Implémentabilité des automates temporisés

**Karine ALTISEN[*] — Nicolas MARKEY[**] —
Pierre-Alain REYNIER[**] — Stavros TRIPAKIS[*]**

[*] *VERIMAG – Centre Équation*
2, avenue de Vignates
38610 GIÈRES, *FRANCE*

e-mail: {Karine.Altisen,Stavros.Tripakis}@imag.fr

[**] *LSV – ENS Cachan & CNRS*
61, avenue du Président Wilson
94235 CACHAN *Cedex, FRANCE*

e-mail: {Nicolas.Markey,Pierre-Alain.Reynier}@lsv.ens-cachan.fr

Article rédigé dans le cadre du projet CORTOS de l'ACI « Sécurité Informatique ».
Site web : http://www.lsv.ens-cachan.fr/aci-cortos/

RÉSUMÉ. *Dans ce papier, nous présentons le problème de l'*implémentabilité *des automates temporisés. Le cadre théorique des automates temporisés ne peut pas être reproduit fidèlement en pratique, sur des ordinateurs digitaux et plus ou moins imprécis ; les propriétés vérifiées mathématiquement sur l'automate pourraient ne plus être satisfaite par l'implémentation. Nous présentons deux approches pour étudier ce problème : l'une est basée sur une modélisation du programme s'exécutant sur la plate-forme, l'étude des propriétés se faisant sur ce modèle, et l'autre propose une sémantique élargie de l'automate temporisé, l'élargissement tenant compte des nouveaux comportements induits par la plate-forme.*

ABSTRACT. *In this paper, we present the problem of the implementability of timed automata. The theoretical semantics of timed automata can not be exactly implemented in practice, because computers are digital and more or less precise; the properties verified on a timed automaton are not necessarily preserved when implementing it. We deal with two approaches: the first one is based on the modeling of the execution platform and the second studies an enlarged semantics for timed automata that takes the imprecision into account.*

MOTS-CLÉS : *Vérification, model-checking, automates temporisés, implémentabilité*

KEYWORDS: *Verification, model-checking, timed automata, implementability*

1. Introduction

La vérification formelle des systèmes critiques a connu un succès incontestable depuis la fin des années 1970. Au cours des quinze dernières années, ces techniques de vérification ont été étendues afin de prendre en compte des notions quantitatives, permettant en particulier de spécifier le délai qui sépare différentes actions du système. Dans ce cadre, le modèle des automates temporisés [ALU 94] a été particulièrment bien étudié, et plusieurs outils de vérification ont été développés pour ces modèles. Sans entrer dans les détails pour l'instant, un automate temporisé est un système de transitions disposant d'*horloges*, qui évoluent continûment, toutes à la même vitesse, et qui servent à spécifier quand une transition est autorisée.

On peut cependant se poser la question de la pertinence de la vérification formelle de ces objets mathématiques, dès lors qu'ils représentent des programmes : la sémantique de ce modèle est en effet extrêmement précise et suppose en particulier des transitions immédiates et des horloges infiniment précises. Aucune plate-forme d'exécution ne permet d'implémenter aussi précisément un tel automate, et rien ne permet d'assurer *a priori* que les propriétés vérifiées sur le modèle théorique seront préservées lors de l'implémentation.

Dans ce papier, nous exposons ce problème sur l'exemple du protocole d'exclusion mutuelle de Fischer, et présentons quelques techniques récentes permettant de prendre en compte les imprécisions des systèmes réels.

2. De la théorie à la mise en œuvre

2.1. *Motivations*

Un automate temporisé est défini par un quintuplet $\mathcal{A} = (L, \ell_0, \mathcal{C}, \Sigma, \delta)$ où L est un ensemble fini d'*états*, $\ell_0 \in L$ est l'état initial, $\mathcal{C}$ est un ensemble fini d'*horloges*, Σ est un alphabet fini, et δ est l'ensemble des transitions. Une transition $\ell \xrightarrow[r:=0]{a,g} \ell'$ est tirable si l'on se trouve dans l'état ℓ, qu'on exécute l'action a, et que les horloges satisfont la condition g, qui est une conjonction de conditions $x \sim n$, où x est une horloge, n est un entier, et $\sim\ \subseteq \{<, \leq, = \geq, >\}$. On passe alors dans l'état ℓ', et les horloges de $r \subseteq \mathcal{C}$ sont remises à zéro. Au lieu de prendre une transition, il est également possible, à tout moment, de laisser s'écouler un délai t, auquel cas on reste dans la même location et les horloges sont augmentées de la valeur t.

Prenons l'exemple du protocole d'exclusion mutuelle de Fischer [KRI 97] : deux systèmes S_1 et S_2 souhaitent accéder à une section critique ; ils sont supervisés par deux contrôleurs chargés d'assurer que les deux systèmes n'accèderont pas simultanément à la section critique. La figure 1 représente un système et son contrôleur.

On peut facilement montrer, grâce à un outil de vérification d'automates temporisés comme Uppaal [BEH 04], HyTech [HEN 97] ou Kronos [DAW 96], que l'exclusion mutuelle est bien réalisée par ces contrôleurs.

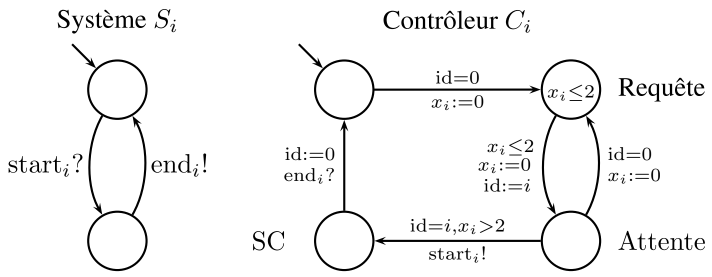

Figure 1. *Le protocole d'exclusion mutuelle de Fischer*

2.2. *Des automates temporisés aux plates-formes d'exécution*

La propriété d'exclusion mutuelle de notre exemple est cependant étroitement liée au caractère infiniment précis (*i.e.* exact) du modèle des automates temporisés qui le composent. En particulier, elle repose sur la garde stricte $x_i > 2$. Celle-ci assure en effet, combinée avec l'invariant $x_i \leq 2$ de l'état Requête, que lorsqu'un processus P_i peut atteindre sa section critique depuis l'état $\mathrm{Attente}$, aucun autre processus P_j ne peut être dans l'état Requête. Ainsi, relâcher la contrainte $x_i > 2$ en la contrainte $x_i \geq 2$ fait perdre la correction du système, ce qui prouve sa fragilité. Par exemple, si la précision des horloges est finie, ou bien si les horloges ne sont pas parfaitement synchronisées, ou bien encore si des délais de communication entre les processus, ou des délais de lecture/écriture des variables partagées viennent à ralentir le système, il est possible que les transitions soient franchies à des instants interdits dans le modèle mathématique. Or de telles imprécisions, aussi petites soient-elles, existent toujours dans un système réel et peuvent entraîner la violation de la propriété d'exclusion mutuelle.

L'exemple précédent met en évidence un certain nombre de propriétés liées au caractère « idéal » du modèle mathématique des automates temporisés qu'il est impossible de reproduire dans un système réel :

1) synchronisation parfaite des différentes composantes du système, réception en temps nul des entrées, émission en temps nul des sorties,

2) précision infinie des horloges,

3) temps d'exécution des actions nul — vitesse infinie du système lors de l'évaluation des gardes des transitions, et du choix de l'action à déclencher.

Pour mettre en défaut ces caractéristiques idéales, nous considérons des plateforme d'exécution réalistes pour lesquelles il faudra préciser :

1) la gestion des entrées/sorties du système vis-à-vis de l'environnement, et/ou des variables partagées si le système est distribué ; les délais nécessaires à ces opérations,

2) la précision de l'horloge globale du système, et des différentes horloges les unes par rapport aux autres si le système est distribué,

3) la vitesse, la fréquence et la précision des calculs du système.

2.3. *Besoins de garanties sur l'implémentation*

Une fois l'étude du modèle terminée, on a en général un automate temporisé qui vérifie formellement un certain nombre de propriétés. Il semble alors assez légitime d'attendre que ces propriétés, prouvées sur le modèle, soient encore satisfaites par l'implémentation. On parle de propriétés *préservées* par l'implémentation. Plus précisément, si l'automate temporisé $\mathcal{A}$ a été développé sous l'hypothèse d'un environnement pour vérifier une propriété logique ϕ, alors l'implémentation de $\mathcal{A}$ préserve ϕ si le programme codant $\mathcal{A}$, s'exécutant sur une certaine plate-forme et dans le même environnement vérifie encore ϕ.

La préservation de ces propriétés selon la remarque précédente est loin d'être immédiate. Une grande partie du travail autour du thème de l'implémentabilité va alors consister à identifier tout ou partie des cas pour lesquels il y a préservation de propriétés. Cela peut porter sur des conditions sur la plate-forme d'exécution et/ou des conditions sur le programme implémentant l'automate temporisé et/ou des conditions sur la propriété à préserver.

Une autre propriété qu'il est assez naturel d'attendre est la suivante : imaginons qu'un automate temporisé ait été implémenté sur une plate-forme P donnée et que le résultat soit satisfaisant (le programme s'exécutant dans un environnement donné sur la-dite plate-forme satisfait tout ce qu'il faut). Imaginons encore que nous changions la plate-forme P pour une plate-forme P' plus performante. On aimerait que « ça marche » encore ! (C'est la propriété « *faster is better* ».) Autrement dit, on aimerait ne pas avoir à refaire tout le travail de développement (implémentation de l'automate temporisé) puis de test/vérification/... éventuellement coûteux qui a conduit à accepter l'implémentation sur P ; on aimerait avoir des garanties pour que la même implémentation soit encore satisfaisante sur P'. Bien entendu, ce genre de garantie est étroitement liée à la signification de : « P' est plus performante que P ».

2.4. *Les solutions proposées*

Une première solution, naturelle s'il en est, est de chercher un pas de temps Δ pour la plate-forme d'exécution pendant lequel elle soit capable 1) de récupérer et transmettre les entrées, 2) de mettre à jour l'horloge du processeur, 3) d'effectuer ses calculs. Il s'agit ensuite de transformer l'automate temporisé en un automate dont les horloges sont non plus continues mais mises à jour toutes les Δ unités de temps. Cette discrétisation conduit alors à un automate interprété facilement programmable. Malheureusement, il n'existe pas toujours de pas de temps Δ tel que le comportement de l'automate interprété soit en tout point comparable à celui de l'automate temporisé [ALU 91]. Par conséquent, on ne peut pas trouver de plate-forme d'exécution pour laquelle cette implémentation préserve toute propriété.

Cette solution facile étant écartée, deux autres propositions sont en cours d'investigation pour résoudre le problème du passage à l'implémentation.

1) *Modélisation de la plate-forme d'exécution :* cette technique, présentée plus en détails dans [ALT 05], consiste à modéliser la plate-forme d'exécution (temps de transmission des entrées, dates de mises à jour de l'horloge, etc.) à l'aide d'automates temporisés. Cette approche est développée dans la section 3.

2) *Adaptation de la sémantique de l'automate temporisé :* cette approche, introduite dans [DEW 04b], consiste à modifier légèrement la sémantique des automates temporisés. Cette nouvelle sémantique *élargit* les gardes des automates temporisés, afin de prendre en compte un défaut de synchronisation des composants du système. Nous exposons cette étude dans la section 4.

3. L'approche « modélisation »

L'approche de [ALT 05] propose une méthode pour implémenter des automates temporisés dans un cadre permettant l'étude des garanties de l'implémentation. La méthode est basée sur la modélisation de la plate-forme d'exécution. D'une part, à partir de l'automate temporisé $\mathcal{A}$ à implémenter est produit automatiquement un automate discret $\mathrm{Prog}(\mathcal{A})$ représentant le programme à implémenter. D'autre part, la plate-forme d'exécution est modélisée à l'aide d'automates temporisés $\mathcal{P}$. La composition parallèle de $\mathcal{P}$ et $\mathrm{Prog}(\mathcal{A})$ modélise l'exécution du programme sur la plateforme. Le but est alors de comparer cette exécution avec la sémantique de $\mathcal{A}$ et d'en déduire la préservation éventuelle de propriétés.

3.1. *Le programme*

Dans le modèle, l'automate temporisé $\mathcal{A}$ est directement en contact avec l'environnement physique Env (ce que nous notons $\mathcal{A} \leftrightarrow \mathrm{Env}$). Les sorties de $\mathcal{A}$ arrivent directement à Env et inversement, le temps physique et les entrées pour $\mathcal{A}$ lui sont directement transmis. On peut voir le passage à une implémentation comme l'introduction d'un intermédiaire entre $\mathcal{A}$ et Env : on a alors $\mathcal{A} \leftrightarrow \mathcal{P} \leftrightarrow \mathrm{Env}$. La transmission du temps physique comme des entrées/sorties de $\mathcal{A}$ sont filtrées et/ou déformées par le passage *via* la plateforme. Nous considérons que la plate-forme transmet à l'implémentation de $\mathcal{A}$ les entrées qu'elle reçoit de Env et qu'elle lui fournit aussi deux entrées additionnelles : `now` représente l'horloge du processeur et stocke la valeur de la date courante, `trig` représente la vitesse d'exécution de la plateforme, elle sert à déclencher une nouvelle exécution du corps de la boucle externe du programme. Ceci définit l'interface du programme implémentant $\mathcal{A}$. Pour obtenir ce programme, $\mathcal{A}$ est transformé en un automate non temporisé, lequel est interprété par :

```
loop each trig :
    | read now ;
    | read inputs ;
    | compute a step in Prog(A) ;
    | write outputs ;
endloop.
```

L'automate $\mathrm{Prog}(\mathcal{A})$ est obtenu par transformation syntaxique de $\mathcal{A}$; un exemple est donné à la figure 2.

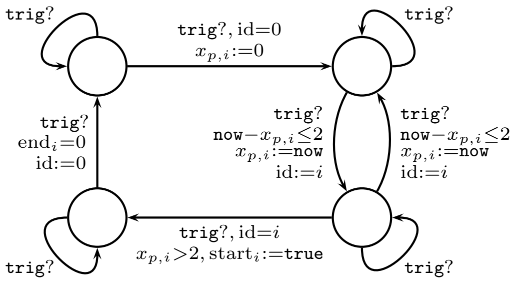

Figure 2. *L'automate* $\mathrm{Prog}(C_i)$*, où* C_i *est un contrôleur du protocole de Fischer illustré à la figure 1.*

3.2. *Modélisation de la plate-forme*

Pour étudier les bonnes propriétés de cette implémentation sur une plate-forme donnée, la plate-forme est modélisée selon le schéma de la figure 3 : les trois points identifiés au paragraphe 2.2 comme mettant en défaut les hypothèses idéales de la sémantique des automates temporisés sont isolés et chacun est modélisé par un automate temporisé.

1) Modèle de l'interface d'entrées/sorties : émet et reçoit les entrées/sorties depuis l'environnement ; transmet au programme les variables correspondantes. Pour exemple, il permet de représenter le délai entre le moment où une sortie est mise à jour par le programme et le moment où elle est émise vers l'environnement ; il peut aussi modéliser la politique de consommation des entrées : qu'advient-il des entrées non consommées, sont elles perdues, et si oui, au bout de combien de temps ? Sont-elles mises en mémoire ? (et quelle est la taille de cette mémoire ?)

2) Modèle d'horloge digitale : met à jour la variable `now`. Il peut représenter le rythme de mise à jour et la précision (peut-être non-déterministe) de l'horloge du processeur. Deux exemples sont fournis à la figure 4.

3) Modèle d'exécution : émet l'événement `trig` qui déclenche l'exécution pour le programme d'un nouveau tour de boucle. Il peut modéliser le pire cas d'exécution d'un tour de boucle, un temps non-déterministe entre le pire et le meilleur temps d'exécution, ou quelquechose de plus fin.

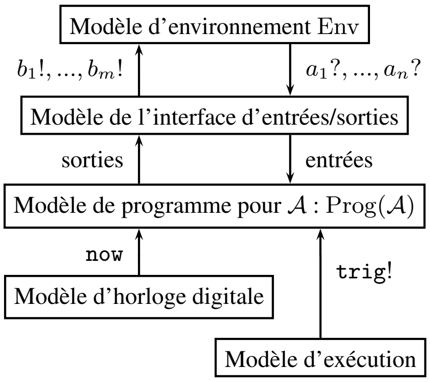

Fig. 3. *Modèle global d'exécution*

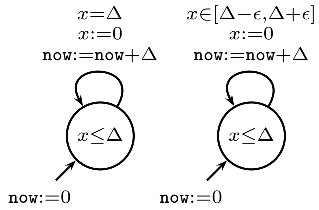

Fig. 4. *Deux modèles d'horloge digitale*

La composition du modèle de la plate-forme $\mathcal{P}$ (donnée par les trois modèles ci-dessus) et de $\mathrm{Prog}(\mathcal{A})$ donne lieu au *modèle global d'exécution*. Il représente l'exécution du programme qui implémente $\mathcal{A}$ sur une plate-forme qui se comporte comme indique $\mathcal{P}$. Le modèle global peut être utilisé pour vérifier si l'implémentation de $\mathcal{A}$ satisfait les propriétés souhaitées. Ceci peut être fait par exemple à l'aide d'un outil de model-checking tel que Kronos ou Uppaal.

Une propriété plus ambitieuse que l'on souhaiterait pour un cadre d'implémentation est la propriété « *faster is better* » mentionnée ci-dessus, qui consiste à dire que si le modèle d'exécution global pour la plate-forme $\mathcal{P}$ satisfait une propriété ϕ, alors, quand $\mathcal{P}$ est remplacée par une « meilleure » plate-forme $\mathcal{P}'$, le modèle d'exécution global pour $\mathcal{P}'$ satisfait encore ϕ. Il reste à définir formellement la notion de « meilleure ». Ceci n'est pas trivial : [ALT 05] montre que, pour une définition assez raisonnable de « meilleure », notamment, qui considère $\mathcal{P}'$ meilleure que $\mathcal{P}$ si les deux sont identiques hormis le fait que $\mathcal{P}'$ fournit une horloge deux fois plus rapide que $\mathcal{P}$, la propriété « *faster is better* » n'est pas vraie. Ceci est dû au fait qu'une horloge plus rapide permet un meilleur « échantillonnage » et par conséquent génère plus de comportements en général.

4. L'approche « sémantique »

Une façon de s'affranchir d'une partie de ces difficultés est de fixer un modèle particulier de plate-forme plus simple, sur lequel on pourra raisonner et étudier la correction de l'implémentation. L'approche « sémantique » est donc nettement moins expressive que l'approche « modélisation », mais permet de vérifier formellement qu'*il existe* une plate-forme assez rapide sur laquelle l'implémentation sera correcte.

4.1. *La plate-forme*

Nous considérons le modèle de plate-forme proposé dans [DEW 04b] : il possède une horloge digitale qui est mise à jour toutes les Δ_P unités de temps, et un processeur central, similaire à celui proposé dans la section précédente, qui effectue en boucle le cycle d'actions suivant :

– lire la valeur de l'horloge ;

– lire les entrées ;

– calculer les valeurs des gardes de l'automate ;

– exécuter une des transitions autorisées, le cas échéant.

Pour prendre en compte le temps de calcul, on considère qu'un tel cycle peut durer jusqu'à Δ_L unités de temps.

4.2. *De l'implémentabilité à la robustesse*

[DEW 04b] étudie alors les propriétés théoriques de ce type de plate-forme en établissant le lien avec une sémantique élargie des automates temporisés :

1) la sémantique $[\![\mathcal{A}]\!]^{\mathrm{Prog}}_{\Delta_L,\Delta_P}$ associée à ce type de plate-forme, étant données les valeurs des deux paramètres Δ_L et Δ_P représente l'exécution de l'implémentation de $\mathcal{A}$ sur cette plate-forme de paramètre Δ_L, Δ_P.

2) la sémantique AASAP, pour « Almost ASAP », et notée $[\![\mathcal{A}_\Delta]\!]$, consiste à introduire un certain délai Δ dans le comportement de l'automate. Grossièrement, cela revient à élargir les gardes du paramètre Δ, *i.e.* à remplacer une garde $x \in [a, b]$ par la nouvelle garde $x \in [a - \Delta, b + \Delta]$.

Les auteurs de [DEW 04b] ont montré que si les paramètres Δ, Δ_L et Δ_P vérifient l'inégalité $(1) \quad \Delta > 3\,\Delta_L \;+\; 4\,\Delta_P$, alors la sémantique $[\![\mathcal{A}]\!]^{\mathrm{Prog}}_{\Delta_L,\Delta_P}$ du programme implémentant $\mathcal{A}$ est simulée par la sémantique AASAP $[\![\mathcal{A}_\Delta]\!]$ de $\mathcal{A}$, au sens où tous les comportements de $[\![\mathcal{A}]\!]^{\mathrm{Prog}}_{\Delta_L,\Delta_P}$ existent, à équivalence près, dans $[\![\mathcal{A}_\Delta]\!]$.

Étant donnée une propriété ϕ désirée sur l'implémentation, le problème de l'existence d'un plate-forme implémentant l'automate $\mathcal{A}$ et satisfaisant la propriété ϕ se ramène donc à celui de l'existence d'un paramètre $\Delta > 0$ tel que la sémantique $[\![\mathcal{A}_\Delta]\!]$ soit satisfaisante. On dit alors que l'automate $\mathcal{A}$ satisfait de façon «robuste» la propriété ϕ. Intuitivement, le terme *robuste* s'oppose à la fragilité constatée au paragraphe 2.2 de la sémantique théorique des automates temporisés : malgré les perturbations induites par la plate-forme, et obtenues à travers l'introduction du paramètre Δ, le système satisfait encore la propriété. La vérification *robuste* d'une propriété consiste alors à décider de l'existence d'un tel Δ. Notons que la propriété « *faster is better* » est trivialement satisfaite par ce choix de plate-forme.

4.3. *Vérification robuste de propriétés*

Au lieu de vérifier une propriété ϕ sur un modèle $\mathcal{A}$, nous cherchons donc maintenant à prouver l'existence d'une valeur du paramètre Δ pour laquelle la sémantique élargie $[\![\mathcal{A}_\Delta]\!]$ de l'automate vérifie ϕ.

Ce problème a été résolu dans [DEW 04a] pour les *propriétés simples de sûreté*, qui expriment que le système ne va jamais entrer dans un ensemble d'états indésirables I. Cette solution repose sur l'équivalence suivante : il existe une valeur de Δ telle qu'aucun état de I n'est accessible dans la sémantique $[\![\mathcal{A}_\Delta]\!]$ si, et seulement si,

$$\bigcap_{\Delta>0} \mathsf{Acc}(\mathcal{A}_\Delta) \cap I = \varnothing$$

où $\mathsf{Acc}(\mathcal{A}_\Delta)$ représente l'ensemble des états accessibles de $[\![\mathcal{A}_\Delta]\!]$. L'intersection des ensembles d'états accessibles qui intervient dans cette équivalence, et que nous noterons $\mathsf{Acc}^*(\mathcal{A})$ par la suite, est bien définie, puisque l'ensemble $\mathsf{Acc}(\mathcal{A}_\Delta)$ est une fonction croissante de Δ. Il représente l'ensemble des états accessibles de $[\![\mathcal{A}_\Delta]\!]$, aussi petit que soit Δ.

Avec la sémantique classique des automates temporisés, les algorithmes sont basés sur une relation d'équivalence d'indice fini, dont les classes d'équivalence sont appelées *régions*. Celle-ci permet la construction d'un automate, appelé *automate des régions*, utilisé pour calculer l'ensemble des états accessibles [ALU 94].

Pour calculer $\mathsf{Acc}^*(\mathcal{A})$, on étend cette construction pour qu'elle tienne compte de l'inexactitude des horloges. Intuitivement, deux phénomènes peuvent ajouter des états dans $\mathsf{Acc}^*(\mathcal{A})$:

– les trajectoires qui passent à la frontière de gardes ouvertes : c'est ce qui se passe dans l'exemple du protocole de Fischer ;

– les trajectoires cycliques de l'automate, qui permettent d'accumuler les imprécisions, si petites soient-elles. Cet aspect, qui n'est pas présent dans notre exemple du protocole de Fischer, est présenté sur un exemple réel dans [DEW 05].

Dans [DEW 04a], il est montré que, précisément, $\mathsf{Acc}^*(\mathcal{A})$ est l'ensemble des état accessibles dans une version étendue de l'automate des régions, dans laquelle on « ferme » les régions et on ajoute des transitions supplémentaires correspondant à ces trajectoires cycliques. Cet algorithme permet de s'assurer de la *sûreté robuste* d'un automate temporisé. Récemment, il a été étendu à la vérification de propriétés de LTL (équité, vivacité, ...), ainsi qu'à quelques exemples de propriétés temporisées (comme par exemple la propriété de « temps de réponse borné ») [BOU 05].

5. Conclusion

Nous avons présenté dans ce papier le problème de l'implémentabilité des automates temporisés. Cela consiste, étant donné un automate temporisé obtenu à l'issue d'un travail de modélisation, à chercher un cadre d'implémentation. Nous avons montré en quoi le modèle mathématique des automates temporisés possède des hypothèses « idéales » et comment en pratique il faut prendre en compte les caractéristiques techniques des plates-formes d'exécution. Deux qualités sont alors souhaitables pour l'implémentation :

– la conservation des propriétés : si l'automate avait été conçu pour satisfaire une certaine propriété, alors son implémentation doit satisfaire également cette propriété.

– « *faster is better* » : si l'implémentation sur une plate-forme est satisfaisante, alors elle doit l'être également sur toute plate-forme plus rapide, plus performante.

Nous avons proposé deux points de vue pour aborder ce problème, avec deux approches différentes. Une approche dite de modélisation d'abord, dans laquelle l'exécution de l'automate sur une plate-forme à caractéristiques réelles est modélisé, ce qui permet de raisonner et/ou prouver que l'implémentation est satisfaisante. La plateforme est ici modélisée comme la composition d'automates temporisés, ce qui permet d'exprimer de nombreuses classes de plates-formes, mais soulève des questions théoriques, notamment pour les propriétés de type « faster is better ». Une approche dite sémantique ensuite, qui consiste à considérer un type de plate-forme particulier, ce qui permet de réduire le problème à la définition d'une nouvelle sémantique, dite *robuste*, des automates temporisés. Cela permet alors d'adapter les algorithmes de model-checking existants et d'obtenir des procédures pour les propriétés de sûreté, les propriétés LTL, ainsi que quelques propriétés temporisées.

Les deux approches présentées nous semblent prometteuses, et il reste encore de nombreuses voies à explorer. Ainsi, l'approche modélisation est très expressive, ce qui rend difficile pour l'instant l'obtention de résultats généraux. Pourtant, nous pensons qu'il est utile de poursuivre l'étude car une réponse complète offrirait des avantages indéniables en termes de modularité et de généralité par rapport aux autres approches. En particulier, nous souhaitons étudier la préservation de certaines classes de propriétés et développer la notion de comparaison de performance entre les plates-formes.

Dans l'approche sémantique, les résultats sont très encourageants, et nous souhaitons les étendre maintenant à l'ensemble de la logique MTL (une extension temporisée de la logique LTL). Nous espérons obtenir des algorithmes efficaces pour cette logique : la vérification de MTL a une complexité très élevée, mais il a été montré que cette complexité est fortement liée à la possibilité d'exprimer des conditions de ponctualité [ALU 96], ce qui n'est plus vraiment possible avec la sémantique élargie des automates.

Une suite logique à l'étude de l'implémentabilité des automates temporisés est celle de la génération de code exécutable. L'outil TIMES [AMN 02] permet, à partir d'automates temporisés, la génération de code avec tâches préemptables ; le but de l'outil est l'étude de l'ordonnançabilité du système plutôt que la préservation de propriétés fonctionnelles. Beaucoup de travaux existent sur la génération de code depuis des modèles de haut niveau autres que les automates temporisés, comme les automates hybrides [ALU 03], GIOTTO [HEN 01], les modèles Simulink/Stateflow[1] [CAS 03, SCA 04], ou les modèles synchrones [HAL 92], pour n'en citer que quelques-uns. Le travail présenté dans [KRČ 04] répond aux mêmes motivations que nous : il propose le modèle des automates *timed-triggered* pour représenter l'exécution sur une architecture *timed-triggered* [KOP 97], mais les délais d'exécution et de communication et la robustesse ne sont pas pris en compte dans ces travaux. Finalement, les travaux (par exemple [HEN 92, PUR 00, OUA 03]) sur la discrétisation et la robustesse des automates temporisés sont aussi corrélés à notre étude bien qu'ils se préoccupent plus de la préservation de propriétés sémantiques par des sémantiques discrètes et de l'application de tels résultats à la vérification.

6. Bibliographie

[ALT 05] ALTISEN K., TRIPAKIS S., « Implementation of Timed Automata : An Issue of Semantics or Modeling ? », rapport n° TR-2005-12, juin 2005, Verimag, Centre Équation, 38610 Gières, France.

[ALU 91] ALUR R., « Techniques for Automatic Verification of Real-Time Systems », PhD thesis, Stanford University, Stanford, CA, USA, 1991.

[ALU 94] ALUR R., DILL D., « A Theory of Timed Automata », *Theoretical Computer Science*, vol. 126, n° 2, 1994, p. 183–235, Elsevier Science.

[ALU 96] ALUR R., FEDER T., HENZINGER T. A., « The Benefits of Relaxing Punctuality », *Journal of the ACM*, vol. 43, n° 1, 1996, p. 116-146, ACM.

[ALU 03] ALUR R., IVANCIC F., KIM J., LEE I., SOKOLSKY O., « Generating Embedded Software from Hierarchical Hybrid Models », *Proceedings of the 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, ACM, 2003, p. 171-182.

[AMN 02] AMNELL T., FERSMAN E., PETTERSSON P., YI W., SUN H., « Code synthesis for timed automata », *Nordic Journal of Computing*, vol. 9, n° 4, 2002, p. 269-300.

[BEH 04] BEHRMANN G., DAVID A., LARSEN K. G., « A Tutorial on Uppaal », BERNARDO M., CORRADINI F., Eds., *Revised Lectures of the International School on Formal Methods for the Design of Real-Time Systems, (SFM-RT'04)*, vol. 3185 de *Lecture Notes in Computer Science*, Springer, septembre 2004, p. 200-236.

[BOU 05] BOUYER P., MARKEY N., REYNIER P.-A., « Robust Model-Checking of Timed Automata », rapport n° LSV-05-08, juin 2005, Lab. Spécification et Vérification, ENS Cachan, France.

[CAS 03] CASPI P., CURIC A., MAIGNAN A., SOFRONIS C., TRIPAKIS S., NIEBERT P., « From Simulink to SCADE/Lustre to TTA : A Layered Approach for Distributed Embedded Applications », *Proceedings of the 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*, ACM, 2003, p. 153-162.

[DAW 96] DAWS C., OLIVERO A., TRIPAKIS S., YOVINE S., « The Tool KRONOS », ALUR R., HENZINGER T. A., SONTAG E. D., Eds., *Proceedings of the 1995 DIMACS/SYCON*

1. Marque déposée par The Mathworks, Inc.

Workshop Hybrid Systems : Verification and Control, vol. 1066 de *Lecture Notes in Computer Science*, Springer, 1996, p. 208-219.

[DEW 04a] DE WULF M., DOYEN L., MARKEY N., RASKIN J.-F., « Robustness and Implementability of Timed Automata », LAKHNECH Y., YOVINE S., Eds., *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)*, vol. 3253 de *Lecture Notes in Computer Science*, Springer, 2004, p. 118-133.

[DEW 04b] DE WULF M., DOYEN L., RASKIN J.-F., « Almost ASAP Semantics : From Timed Models to Timed Implementations », ALUR R., PAPPAS G. J., Eds., *Proceedings of the 7th International Workshop on Hybrid Systems : Computation and Control (HSCC'04)*, vol. 2993 de *Lecture Notes in Computer Science*, Springer, 2004, p. 296-310.

[DEW 05] DE WULF M., DOYEN L., RASKIN J.-F., « Systematic Implementation of Real-Time Models », *Proceedings of the International Symposium of Formal Methods Europe (FM'05)*, Lecture Notes in Computer Science, Springer, 2005, À paraître.

[HAL 92] HALBWACHS N., *Synchronous Programming of Reactive Systems*, Kluwer, 1992.

[HEN 92] HENZINGER T., MANNA Z., PNUELI A., « What Good are Digital Clocks ? », KUICH W., Ed., *Proceedings of the 19th International Colloquiumon Automata, Languages and Programming (ICALP'92)*, vol. 623 de *Lecture Notes in Computer Science*, 1992.

[HEN 97] HENZINGER T. A., HO P.-H., WONG TOI H., « HyTech : A Model Checker for Hybrid Systems », *Software Tools for Technology Transfer*, vol. 1, 1997, p. 110-122.

[HEN 01] HENZINGER T. A., HOROWITZ B., KIRSCH C. M., « Giotto : A Time-Triggered Language for Embedded Programming », *Proceedings of the First International Workshop on Embedded Software (EMSOFT'01)*, vol. 2211 de *Lecture Notes in Computer Science*, Springer, 2001.

[KOP 97] KOPETZ H., *Real-Time Systems Design Principles for Distributed Embedded Applications*, Kluwer, 1997.

[KRČ 04] KRČÁL P., MOKRUSHIN L., THIAGARAJAN P., YI W., « Timed vs Time Triggered Automata », GARDNER P., YOSHIDA N., Eds., *Proceedings of the 15th International Conference on Concurrency Theory (CONCUR'04)*, vol. 3170 de *Lecture Notes in Computer Science*, Springer, 2004.

[KRI 97] KRISTOFFERSEN K. J., LAROUSSINIE F., LARSEN K. G., PETTERSSON P., YI W., « A Compositional Proof of a Real-Time Mutual Exclusion Protocol », BIDOIT M., DAUCHET M., Eds., *Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development (TAPSOFT'97)*, vol. 1214 de *Lecture Notes in Computer Science*, Lille, France, avril 1997, Springer, p. 565-579.

[OUA 03] OUAKNINE J., WORRELL J., « Revisiting Digitization, Robustness, and Decidability for Timed Automata », *Proceedings of the 18th Annual IEEE Symposium on Logics in Computer Science (LICS'03)*, IEEE Comp. Soc. Press, 2003.

[PUR 00] PURI A., « Dynamical properties of timed automata », *Discrete Event Dynamic Systems*, vol. 10, n° 1-2, 2000, p. 87-113, Kluwer.

[SCA 04] SCAIFE N., SOFRONIS C., CASPI P., TRIPAKIS S., MARANINCHI F., « Defining and translating a "safe" subset of Simulink/Stateflow into Lustre », *Proceedings of the 4th ACM International Conference on Embedded Software (EMSOFT'04)*, 2004.