'une garde. Ainsi, chaque station est capable d'effectuer cette action à tout moment.
Ainsi, aucun blocage ne peut être introduit par l'ajout de nouvelles stations. La propriété de détection de collision, vérifiée sur le modèle à deux stations, est donc garantie quel que soit le nombre de stations considérées.

Vérification classique

La vérification classique consiste ici à vérifier la propriété sur un modèle constitué de n stations, n étant un paramètre du système. Nous avons effectué cette vérification avec KRONOS et obtenu les résultats suivants :

- Pour un modèle contenant jusqu'à six stations (S_1 à S_6), la propriété peut être vérifiée avec succès. Les temps de calcul pour la vérification varient de moins de 0.5 secondes (trois stations) à plus de 57 minutes (six stations). Ces temps de calcul présentés prennent en compte à la fois le temps nécessaire à la construction de la composition parallèle des composants du système, et le temps de vérification de la propriété sur cette composition.
- Pour un modèle contenant sept stations ou plus, nous n'avons pas pu mener la vérification. En effet, à partir de ce seuil, la construction de la composition parallèle de tous les composants du système prend un temps considérable. Par exemple, pour sept stations, nous avons interrompu la construction de la composition car elle n'avait pas abouti après dix heures de calcul. De ce fait, sur ce modèle complet avec $n > 7$, il a été impossible d'effectuer la vérification de la propriété.

9.3 Bilan

Ces deux études de cas nous ont permis de voir l'intérêt d'une approche basée sur la vérification locale des propriétés et leur préservation grâce à la τ-simulation temporisée DS, par rapport à la vérification classique effectuée sur le modèle complet.
Dans le premier exemple de la cellule de production, les comparaisons des temps de calcul pour les deux approches sont prometteuses, même si l'approche mériterait d'être validée sur des exemples de taille plus importante. Pour cela, le prototype VeSTA ,réalisé pour garantir la préservation, doit être amélioré. En effet, la mémoire utilisée par VeSTA pour effectuer cette vérification reste trop importante. Les tests effectués (sur un PC à 1024 Mo de RAM) nous permettent de dire que VeSTA n'est pour l'instant capable de traiter qu'environ 350000 à 400000 états[21]. Or, les exemples de systèmes dont nous disposions dépassaient tous cette limite, et n'ont pu être vérifiés en utilisant cette approche.
Avec la deuxième étude de cas, le protocole CSMA-CD, nous avons toutefois voulu esquisser l'intérêt de l'approche pour un certain type de systèmes paramétrés, ceux dans lesquels un même composant est répété un nombre indéterminé de fois (dans le cas du protocole traité, ce composant est celui représentant une station). Une propriété essentielle du protocole peut être vérifiée en ne considérant que le nombre minimum de stations, et la préservation assurée par une preuve informelle s'appuyant sur la τ-simulation temporisée DS, quel que soit le nombre de stations. A titre de comparaison, nous n'avons pas pu effectuer une vérification classique (en utilisant l'outil KRONOS) pour un nombre de stations supérieur à 7.

Ce cas de figure concernant les systèmes paramétrés mériterait d'être approfondi. De même, il faudrait rendre le prototype VeSTA plus efficace afin de traiter des exemples de taille plus importante. Avant d'aborder ces perspectives permettant de valider l'intérêt de la méthode sur des exemples, ainsi que des perspectives plus générales aux travaux présentés dans ce document, nous effectuons une synthèse et un bilan de ces travaux dans la partie suivante.

[21] Ce chiffre est tiré d'une expérimentation sur un système à 15 horloges.

Quatrième partie

Conclusion et Perspectives

Conclusion

Le cadre général de cette thèse est celui de la vérification par model-checking des systèmes temporisés. Nous considérons des systèmes temporisés modélisés à base de composants par des automates temporisés. Leurs propriétés sont exprimées à l'aide de la logique MITL. Notre objectif était de concilier la conception et la vérification incrémentale de tels systèmes. Bien souvent, la conception incrémentale ne permet pas de garantir la préservation des propriétés au fur et à mesure du développement. Nous nous sommes intéressés à cette problématique dans le but d'une part de faciliter la conception des modèles, et d'autre part, de faciliter leur vérification. En préservant les propriétés, la conception incrémentale permet ainsi de faire face au problème d'explosion combinatoire de leur vérification par model-checking (et qui est accentué dans le cadre des systèmes temporisés). Ce chapitre présente une synthèse et un bilan du travail réalisé, puis le chapitre suivant en annonce différentes perspectives.

1 Synthèse

Rappelons tout d'abord que les méthodes de développement incrémental consistent à modéliser un système de manière progressive. Une vue abstraite du système est d'abord considérée, puis détaillée pas à pas. Au niveau de la vérification, l'idée est de vérifier à chaque étape du processus les propriétés portant sur le niveau d'abstraction considéré, plutôt que de les vérifier sur le modèle complet de taille plus importante. Ainsi, les propriétés peuvent être vérifiées à moindre coût. Nous avons considéré deux méthodes de développement incrémental : le raffinement et l'intégration de composants. Bien sûr, ces processus de développement ne sont valables que si les propriétés établies à un certain niveau d'abstraction sont préservées par le raffinement ou l'intégration de composants.

Nous avons tout d'abord étudié comment donner un cadre formel à ces processus de développement incrémental, afin qu'ils préservent les propriétés exprimées en MITL, qui est le formalisme que nous considérons pour l'expression des propriétés d'un système. Les relations de τ-simulation sont un moyen d'assurer la préservation de propriétés, puisqu'il est connu qu'elles préservent les propriétés de sûreté. Nous avons donc commencé par étendre cette notion de τ-simulation aux automates temporisés. Pour préserver un plus large éventail de propriétés, et en particulier les propriétés de vivacité, la τ-simulation doit également être *sensible à la divergence et respecter la stabilité*. Ainsi, nous avons défini une seconde relation de τ-simulation temporisée, possédant ces deux caractéristiques : la

τ-simulation temporisée DS[22]. Nous avons montré qu'elle permet effectivement de prendre en compte un spectre plus important de propriétés, puisqu'elle préserve toute la MITL (et donc en particulier les propriétés de vivacité). D'autres propriétés qui ne s'expriment pas en MITL sont préservées, telles que l'absence de blocage et le non-zénonisme. En revanche, les automates de Büchi temporisés et l'atteignabilité ne sont pas préservés.

Dans un second temps, comme nous nous situons dans un cadre compositionnel, nous nous sommes intéressés aux propriétés de ces relations vis-à-vis de différents opérateurs de composition : un premier opérateur utilisant un paradigme de composition *à la CSP*, et un second utilisant un paradigme *à la CCS* et une notion de priorités entre actions. Nous avons retenu trois propriétés essentielles à garantir : la propriété que nous avons appelée *intégration*, la *compatibilité* et la *compositionnalité*. La propriété d'intégration permet de garantir la composabilité, c'est-à-dire la préservation de propriétés par intégration de composants. La compatibilité et la compositionnalité permettent quant à elles de vérifier compositionnellement le raffinement : la préservation de propriétés pourra être garantie uniquement en vérifiant le raffinement entre chaque composant raffiné et sa version abstraite, plutôt qu'en vérifiant le raffinement entre le modèle complet raffiné et sa version abstraite. Ainsi le modèle complet raffiné n'aura jamais à être construit entièrement.

Sous certaines hypothèses simples, la τ-simulation temporisée DS possède les trois propriétés précédentes vis-à-vis de l'opérateur *à la CCS*. Ainsi, cet opérateur est donc bien adapté à un développement incrémental formalisé par les τ-simulations que nous avons définies. Notamment, lors d'une intégration de composants, la préservation de toute propriété exprimée en MITL est assurée gratuitement (c'est-à-dire sans aucune vérification). Du point de vue du raffinement et pour ces mêmes propriétés, la compositionnalité n'est assurée qu'au prix d'une vérification globale de la sensibilité à la divergence.
En revanche, l'opérateur *à la CSP* n'est bien adapté que lorsque seules des propriétés de sûreté sont considérées et doivent être préservées. En effet, seule la τ-simulation préservant les sûretés possède les trois propriétés précédentes vis-à-vis de cet opérateur. Le raffinement ne peut donc être vérifié de manière compositionnelle que si seules des propriétés de sûreté doivent être préservées. Dans le cadre d'une intégration de composants, seule la préservation de ces propriétés est garantie gratuitement. Pour assurer la préservation de toute propriété MITL, en particulier, les propriétés de vivacité, il est nécessaire de vérifier la τ-simulation temporisée DS.

Cette vérification a été implantée dans le prototype VESTA (*Ve*rification of *S*imulations for *T*imed *A*utomata). VESTA considère des systèmes temporisés modélisés à base de composants par des automates temporisés, où l'opérateur de composition considéré est celui *à la CSP*. VESTA nous a permis de mener quelques études de cas pour étudier l'intérêt de la τ-simulation temporisée DS pour les méthodes de développement incrémental, en particulier l'intégration de composants. Ces études nous ont permis de constater que, même sur des exemples de taille raisonnable, le temps nécessaire à la vérification locale et à la garantie de la préservation (par vérification de la τ-simulation temporisée DS) est

[22]pour sensible à la Divergence et respectant la Stabilité.

inférieur à celui nécessaire pour la vérification classique. De plus, ces expérimentations nous ont permis de mettre en évidence que cette approche par préservation peut être également profitable dans le cas de systèmes paramétrés dans lesquels un même composant peut être répété un nombre indéterminé de fois, ce nombre représentant ainsi le paramètre du système.

2 Bilan

Dans ce document, nous avons donc étendu la notion de τ-simulation aux systèmes temporisés. L'idée développée est d'utiliser ces τ-simulations temporisées pour garantir la préservation de propriétés lors du développement incrémental des systèmes temporisés à composants.

On trouve dans la littérature de nombreuses notions de simulation étendues aux systèmes temporisés, mais ces relations ne permettent de préserver que les propriétés de sûreté. Même si nous avons défini une τ-simulation temporisée qui préserve également les propriétés de sûreté, nous nous sommes concentrés sur une τ-simulation temporisée, dite sensible à la divergence et respectant la stabilité, qui préserve un plus large éventail de propriétés et notamment les propriétés de vivacité et de réponse bornée.

Pour le développement incrémental des systèmes à composants, le bilan quant à l'utilisation de cette relation pour la préservation de propriétés s'avère plutôt positif. En effet, nous avons identifié un opérateur de composition, *à la CCS*, pour lequel cette relation est bien adaptée. Ce n'est pas le cas de l'autre opérateur étudié, *à la CSP*, pour lequel la vérification de la relation doit être effectuée. Malgré cela, par rapport à une vérification classique, le gain en pratique en termes de temps de calcul reste néammoins intéressant, comme l'atteste un exemple que nous avons traité.

Le principal inconvénient de ce dernier opérateur provient du fait qu'il possède des règles de synchronisation trop fortes, pouvant mener à l'introduction de blocages. Des travaux actuels sur la composition tendent à aller vers la définition d'opérateurs de composition utilisant un paradigme de synchronisation dit plus *flexible*, comme c'est le cas pour l'opérateur *à la CCS* étudié dans ce document. La τ-simulation temporisée DS semblerait donc plus adaptée à de tels cadres de composition. Une étude prospective d'opérateurs de composition ou de "types" de synchronisation appropriés constitue donc une suite logique aux travaux effectués. D'autres pistes sont à envisager afin de mettre encore plus en valeur l'intérêt de cette approche, ainsi que des axes de travail plus généraux que ce soit dans le domaine des systèmes à composants, ou des automates temporisés. Nous les présentons dans le chapitre suivant.

Perspectives

Nous présentons dans ce chapitre les perspectives liées aux travaux présentés dans ce document. Elles sont répertoriées en trois catégories. Tout d'abord, nous introduisons dans la section 1 des perspectives permettant de valoriser l'intérêt des relations de simulation proposées. Ces perspectives sont dans la continuité directe des travaux présentés dans ce document, et ont déjà été évoquées précédemment pour certaines.
Par ailleurs, nous avons examiné la question de la préservation de propriétés lors du raffinement d'automates temporisés. Une autre question qui se pose concerne la préservation de ces propriétés sur une implantation de ces automates temporisés. Récemment, des solutions ont été apportées, mais des difficultés subsistent auxquelles il convient d'apporter des solutions. Nous abordons ce point dans la section 2.
Enfin, des perspectives plus larges sont également à envisager dans le domaine plus général du développement à base de composants. Les travaux que nous avons exposés permettent d'apporter une solution à l'un des points à étudier dans ce domaine, celui de la composabilité lors de l'intégration de composants. Néammoins, d'autres questions-clés se posent. Les différentes investigations à effectuer dans ce domaine sont présentées dans la section 3.

1 Valorisation de l'intérêt des relations de simulation

Pour renforcer l'intérêt des τ-simulations, trois pistes peuvent être envisagées. La première, avancée dans la section 1.1 est une suite logique à l'étude qui a été effectuée dans le chapitre 6, consistant à étudier les propriétés de ces τ-simulations vis-à-vis d'autres paradigmes de composition et de synchronisation, afin de bien se rendre compte des cadres pour lesquels elles sont le plus adaptées. La seconde voie concerne l'amélioration du prototype VeSTA en y ajoutant des optimisations et de nouvelles fonctionnalités, afin d'étudier l'apport en pratique sur des systèmes de plus grande taille. Nous détaillons cela dans la section 1.2. Enfin, nous avons évoqué précédemment que les τ-simulations peuvent avoir un intérêt tout particulier lors de la vérification des systèmes paramétrés dans lesquels un même composant est susceptible d'être répété un nombre indéterminé de fois. Nous développons cette idée dans la section 1.3.

1.1 Etude d'autres paradigmes de composition

Nous avons étudié l'intérêt des τ-simulations dans le cadre de deux paradigmes différents de composition. Il serait intéressant d'étudier d'autres environnements de com-

position afin de pouvoir généraliser l'intérêt de la méthode. Comme nous l'avons dit précédemment, la τ-simulation temporisée DS ne semble pas adaptée aux opérateurs de composition possédant des règles de synchronisation trop fortes. En effet, l'utilisation de tels opérateurs amène généralement à l'introduction de blocages dans la composition. Il convient d'étudier des opérateurs de composition dont les règles de synchronisation sont plus souples.

Dans ce cadre, un concept de composition particulièrement intéressant pourrait être celui présenté dans [GS03, GS05]. Il a notamment été utilisé pour traiter des applications non-triviales, comme dans [GOO04]. Un système à composants est ici modélisé en trois couches : une couche *basse* représentant les composants, une couche *intermédiaire* représentant les interactions entre composants, et enfin une dernière couche indiquant par une contrainte les restrictions à apporter au modèle complet pour qu'il se comporte de la manière attendue. Cette dernière couche peut notamment être utilisée pour exprimer des politiques d'ordonnancement sur le système comme cela est réalisé dans [AGS00, Alt01]. Ce principe a été récemment mis en œuvre dans la plate-forme d'exécution BIP[23] [BBS06]. Un opérateur de composition a également été défini pour les systèmes modélisés par ce concept de couches. Il pourrait être intéressant d'étudier les propriétés de la τ-simulation temporisée DS dans ce cadre compositionnel précis.

1.2 Amélioration du prototype VeSTA

Des évolutions peuvent être apportées au prototype VeSTA. Ces évolutions concernent deux directions : optimiser les structures de données utilisées afin de réduire l'espace mémoire nécessaire à l'exploration des graphes de simulation, et généraliser la vérification partielle des simulations évoquée dans le chapitre 8.

Optimisations

Jusqu'à présent, nous n'avons pu tester l'impact en pratique de la méthode proposée que sur des exemples de taille raisonnable. La raison principale est que, d'après nos expérimentations[24], le prototype VeSTA ne nous a pas permis de traiter des systèmes de plus de 400000 états (symboliques), avec la mémoire dont nous disposions. Ce chiffre est toutefois à nuancer suivant le nombre d'horloges présentes dans le système traité (15 horloges pour le chiffre donné précédemment). En effet, pour n horloges, la structure de données utilisée pour représenter les zones, les matrices de différences bornées (*Difference Bound Matrices*), nécessitent un espace mémoire en $\mathcal{O}(n^2)$ pour représenter une zone. Le nombre d'horloges considérées est un facteur prépondérant.

De ce point de vue, il serait opportun de pouvoir diminuer le nombre d'horloges considérées, en utilisant des abstractions adaptées. Ainsi, une optimisation, d'ailleurs implantée dans des outils d'analyse de systèmes temporisés tels que Kronos et Uppaal, serait d'utiliser l'*Active-clock reduction*[DY96, DT98] qui est une abstraction consistant à éliminer

[23] `http : //www − verimag.imag.fr/ ∼ async/BIP/bip.html`
[24] sur un PC à 1Go de RAM.

les horloges inactives. Elle consiste, pour chaque état du graphe, à ne conserver que les horloges qui sont actives dans ces états. Une horloge x est dite active dans un état q si :

- soit elle apparaît dans l'invariant de q ou dans la garde d'une action issue de q, ou
- soit elle apparaît dans l'invariant ou dans la garde d'une action issue d'un état pouvant être atteint depuis q, sans que x soit remise à zéro entre temps.

Cette abstraction permettrait de réduire l'espace mémoire nécessaire à l'exploration de l'espace d'états du système.

Généralisation de la vérification partielle de la τ-simulation temporisée DS

Nous avons commencé à implanter une vérification *partielle* de la τ-simulation temporisée DS et de la préservation de propriétés dans VeSTA. L'objectif d'une telle vérification est de ne vérifier la préservation que pour les propriétés qui doivent être préservées. En effet, comme nous l'avons présenté dans le chapitre 8, la vérification de la simulation peut échouer, dû notamment à des introductions de blocage dans certains chemins, alors que ces chemins ne concernent pas les propriétés à préserver.
Cette vérification partielle n'est pour l'instant disponible dans VeSTA que pour des propriétés (non temporisées) de réponse du type $\Box(p \Rightarrow \Diamond q)$. Il serait intéressant d'étendre ce type de vérification à d'autres schémas de propriétés de vivacité ou de réponse bornée. En outre, ce type de vérification devrait permettre d'optimiser le temps de vérification. En effet, certaines opérations très coûteuses telles que le calcul de compléments de DBM (pour la vérification du respect de la stabilité), ne seraient plus effectuées systématiquement, mais uniquement pour les états des chemins concernés par les propriétés à préserver.

1.3 Intérêt pour les systèmes paramétrés

Les expérimentations effectuées sur la cellule de production et le protocole CSMA-CD nous ont permis d'entrevoir l'intérêt de la méthode pour un type particulier de systèmes paramétrés utilisant l'opérateur de composition *à la CSP*. Dans le cas où un système S est susceptible d'inclure un nombre indéterminé n de composants $C_{i,i=1..n}$ identiques (modulo le renomage), on dit que ce nombre n est un paramètre du système S. Le système S est obtenu par composition des C_i avec un environnement E, c'est-à-dire, $S = E||C_1|| \cdots ||C_n$.
Si l'on reprend les exemples traités dans le chapitre 9, dans le cas de la cellule de production, le composant identique est celui représentant une pièce, et dans le cas du protocole CSMA-CD, il s'agit de celui modélisant une station.
Dans de tels systèmes paramétrés, les cas intéressants sont quand la vérification peut être effectuée avec un petit nombre fixé l de composants identiques, c'est-à-dire $S_l = E||C_1|| \cdots ||C_l$, tout en garantissant la préservation des propriétés sur $S_m = E||S_1|| \cdots ||S_m$, $\forall m \geq l$, sous certaines conditions simples.
En utilisant les propriétés de la τ-simulation temporisée DS dans le cas de l'opérateur *à la CSP*, il serait suffisant d'obtenir des conditions sur les C_i et/ou leurs synchronisations qui garantissent qu'ajouter ces C_i n'introduit pas de blocages, outre le fait de s'assurer qu'ils ne contiennent pas de cycles d'activité interne. Par exemple, pour la cellule de production, la condition est simplement qu'il n'y a pas de synchronisations entre les pièces. Un travail intéressant serait de déterminer de telles conditions.

Dans le cas non temporisé, des travaux dans cette direction ont été effectués notamment dans [EN95] pour le cas spécifique des réseaux en anneau de processus concurrents identiques. Suivant le type de propriétés à préserver, et en utilisant les spécificités d'un réseau en anneau, les auteurs montrent que la vérification peut être menée sur des systèmes ne contenant qu'un petit nombre de processus.

2 Préservation de propriétés sur l'implantation des automates temporisés : le problème de l'implémentabilité

Nous abordons à présent des perspectives plus larges. L'implémentabilité des automates temporisés est un problème en cours d'investigation. Il consiste à étudier si les propriétés d'un automate temporisé peuvent être préservées sur l'implantation correspondant à cet automate, lorsqu'on l'exécute sur une plate-forme donnée. Une autre question essentielle est de savoir si cette préservation peut être garantie par le passage d'une plate-forme à une autre plus *efficace*.
Dans ce contexte, on peut reprocher aux automates temporisés de posséder une sémantique souvent qualifiée d'*idéale* pour représenter les aspects temporisés d'un système : temps d'exécution des actions négligé puisqu'elles sont considérées immédiates, horloges continues et infiniment précises, etc. En pratique, ces concepts *parfaits* ne sont plus vrais. Si l'on considère des plate-formes d'exécution réalistes, trois critères sont à prendre en compte selon [AMRT05] : (i) gestion des entrées / sorties du système, ou des variables partagées, ainsi que les délais nécessaires aux opérations se rapportant à cette gestion, (ii) précision de l'horloge globale du système, et (iii) vitesse, fréquence et précision des calculs.
Ainsi, pour répondre à ces questions de préservation des propriétés d'un automate temporisé A, deux solutions en particulier ont été proposées récemment :

- Une *approche basée sur la modélisation* [AT05] consistant à modéliser la plate-forme d'exécution sur laquelle doit s'exécuter l'implantation par trois automates temporisés (modèle de l'interface d'entrées/sorties, modèle de l'horloge digitale et modèle d'exécution), et à transformer A en un automate (non temporisé) représentant son implantation. La composition de tous ces automates, appelée *modèle d'exécution* et notée M, permet de modéliser l'exécution de A sur une plate-forme donnée. Cette approche permet de vérifier les propriétés de A sur M, en utilisant des techniques de vérification classique pour les automates temporisés.
- Une *approche basée sur la sémantique* [dWDR04] consistant cette fois à considérer un modèle particulier de plate-forme. La sémantique de l'implantation de A sur cette plate-forme, appelée *program semantics*, est paramétrée par deux délais Δ_P et Δ_L. Le délai Δ_P représente la période de l'horloge digitale de la plate-forme. Le délai Δ_L représente le temps maximal d'exécution d'un pas du programme (chaque pas consiste à enregistrer la valeur de l'horloge digitale, mettre à jour les "informations" envoyées par l'environnement, déterminer quelles transitions de A peuvent être déclenchées en calculant leur garde et enfin exécuter une des transitions déclen-

chables). Une nouvelle sémantique nommée *Almost ASAP semantics* et paramétrée par un délai Δ, est introduite pour les automates temporisés. Intuitivement, ce paramètre Δ représente un "délai de réaction", et permet ainsi de relaxer les contraintes d'horloges. Par exemple, une action devant initialement se déclencher à un temps t peut s'effectuer pendant l'intervalle de temps $[t - \Delta, t + \Delta]$.

Dans cette dernière approche, une condition suffisante portant sur les paramètres Δ, Δ_L et Δ_P permet de garantir la préservation des propriétés sur l'implantation [dWDR04]. La préservation sur une plate-forme plus efficace est également garantie. La question est alors de savoir s'il existe une valeur de Δ telle que les propriétés vérifiées sur A (avec la sémantique classique) sont satisfaites avec cette nouvelle sémantique. C'est ce que l'on appelle la satisfaction *robuste* et le model-checking avec une telle sémantique des automates est qualifié de *model-checking robuste*. Un algorithme de model-checking robuste a été proposé dans [dWDMR04] pour les propriétés de sûreté et a été étendu dans [BMR06] pour prendre en compte toutes les propriétés LTL, ainsi que certaines propriétés temporisées.

Dans le cadre de l'approche basée sur la modélisation, une perspective intéressante serait d'étudier la compatibilité des simulations que nous avons définies dans le cadre spécifique de A et de son modèle d'exécution M, en tenant compte de la manière spécifique de modéliser la plate-forme d'exécution. A terme, une telle relation pourrait permettre de dégager des conditions suffisantes sur les automates modélisant la plate-forme, garantissant la préservation des propriétés sur cette plate-forme.
L'autre question essentielle est de savoir si passer à une plate-forme plus *efficace* permet toujours d'assurer la préservation des propriétés. Compte tenu de l'expressivité du modèle de la plate-forme d'exécution, cette question est loin d'être triviale. En particulier, les propriétés de type $\Box\neg\texttt{bad}$ ne sont pas préservées [AT05]. La question de disposer d'une relation formelle et *réaliste* de raffinement pour ces plates-formes d'exécution est donc un challenge actuel, afin d'éluder cette question *faster is better ?*.

3 Perspectives générales au développement à base de composants

Dans un autre registre, de nombreux champs d'investigation existent dans le cadre général du développement à base de composants, du fait que ce type de développement reçoit de plus en plus d'attention. Le principe est de construire des systèmes complexes en assemblant des unités plus petites, pré-existantes : les composants. L'idée sous-jacente à cette démarche est qu'un composant doit être réutilisable, c'est-à-dire qu'il doit pouvoir être intégré dans différents systèmes, dans des contextes différents de celui pour lequel il avait été construit initialement. Ce concept de composants sur étagère[25], a fait naitre de nombreuses questions à propos de la construction d'un système à partir de tels composants. D'après [Sif05], deux points essentiels sont les suivants :

[25] les COTS (*commercial off-the-shelf components*)

1. *"Encompass heterogeneous composition to ensure interoperability of components"*. Les composants réutilisés au sein d'un même système n'ont pas été nécessairement conçus au départ pour fonctionner ensemble. Il est donc indispensable d'avoir la possibilité d'assembler, de faire interagir de tels composants hétérogènes. C'est ce que l'on appelle garantir l'interopérabilité des composants. On dénote deux enjeux majeurs : la compatibilité des composants, et la capacité d'un composant de se substituer à un autre (la *substitutability*).
2. *"Provide results guaranteeing correctness-by-construction for essential system properties such as deadlock-freedom, progress and liveness"*. Ce point recouvre deux aspects essentiels à garantir, que nous avons mentionnés précédemment : la composabilité, c'est-à-dire le fait de préserver les propriétés des composants après intégration, et la compositionnalité, permettant de déduire des propriétés globales du système à partir des propriétés des composants.

Ces deux points constituent donc un domaine de recherche très actif. Nous avons abordé dans ce document le thème de la composabilité, pour lequel la τ-simulation temporisée DS peut servir de condition suffisante. Il reste toutefois beaucoup à faire dans le domaine de la compositionnalité. Des conditions suffisantes pour garantir l'absence de blocages dans un système obtenu par intégration de composants sans blocages sont étudiées dans [GS03, GS05]. Il est nécessaire de pouvoir faire plus, afin d'étendre ce principe à un plus large éventail de propriétés, sûreté et/ou vivacité, en utilisant par exemple un paradigme de type *assume-guarantee* qui peut permettre d'inférer des propriétés globales du système à partir des propriétés des composants, et des hypothèses que chacun nécessite sur les autres composants.
Le premier point, concernant l'interopérabilité, ne doit également pas être négligé et doit faire l'objet de nombreuses investigations. En effet, avant de pouvoir étudier les questions de composabilité et de compositionnalité, il est nécessaire d'avoir la possibilité d'assembler les composants. Or, les composants étant couramment développés de manière indépendante puis réutilisés, leurs interfaces risquent généralement de ne pas coincider. Ainsi, la notion d'adaptateurs entre interfaces de composants a été définie. De nombreux travaux, citons notamment [YS97, MLS06], se sont attelés et doivent encore être effectués dans le domaine de la synthèse d'adaptateurs ou de la vérification qu'un adaptateur donné garantit l'interopérabilité des composants pour lesquels il a été conçu.

Bibliographie

[Abr96a] J.-R. Abrial. *The B Book : Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0-52-149619-5.

[Abr96b] J.-R. Abrial. Extending B without changing it (for developing distributed systems). In *1st Conference on the B method*, pages 169–190, Nantes, France, November 1996.

[Abr97] J.-R. Abrial. Constructions d'automatismes industriels avec b. In *Proceedings of Approches Formelles dans l'Assistance au Développement de Logiciels (AFADL'97)*, Toulouse, France, Mai 1997. Invited Lecture.

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. Model-Checking in Dense Real-time. *Information and Computation*, 104(1) :2–34, 1993.

[ACHH93] R. Alur, C. Courcoubetis, T. Henzinger, and P.-H. Ho. Hybrid Automata : an Algorithmic Approach to Specification and Verification of Hybrid Systems. In *Proceedings of Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, pages 209–229. Springer-Verlag, 1993.

[AD94] R. Alur and D.L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2) :183–235, 1994.

[AFH96] R. Alur, T. Feder, and T.A. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43 :116–146, 1996.

[Age91] European Space Agency. ESA software engineering standards. ESA PSS-05-0 Issue 2, February 1991.

[AGS00] K. Altisen, G. Goessler, and J. Sifakis. A methodology for the construction of scheduled systems. In *Proceedings of 6^{th} International Symposium on Formal on Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'00)*, volume 1926 of *Lecture Notes in Computer Science*, pages 106–120, Pune, India, September 2000. Springer-Verlag.

[Alt01] K. Altisen. *Application de la synthèse de contrôleur à l'ordonnancement de systèmes temps-réel*. PhD thesis, Institut National Polytechnique, Grenoble, France, December 2001. In French.

[Alu91] R. Alur. *Techniques for Automatic Verification of Real-Time Systems*. PhD thesis, Department of Computer Science, Stanford University, 1991.

[AMRT05] K. Altisen, N. Markey, P.-A. Reynier, and S. Tripakis. Implémentabilité des automates temporisés. In *5^e Colloque sur la Modélisation des Systèmes*

Réactifs (MSR'05), pages 395–406, Autrans, France, October 2005. Hermès. Invited paper.

[AN01] P.A. Abdulla and A. Nylen. Timed Petri nets and BQOs. In *Proceedings of 22nd International Conference on Applications and Theory of Petri Nets (ICATPN'01)*, volume 2075 of *Lecture Notes in Computer Science*, pages 53–70, Newcastle, UK, 2001. Springer-Verlag.

[Arn92] A. Arnold. *Systèmes de transitions finis et sémantique des processus communiquants*. Masson, 1992.

[AT05] K. Altisen and S. Tripakis. Implementation of Timed Automata : an Issue of Semantics or Modeling ? Technical Report TR-2005-12, Verimag, Grenoble, France, June 2005.

[BB94] J.C.M. Baeten and J.A. Bergstra. Real Time Process Algebra with infinitesimals. In C. Verhoef A. Ponse and S.F.M. van Vlijmen, editors, *Algebra of Communicating Processes*, pages 148–187, 1994.

[BBS06] A. Basu, M. Bozga, and J. Sifakis. Modeling Heterogeneous Real-time Systems in BIP. In *Proceedings of the 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM'06)*, pages 3–12, Pune, India, September 2006.

[BD91] B. Berthomieu and M. Diaz. Modeling and Verification of time dependent systems using time Petri nets. *IEEE Transactions on Software Engineering*, 17 :259–273, March 1991.

[BD00] B. Bérard and C. Dufourd. Timed Automata and Additive Clock Constraints. *Information Processing Letters*, 75(1-2) :1–7, July 2000.

[BDFP04] B. Bérard, C. Dufourd, E. Fleury, and A. Petit. Updatable Timed Automata. *Theoretical Computer Science*, 321(2-3) :291–345, August 2004.

[BDGP98] B. Bérard, V. Diekert, P. Gastin, and A. Petit. Characterization of the Expressive Power of Silent Transitions in Timed Automata. *Fundamenta Informaticae*, 36(2) :145–182, 1998.

[BDL+01] G. Behrmann, A. David, K. Larsen, O. Möller, P. Pettersson, and W. Yi. UPPAAL - Present and Future. In *Proceedings of the 40th IEEE Conference on Decision and Control*. IEEE Computer Society Press, 2001.

[BDM+98] M. Bozga, C/ Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. KRONOS : a Model-Checking Tool for Real-Time Systems. In *Proceedings of the 10th International Conference on Computer-Aided Verification (CAV'98)*, volume 1427 of *Lecture Notes in Computer Science*, pages 546–550, Vancouver, Canada, June 1998. Springer-Verlag.

[BGK+96] J. Bengtsson, W.O.D. Griffioen, K.J. Kristoffersen, K.G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an Audio Protocol with Bus Collision Using UPPAAL. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the 8th International Conference on Computer-Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 244–256. Springer-Verlag, July 1996.

[BGP97a] B. Bérard, P. Gastin, and A. Petit. Refinement and abstraction for timed languages. Research Report LSV-97-3, Laboratoire Spécification et Vérification (LSV), ENS Cachan, France, 1997.

[BGP97b] B. Bérard, P. Gastin, and A. Petit. Timed automata with non observable actions : expressive power and refinement. Research Report 97.23, Laboratoire d'Informatique Algorithmique, Fondements et Applications (LIAFA), Paris, France, 1997.

[BJK00] F. Bellegarde, J. Julliand, and O. Kouchnarenko. Ready-simulation is not Ready to Express a Modular Refinement Relation. In *Proceedings of the 3rd International Conference on Fondamental Aspects of Software Engineering (FASE'00)*, volume 1783 of *Lecture Notes in Computer Science*, pages 266–283, Berlin, Germany, April 2000. Springer-Verlag.

[BJMO05] F. Bellegarde, J. Julliand, H. Mountassir, and E. Oudot. On the contribution of a τ-simulation in the incremental modeling of timed systems. In *Proceedings of the 2nd International Workshop on Formal Aspects of Component Software (FACS'05)*, volume 160 of *Electronic Notes in Theoretical Computer Science*, pages 97–111, Macao, Macao, October 2005. Elsevier.

[BJMO06a] F. Bellegarde, J. Julliand, H. Mountassir, and E. Oudot. Experiments in the use of τ-simulations for the components-verification of real-time systems. In *Proceedings of the 5th International Workshop on Specification And Verification of Component-Based Systems (SAVCBS'06)*, Portland, Oregon, USA, November 2006. Also available on ACM Digital Library.

[BJMO06b] F. Bellegarde, J. Julliand, H. Mountassir, and E. Oudot. The Tool VeSTA : Verification of Simulations for Timed Automata. Technical Report RT2006-01, LIFC, Laboratoire d'Informatique de l'Université de Franche-Comté, July 2006.

[BK84] J.A. Bergstra and J.W. Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3) :109–137, 1984.

[BM01] J.C.M. Baeten and C.A. Middelburg. *Process Algebra with Timing : real time and discrete time*, chapter 10. Elsevier, 2001.

[BMR06] P. Bouyer, N. Markey, and P.-A. Reynier. Robust Model-Checking of Linear-Time Properties in Timed Automata. In *Proceedings of the 7th Latin American Symposium on Theoretical Informatics (LATIN'06)*, volume 3887 of *Lecture Notes in Computer Science*, pages 238–249, Valdivia, Chile, March 2006. Springer-Verlag.

[Bor98] S. Bornot. *De la composition des systèmes temporisés.* PhD thesis, Universite Joseph Fourier, Grenoble, France, December 1998.

[Bou02] P. Bouyer. *Modèles et Algorithmes pour la Vérification des Systèmes Temporisés.* PhD thesis, ENS Cachan, France, April 2002.

[Bou03] P. Bouyer. Untameable Timed Automata! In *Proceedings of the 20th Symposium on Theoretical Aspects of Computer Science (STACS'03)*, volume 2607 of *Lecture Notes in Computer Science*, pages 620–631, Berlin, Germany, February 2003. Springer-Verlag.

[Bou04] P. Bouyer. Forward Analysis of Updatable Timed Automata. *Formal Methods in System Design*, 24(3) :281–320, May 2004.

[Bow96] F.D.J Bowden. Modeling time in petri nets, 1996. Presented at the 2^{nd} Australian-Japan workshop on Stochastic Models.

[Bry86] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computer*, 35(8) :677–691, 1986.

[BS91] J.A. Brzozowski and C.J.H. Seger. Advances in asynchronous circuit theory. part ii : Bounded inertial delay models, mos circuits, design techniques. *Bulletin of the European Association for Theoretical Computer Science*, 43(3) :199,263, 1991.

[BS00] S. Bornot and J. Sifakis. An Algebraic Framework for Urgency. *Information and Computation*, 163 :172–202, 2000.

[BST97] S. Bornot, J. Sifakis, and S. Tripakis. Modeling Urgency in Timed Systems. In *COMPOS'97*, volume 1536 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[Bur03] A. Burns. How to verify a safe real-time system : The application of model-checking and timed automata to the production cell case study. *Real-Time Systems Journal*, 24(2) :135–152, 2003.

[CE81] E. Clarke and E. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Proceedings of Workshop on Logic of Programs*, volume 131 of *Lecture Notes in Computer Science*. Springer-Verlag, 1981.

[Cer92a] K. Cerans. *Algorithmic problems in analysis of real time system specifications*. PhD thesis, University of Latvia, 1992.

[Cer92b] K. Cerans. Decidability of bisimulation equivalence for parallel timer processes. In *Proceedings of the 4^{th} workshop on Computer-Aided Verification (CAV'92)*, volume 663 of *Lecture Notes in Computer Science*, pages 302–315. Springer-Verlag, 1992.

[CG00] C. Choffrut and M. Goldwurm. Timed Automata with Periodic Clock Constraints. *Journal of Automata, Languages and Combinatorics*, 5 :371–404, 2000.

[CGL94] E.M. Clarke, O. Grumberg, and D.E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5) :1512–1542, 1994.

[CGP99] E. Clarke, O. Grumberg, and A. Peled. *Model Checking*. The MIT Press, 1999. ISBN 0-262-03270-8.

[Che92] L. Chen. An interleaving model for real-time systems. In *Proceedings of the 2^{nd} International Symposium on Logical Foundations of Computer Science*, volume 620 of *Lecture Notes in Computer Science*, pages 81–92. Springer-Verlag, 1992.

[Dij75] E. W. Dijkstra. Guarded commands, nondeterminancy and formal derivation of programs. *Communications of the ACM*, 18(8) :453–457, 1975.

[Dij76] E. W. Dijkstra. *A discipline of programming.* Prentice-Hall, 1976.

[DJK03] C. Darlot, J. Julliand, and O. Kouchnarenko. Refinement Preserves PLTL Properties. In *Proceedings of 3rd International Conference on B and Z Users (ZB'03)*, volume 2651 of *Lecture Notes in Computer Science*, pages 408–420, Turku, Finlande, June 2003. Springer-Verlag.

[DNS05] D. Detlefs, G. Nelson, and J.B. Saxe. Simplify : a theorem prover for program checking. *Journal of the ACM*, 52(3) :365–473, 2005.

[DOTY96] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Proceedings of Hybrid Systems III, Verification and Control*, volume 1066 of *Lecture Notes in Computer Science*. Springer-Verlag, 1996.

[DR03] D. Déharbe and S. Ranise. Applying light-weight theorem proving to debugging and verifying pointer programs. In *Proceedings of the 4th workshop on First-order Theorem Proving (FTP'03)*, volume 86 of *Electronic Notes in Theoretical Computer Science*. Elsevier, May 2003.

[DS95] J. Davies and S. Schneider. A brief history of timed CSP. *Theoretical Computer Science*, 138(2) :243–271, 1995.

[DT98] C. Daws and S. Tripakis. Model checking of real-time reachability properties using abstractions. In *Proceedings of the 4th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, volume 1384 of *Lecture Notes in Computer Science*, pages 313–329, Lisbon, Portugal, 1998. Springer-Verlag.

[dWDMR04] M. de Wulf, L. Doyen, N. Markey, and J.-F. Raskin. Robustness and Implementability of Timed Automata. In *Proceedings of the Joint Conferences Formal Modelling and Analysis of Timed Systems (FORMATS'04) and Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'04)*, volume 3253 of *Lecture Notes in Computer Science*, pages 118–133. Springer-Verlag, 2004.

[dWDR04] M. de Wulf, L. Doyen, and J.-F. Raskin. Almost ASAP Semantics : From Timed Models to Timed Implementations. In *Proceedings of the 7th International Workshop on Hybrid Systems : Computation and Control (HSCC'04)*, volume 2993 of *Lecture Notes in Computer Science*, pages 296–310. Springer-Verlag, 2004.

[DY95] C. Daws and S. Yovine. Two examples of verification of multirate timed automata with KRONOS. In *Proceedings of the 16th IEEE Real Time Systems Symposium (RTSS'98)*, Pisa, Italy, 1995. IEEE Computer Society Press.

[DY96] C. Daws and S. Yovine. Reducing the number of clock variables in timed automata. In *Proceedings of the 17th IEEE Real-Time Systems Symposium (RTSS'96)*. IEEE Computer Society Press, December 1996.

[EH82] E.A. Emerson and J.Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *Proceedings of the 14th ACM Symp. Theory of Computing (STOC'82)*, pages 169–180, San Francisco, CA, USA, May 1982.

[Eme90] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, volume B : Formal Models and Semantics*, pages 995–1072. Elsevier, 1990.

[EN95] E.A. Emerson and K.S. Namjoshi. Reasoning about rings. In *Proceedings of the 22^{nd} symposium on Principles of programming Languages (POPL'95)*, pages 85–94, San Francisco, California, USA, 1995. ACM Press.

[Esp97] J. Esparza. Petri nets, commutative context-free grammars and basic parallel processes. *Fundamenta Informaticae*, 31(1) :13–25, 1997.

[Gar98] H. Garavel. OPEN/CAESAR : An Open Software Architecture for Verification, Simulation and Testing. In Bernhard Steffen, editor, *Proceedings of 1^{st} International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, Lisboa, Portugal, March 1998.

[GFL05] F. Gervais, M. Frappier, and R. Laleau. Vous avez dit raffinement ? Technical Report 829, CEDRIC-CNAM, March 2005.

[GHG+93] J. Guttag, J. Horning, S. Garland, K. Jones, A. Modet, and J. Wing. *Larch : languages and tools for formal specification.* Springer-Verlag, 1993.

[Gla90] R.J. van Glabbeck. The Linear Time - Branching Time Spectrum. In *Proceedings of the 1^{st} international Conference on Concurrency Theory (CONCUR'90)*, volume 458 of *Lecture Notes in Computer Science*, pages 278–297, Amsterdam, Netherlands, 1990. Springer-Verlag.

[Gla93] R.J. van Glabbeck. The Linear Time - Branching Time Spectrum II ; The semantics of sequential systems with silent moves. In *Proceedings of 4^{th} international Conference on Concurrency Theory (CONCUR'93)*, volume 715 of *Lecture Notes in Computer Science*, pages 66–81, Hildesheim, Germany, august 1993. Springer-Verlag.

[GM93] M.J.C. Gordon and T.F. Melham, editors. *Introduction to HOL (A theorem-proving environment for higher order logic).* Cambridge University Press, 1993.

[GMMP91] C. Ghezzi, D. Mandrioli, S. Morasca, and M. Pezze. A unified high-level Petri net formalism for time-critical systems. *IEEE Transactions on Software Engineering*, 17 :160–172, February 1991.

[Gog79] J. Goguen. Some design principles and theory for OBJ-0, a language for expressing and executing algebraic specifications of programs. In *Proceedings of the Conference on Mathematical Studies of Information Processing*, volume 75 of *Lecture Notes in Computer Science*, pages 425–473. Springer-Verlag, 1979.

[GOO04] S. Graf, I. Ober, and I. Ober. Model-checking of UML models via a mapping to communicating extended timed automata. In *Proceedings of the 11^{th} International SPIN Workshop on Model Checking of Software*, volume 2989 of *Lecture Notes in Computer Science*. Springer-Verlag, 2004.

[GRR03] G. Gardey, O. (H) Roux, and O. (F) Roux. A zone-based method for computing the state-space of a time Petri net. In *Proceedings of the 1^{st}*

International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS'03, volume 2791 of *Lecture Notes in Computer Science*, pages 246–259, Marseille, France, September 2003. Springer-Verlag.

[GRR06] G. Gardey, O. (H) Roux, and O. (F) Roux. State space computation and analysis of time Petri nets. *Theory and practice of Logic Programming - Special Issue on Specification Analysis and Verification of Reactive Systems*, 6(3) :301–320, 2006.

[GS03] G. Gössler and J. Sifakis. Component-Based Construction of Deadlock-Free Systems. In *Proceedings of the 23rd Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS'03)*, volume 2914 of *Lecture Notes in Computer Science*, pages 420–433, Mumbai, India, December 2003. Springer-Verlag.

[GS05] G. Gössler and J. Sifakis. Composition for Component-Based Modeling. *Science of Computer Programming*, 55 :161–183, March 2005.

[Hen96] T.A. Henzinger. The Theory of Hybrid Automata. In *Proceedings of the* 11th *Annual Symposium on Logic in Computer Science (LICS)*, pages 278–292. IEEE Computer Society Press, 1996.

[HH95] T.A. Henzinger and P.-H. Ho. Hytech : the cornell hybrid technology tool. In *Proceedings of Hybrid Systems II*, volume 999 of *Lecture Notes in Computer Science*, pages 265–293, Ithaca, NY, USA, 1995. Springer-Verlag.

[HHK95] M.R. Henzinger, T.A. Henzinger, and P.W. Kopke. Computing simulations on finite and infinite graphs. In *Proceedings of the 36th IEEE Symposium on Foundations of Computer Science*, pages 453–462, 1995.

[HHWT95] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. HyTech : the next generation. In *Proceedings of the* 16th *IEEE Real-Time Systems Symposium (RTSS'95)*, pages 56–65, Pisa, Italy, 1995. IEEE Computer Society Press.

[HHWT97] T.A. Henzinger, P.-H. Ho, and H. Wong-Toi. Hytech : a model-checker for hybrid systems. *Journal of Software Tools for Technology Transfer*, 1(1/2) :110–122, 1997.

[HJJ$^+$96] J.G. Henriksen, J.L. Jensen, M.E. Jorgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona : Monadic second-order logic in practice. In *Proceedings of the 2nd International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1019 of *Lecture Notes in Computer Science*, pages 89–110, Passau, Germany, 1996. Springer-Verlag.

[HK94] T. Henzinger and P.W. Kopke. Undecidability Results for Hybrid Systems. In *Proceedings of Workshop on Hybrid Systems and Autonomous Control*, 1994.

[HKPV98] T. Henzinger, P.W. Kopke, A. Puri, and P. Varaiya. What's Decidable about Hybrid Automata ? *Journal of Computer and System Sciences*, 57 :94–124, 1998.

[HNSY94] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model-Checking for Real-Time Systems. *Information and Computation*, 111 :193–244, 1994.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1985.

[Hol97a] G.J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5) :279–295, 1997.

[Hol97b] G.J. Holzmann. State compression in SPIN. In *Proceedings of the* 3^{rd} *SPIN Workshop*, April 1997.

[Hol98] G.J. Holzmann. An analysis of bitstate hashing. *Formal Methods in System Design*, 13(3) :287–305, 1998.

[HRS98] T.A. Henzinger, J.-F. Raskin, and R.-Y. Schobbens. The Regular Real-Time Languages. In *Proceedings of the* 25^{th} *International Colloquium on Automata, Languages and Programming (ICALP'98)*, volume 1443 of *Lecture Notes in Computer Science*, pages 580–591. Springer-Verlag, 1998.

[HSLL97] K. Havelund, A. Skou, K.G. Larsen, and K. Lund. Formal Modeling and Analysis of an Audio/Video Protocol : An Industrial Case Study Using UPPAAL. In *Proceedings of the* 18^{th} *IEEE Real-Time Systems Symposium*, pages 2–13, December 1997.

[IEE85] IEEE. *ANSI/IEEE 902/3, ISO/DIS 8802/3.* IEEE Computer Society Press, 1985.

[JLS00] H.E. Jensen, K.G. Larsen, and A. Skou. Scaling up UPPAAL : Automatic verification of real-time systems using compositionnality and abstraction. In *Proceedings of the* 6^{th} *international symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'00)*, pages 19–30, London, UK, 2000. Springer-Verlag.

[JM95] P. Jancar and F. Moller. Checking regular properties of Petri Nets. In *Proceedings of the* 6^{th} *International Conference on Concurrency Theory (CONCUR'95)*, volume 962 of *Lecture Notes in Computer Science*, pages 348–362, Philadelphia, Pennsylvania, USA, August 1995. Springer-Verlag.

[Jon90] C.B. Jones. *Systematic software development using VDM.* Prentice-Hall, 1990.

[KL03] O. Kouchnarenko and A/ Lanoix. Refinement and verification of synchronized component-based systems. In *Proceedings of the* 12^{th} *International Conference on Formal Methods (FM'03)*, volume 2805 of *Lecture Notes in Computer Science*, pages 341–358, Pisa, Italy, September 2003. Springer-Verlag.

[KL04] O. Kouchnarenko and A/ Lanoix. Verifying invariants of component-based systems through refinement. In *Proceedings of the* 10^{th} *International Conference on Algebraic Methodology and Software Technology (AMAST'04)*, volume 3116 of *Lecture Notes in Computer Science*, pages 289–303, Stirling, Scotland, July 2004. Springer-Verlag.

[Lan05] A. Lanoix. *Systèmes à composants synchronisés : contributions à la vérification compositionnelle du raffinement et des propriétés.* PhD thesis, Université de Franche-Comté, 2005.

[LL95] F. Laroussinie and K.G. Larsen. Compositional model checking for real-time systems. In *Proceedings of the 6th International Conference on Concurrency Theory (CONCUR'95)*, volume 962 of *Lecture Notes in Computer Science*, pages 27–41, Philadelphia, USA, August 1995. Springer-Verlag.

[LL98] F. Laroussinie and K.G. Larsen. CMC : a tool for compositional model-checking of real-time systems. In *Proceedings of IFIP joint International Conference on Formal Description Techniques and Protocol Specification, Testing and Verification (FORTE-PSTV'98)*, pages 439–456. Kluwer Academic, 1998.

[LPY97a] K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Journal on Software Tools for Technology Transfer*, 1(1–2) :134–152, October 1997.

[LPY97b] K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL : Status and Developments. In Orna Grumberg, editor, *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV'97)*, volume 1254 of *Lecture Notes in Computer Science*, pages 456–459. Springer-Verlag, Jun 1997.

[LR03] D. Lime and O. (H) Roux. State class timed automaton of a time petri net. In *Proceedings of the 10th International Workshop on Petri Nets and Performance Models (PNPM'03)*, September 2003.

[LT93] N.G. Leveson and C.S. Turner. Investigation of the Therac-25 Accidents. *IEEE Computer*, 26(7) :18–41, 1993.

[McM93] K.L. McMillan. *Symbolic Model-Checking.* Kluwer Academic Publishers, 1993.

[MF76] P. Merlin and D.J. Farber. Recoverability of communication protocols. *IEEE Transactions on Communications*, 24(9) :1036–1043, September 1976.

[Mil89] R. Milner. *Communication and Concurrency.* Prentice Hall, 1989.

[MLS06] I. Mouakher, A. Lanoix, and J. Souquières. Component adaptation : Specificaton and verification. In *Proceedings of the 11st International Workshop on Component-Oriented Programming (WCOP'06)*, Nantes, France, July 2006.

[MT90] F. Moller and C. Tofts. A temporal calculus of communicating systems. In *Proceedings of the 1st International Conference on Concurrency Theory (CONCUR'90)*, volume 458 of *Lecture Notes in Computer Science*, pages 401–415, Amsterdam, Netherlands, 1990. Springer-Verlag.

[ORS92] S. Owre, J.M. Rushby, and N. Shankar. PVS : a prototype verification system. In *Proceedings of the 11th International Conference on Automated Deduction (CADE'92)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[Pel96] D. Peled. Combining partial order reductions with on-the-fly model-checking. *Formal Methods in System Design*, 8(1) :39–64, 1996.

[Pnu81] A. Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13 :1–20, 1981.

[QS82] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems is CESAR. In *Proceedings of the International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 337–351, 1982.

[RR88] G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58(1-3) :249–261, 1988.

[SBB+99] P. Schnoebelen, B. Bérard, M. Bidoit, F. Laroussine, and A. Petit. *Vérification de logiciels - Techniques et outils du model-checking.* Vuibert, 1999. ISBN 2-7117-8646-3.

[Sch99] S. Schneider. *Concurrent and Real-Time Systems : The CSP Approach.* Wiley, 1999.

[Sif77] J. Sifakis. Use of petri nets for performance evaluation. In *Proceedings of the 3^{rd} International Symposium on Measuring, modeling and evaluating computer systems*, pages 75–93. North-Holland, 1977.

[Sif05] J. Sifakis. A framework for Component-based construction. In *Proceedings of the 4^{st} IEEE International Conference on Software Engineering and Formal Methods (SEFM'05)*, pages 293–300, September 2005. keynote talk.

[Spi87] J. Spivey. *Understanding Z : a specification language and its formal semantics.* Cambridge University Press, 1987.

[SY96] J. Sifakis and S. Yovine. Compositional Specification of Timed Systems. In *Proceedings of the 13^{th} Annual Symp. on Theoretical Aspects of Computer Science (STACS'96) - Invited Paper*, volume 1046 of *Lecture Notes in Computer Science*, pages 347–359, 1996.

[TAKB96] S. Tasiran, R. Alur, R.P. Kurshan, and R.K. Brayton. Verifying Abstractions of Timed Systems. In *Proceedings of the 7^{th} Conference on Concurrency Theory (CONCUR'96)*, volume 1119 of *Lecture Notes in Computer Science*, pages 546–562, Pisa, Italy, 1996.

[Tan89] A.S. Tanenbaum. *Computer Networks.* Prentice Hall, 1989.

[Tas98] S. Tasiran. *Compositional and Hierarchical Techniques for the Formal Verification of Real-Time Systems.* PhD thesis, University of California at Berkeley, 1998.

[Tea01] The Coq Development Team. *The Coq Proof Assistant Reference Manual Version 7.2.* INRIA-Rocquencourt, December 2001. http ://coq.inria.fr/doc-eng.html.

[Tho90] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, volume B : Formal Models and Semantics*, pages 133–192. Elsevier, 1990.

[Tri98] S. Tripakis. *The analysis of timed systems in practice.* PhD thesis, Universite Joseph Fourier, Grenoble, France, December 1998.

[TY96] S. Tripakis and S. Yovine. Analysis of timed systems based on time-abstracting bisimulations. In *Proceedings of the 8th Conference on Computer-Aided Verification (CAV'96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 232–243, New Brunswick, New Jersey, USA, July 1996. Springer-Verlag.

[TY01] S. Tripakis and S. Yovine. Analysis of Timed Systems using Time-abstracting Bisimulations. *Formal Methods in System Design*, 18(1) :25–68, January 2001.

[TYB05] S. Tripakis, S. Yovine, and A. Bouajjani. Checking Timed Büchi Automata Emptiness Efficiently. *Formal Methods in System Design*, 26(3) :267–292, May 2005.

[Yi90] W. Yi. Real-Time behavior of asynchronous agents. In *Proceedings of the 1st international Conference on Concurrency Theory (CONCUR'90)*, volume 458 of *Lecture Notes in Computer Science*, pages 502–520, Amsterdam, Netherlands, 1990. Springer-Verlag.

[Yov97] S. Yovine. KRONOS : A verification tool for real-time systems. *Journal of Software Tools for Technology Transfer*, 1(1/2) :123–133, October 1997.

[Yov98] S. Yovine. Model Checking Timed Automata. In G. Rozenberg and F. Vaandrager, editors, *Lectures on Embedded Systems*, volume 1494 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.

[YS97] D.D.M. Yellin and R.E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2) :292–333, 1997.

Résumé

Ces travaux se placent dans le contexte de la vérification par model-checking des systèmes temporisés modélisés par des automates temporisés dans un cadre compositionnel. Nous considérons la logique MITL comme formalisme de description des propriétés de ces systèmes. Le problème d'explosion combinatoire inhérent à l'utilisation du model-checking rend cette méthode difficile à appliquer en pratique sur des systèmes de grande taille, notamment en ce qui concerne la vérification des propriétés temporelles linéaires, et en particulier les propriétés de vivacité. Ces difficultés sont encore accentuées dans le cadre temporisé par la présence des contraintes de temps.
Une issue est d'utiliser des méthodes de développement incrémental, c'est-à-dire, dans le cadre des systèmes à composants, le raffinement ou l'intégration de composants. De telles méthodes consistent à modéliser progressivement un système. Du point de vue de la vérification, des propriétés doivent pouvoir être vérifiées à chaque étape du processus de modélisation, plutôt que directement sur le modèle complet du système obtenu à la fin du processus. Ce dernier point n'a d'intérêt que si les propriétés établies à une certaine étape sont préservées au fur et à mesure du développement.
Les τ-simulations sont un moyen de garantir la préservation de propriétés. Ce document présente deux τ-simulations adaptées aux systèmes temporisés : une τ-simulation temporisée classique préservant les propriétés de sûreté, et une seconde τ-simulation temporisée, dite sensible à la divergence et respectant la stabilité, préservant toutes les propriétés MITL, en particulier les propriétés de vivacité. Les propriétés d'absence de blocages et de non-zénonisme sont également préservées par cette dernière relation. Dans le cadre compositionnel dans lequel nous nous situons, les propriétés de compositionnalité, compatibilité et composabilité des τ-simulations vis-à-vis des opérateurs de composition sont des propriétés essentielles. Elles permettent d'établir ou non que les τ-simulations sont appropriées au développement incrémental de tels systèmes. Nous étudions donc ces propriétés vis-à-vis de deux opérateurs de composition, le premier utilisant un paradigme de composition *à la CSP*, et le second se rapprochant du paradigme de CCS et considérant de plus des priorités entre actions.
Pour montrer l'intérêt de ces simulations en pratique, nous avons réalisé le prototype VESTA. VESTA se focalise sur un développement incrémental des systèmes temporisés par intégration de composants. Il permet de garantir la préservation des propriétés MITL en vérifiant la τ-simulation temporisée sensible à la divergence et respectant la stabilité. VESTA a de plus été concu à la manière d'OPEN-KRONOS afin de pouvoir connecter les modèles considérés à la plate-forme de vérification OPEN-CAESAR.

MOTS-CLÉS : τ-simulation temporisée, systèmes temporisés à composants, vérification incrémentale, préservation de propriétés.

Abstract

We are interested in the verification of timed systems modeled as timed automata in a compositional framework, using model-checking. The properties of these systems are expressed in the logical formalism MITL. Model-checking is a verification method which is difficult to apply on large-sized systems, due to an exponential blow-up of the state space. This is especially the case for the verification of linear-time properties, and in particular for liveness properties. This problem is largely accentuated in the case of timed systems, with the presence of timing constraints modeling the time elapsing.
A way out is to use incremental development methods, i.e., for component-based systems, either refinement or integration of components. Such methods consist in achieving a progressive modeling of the system. From a verification point of view, they must give the possibility to check properties at each step of the modeling process, instead of performing the verification on the model obtained at the end of the process. This framework is applicable only if already established properties are preserved by the development process.
A way to guarantee this preservation is to use τ-simulation relations. This thesis presents two such relations for timed systems : a classic timed τ-simulation preserving safety properties, and a second one, called divergence sensitive and stability respecting, to handle the preservation of all MITL properties, and in particular liveness properties. Deadlock-freedom properties as well as non-zenoness are also preserved by this second relation. In compositional frameworks, the following properties of the τ-simulations are essential w.r.t. composition operators : compositionality, compatibility and composability. They show the interest of the τ-simulations for incremental development. Thus, we study these properties w.r.t. two composition operators : one using a CSP composition paradigm, and the other closer to the CCS paradigm and using priorities between actions.
To show the interest in practice of the τ-simulations, we developped the prototype VESTA. VESTA focuses on an incremental development process of timed systems achieved by integration of components. It allows to guarantee the preservation of MITL properties by verifying the divergnece-sensitive and stability-respecting timed τ-simulation. Moreover, the design of VESTA was inspired by the tool OPEN-KRONOS, to give the possibility to connect the models considered in VESTA to the verification platform OPEN-CAESAR.

KEY-WORDS : timed τ-simulation, component-based timed systems, incremental verification, preservation of properties.