Contrôle et implémentation des systèmes temporisés

Franck CASSEZ
IRCCyN
1, rue de la Noë – BP 92 101
44321 NANTES Cedex 3, France
franck.cassez@cnrs.irccyn.fr

Nicolas MARKEY
LSV – ENS Cachan & CNRS
61, avenue du Président Wilson
94235 CACHAN Cedex, France
Nicolas.Markey@lsv.ens-cachan.fr

Résumé

Cet article est une introduction au contrôle des systèmes modélisés par des automates temporisés. Nous définissons les notions de base du contrôle vu comme un jeu : stratégie, objectif de contrôle, états gagnants. On s'intéresse ensuite aux opérateurs de base permettant de résoudre les jeux temporisés. Nous considérons le cas des jeux de sûreté pour lesquels nous donnons un exemple complet de calcul de contrôleur. Nous abordons ensuite la notion importante d'implémentabilité des contrôleurs. Deux points de vue sont proposés : l'un syntaxique, qui consiste à décrire la plate-forme cible pour l'implémentation par des automates temporisés ; l'autre sémantique, qui considère une sémantique implémentable des automates temporisés.

Le contenu de cet article est basé sur les articles [1, 2].

1 Introduction

Modélisation et vérification des systèmes temps-réel. La plupart des appareils domestiques (machines à laver, téléphones, téléviseurs, lecteurs CD/DVD) que nous utilisons contiennent désormais de nombreux *logiciels*. Si des erreurs dans ces logiciels ne sont pas dommageables, il n'en va pas de même pour les logiciels qui contrôlent des centrales nucléaires ou des trajectoires d'avions, de fusées ou de voitures : dans ces systèmes, une défaillance logicielle conduit souvent à des dégâts importants ou des pertes en vies humaines. On parle alors de systèmes *critiques*, et il convient donc, lors du développement de telles applications, de s'assurer qu'elles satisfont un certain nombre de *propriétés*, notamment des propriétés de sûreté (*i.e.* l'absence de défaillance grave).

Une approche classique de vérification, le *model-checking* [42], consiste à construire un modèle complet S du comportement du système étudié (par exemple un automate fini), à formaliser la propriété de correction attendue par une formule ψ de logique (temporelle), puis à utiliser un algorithme (de model-checking) pour vérifier que S satisfait (ou non) ψ (ce qui est noté $S \models \psi$). Ces techniques sont maintenant bien connues et utilisées dans le domaine industriel. Cependant, pour certains types d'applications, il est nécessaire de prendre en compte des caractéristiques temporelles autres que le temps *logique* capturé par un modèle du type automate fini. Par exemple, pour des problèmes d'ordonnancement, les durées des tâches doivent être prises en compte explicitement. Il est parfois possible de se ramener à un modèle (en temps) discret mais cela peut s'avérer un facteur limitant : par exemple, si un système a des *constantes* de temps variant de 1 à 10000, considérer une horloge discrète de granularité 1 engendre un nombre d'états prohibitif pour les algorithmes de model-checking ; d'autre part, l'utilisation du temps discret suppose que l'on connaît exactement les durées des diverses opérations à réaliser. Pour prendre en compte une *incertitude* sur ces durées (durée des tâches, des communications), on utilise donc des modèles plus fins que les automates finis, par exemple les *automates temporisés* [8] où l'on peut utiliser des *horloges* pour spécifier le comportement temporel du système. Les algorithmes de model-checking utilisés sur les structures discrètes (automates finis, logique temporelle) ont été étendus aux modèles temporisés et ont conduit au développement d'outils de vérification dédiés tels que UPPAAL [9], KRONOS [46], CMC [37] ou encore HyTech [33] ou PHAver [32].

Le problème de la synthèse de contrôleur. Dans le cas du *model-checking*, on dit que les systèmes considérés sont *fermés* : on travaille sur un modèle complet du système incluant l'environnement à contrôler, les éventuels actionneurs et le contrôleur ; ce système évolue sans influence extérieure. Dans le cadre de la synthèse de contrôleurs, on part d'un système *ouvert*, le but étant de le fermer : si S est un modèle du système ouvert à contrôler, on lui ajoute un contrôleur C, la composition parallèle ($S \parallel C$) représentant alors le modèle complet ou *fermé* du système.

Si la propriété de correction à satisfaire est φ, les techniques de model-checking permettent de répondre à la

question « est-ce que S contrôlé par C (c'est-à-dire le système $S \parallel C$) satisfait φ ? ». Le problème de model-checking (MC) s'écrit alors formellement :

Étant donnés S, C et φ, est-ce que $(S \parallel C) \models \varphi$?

Cette approche nécessite donc d'abord de construire (éventuellement à la main) un contrôleur C, de manière à définir complètement le système, et ensuite à vérifier le système contrôlé $(S \parallel C)$. Ceci peut s'avérer difficile dans le cas de systèmes complexes ou/et avec des propriétés difficiles à mettre en œuvre (comme par exemple des propriétés dépendant du temps). De plus, si la propriété attendue n'est pas vérifiée, on doit modifier le contrôleur et vérifier de nouveau le système de façon itérative jusqu'à obtenir un contrôleur correct, ce qui peut être très long si un tel contrôleur n'existe pas .... Idéalement, on souhaiterait ne décrire que le système à contrôler et *calculer* (ou *synthétiser*) un contrôleur (s'il en existe un), de manière à ce que la spécification φ soit satisfaite. C'est la problématique du *contrôle*, plus générale que celle du model-checking. Le problème du *contrôle* (CP) pour les systèmes ouverts est formellement le suivant :

Étant donnés S et φ, existe-t-il C tel que $(S \parallel C) \models \varphi$?

Pour répondre au problème (CP) il est souvent nécessaire de restreindre la classe de modèles dans laquelle on va chercher un contrôleur. Par exemple on peut chercher des contrôleurs qui sont des automates finis (donc à mémoire bornée), ou bien des automates temporisés. Le problème naturel qui se pose après (CP) est celui de la *synthèse de contrôleur* (CSP) : Si la réponse à (CP) est « oui »,

Peut-on construire C tel que $(S \parallel C) \models \varphi$?

Implémentabilité des contrôleurs temporisés. Le but de la synthèse de contrôleur est évidemment d'aller encore un peu plus loin : il s'agit d'implanter (de manière aussi automatique que possible) le contrôleur synthétisé afin d'obtenir un système *réel* qui contrôle effectivement le système étudié.

Si le contrôleur est un automate fini, cette étape pose peu de difficultés. Dans le cas des automates temporisés, cette étape n'est pas aussi simple : les automates temporisés sont un modèle mathématique, très efficace pour les raisonnements formels, mais assez éloignés des systèmes informatiques réels. L'horloge d'un système informatique est digitale et n'a qu'une précision finie, et deux systèmes ne peuvent pas se synchroniser parfaitement.

De manière assez informelle pour l'instant, nous définissons le problème de l'*implémentabilité d'un contrôleur* (Impl) :

Peut-on implanter *fidèlement* le contrôleur C ?

Par « fidèlement », on entend ici que le *programme physique* doit lui aussi contrôler correctement le système étudié. En fait, cette question ne se pose pas uniquement dans le cas des contrôleurs : elle concerne les automates temporisés dans leur ensemble, et c'est dans ce cadre que nous l'étudierons, dans la deuxième partie de cet article.

Première partie

Contrôle des systèmes temporisés

Du contrôle aux jeux. Un problème de contrôle peut être formulé simplement dans le cadre de la *théorie des jeux* discrets [39, 44, 10, 41] ou temporisés [38, 12, 31, 25]. On peut modéliser un système ouvert par un automate temporisé *de jeu* (*Timed Game Automaton*, TGA) comme celui de la Figure 1. Un TGA est un TA avec les actions partitionnées en deux sous-ensembles disjoints : les actions *contrôlables* et *incontrôlables*. Par exemple le système ouvert de la Figure 1 comporte des transitions *contrôlables* (traits pleins) et *incontrôlables* (traits pointillés). Le type d'une transition est déterminé par son étiquette : on classe les actions de transitions $\{c_1, c_2, c_3, u\}$ en deux groupes, $\Sigma_c = \{c_1, c_2, c_3\}$ pour les actions contrôlables, et $\Sigma_u = \{u\}$ pour les actions incontrôlables. Ainsi une transition est contrôlable (resp. incontrôlable) si son étiquette est dans Σ_c (resp. Σ_u). Les protagonistes du jeu sont le contrôleur (qui détermine quand une action de Σ_c est faite) et l'environnement (qui peut faire des actions de Σ_u). Dans l'exemple précédent, l'objectif (pour le contrôleur) est d'éviter que le système aille dans l'état Bad, et ce, quelles que soient les actions (de Σ_u) de l'environnement. On parle dans ce cas d'un objectif de *sûreté* (*safety*).

Automates temporisés. Un automate temporisé (*Timed Automaton*, TA) possède des horloges prenant leur valeur dans $\mathbb{R}_{\geq 0}$ comme x dans l'exemple de la Figure 1. Les trajectoires définies par un tel automate commencent dans l'état initial $(\ell_0, x = 0)$. Le temps peut ensuite s'écouler d'une durée δ ($\in \mathbb{R}_{\geq 0}$) et le système atteint l'état $(\ell_0, x = \delta)$ (les horloges vont à la même vitesse que le temps physique). La *localité* ℓ_0 est contrainte par un invariant $[x \leq 4]$ qui impose que le temps qui s'écoule dans ℓ_0 à partir de $x = 0$ soit inférieur à 4 unités de temps. Enfin, le tir des transitions est lui aussi contraint : c_1 ne peut être tirée que si $x \leq 4$. La spécification de la Figure 1 impose donc qu'on fasse c_1 avant 4 unités de temps. La sémantique d'un automate temporisé est un *système de transitions temporisé* (*Timed Transition System*, TTS), qui est constitué de deux types d'évolution : *passage du temps* et *action discrète*. Une exécution est une séquence de telles évolutions avec une alternance entre le passage du temps et les actions discrètes. Des exécutions possibles pour l'automate temporisé de la Figure 1 sont

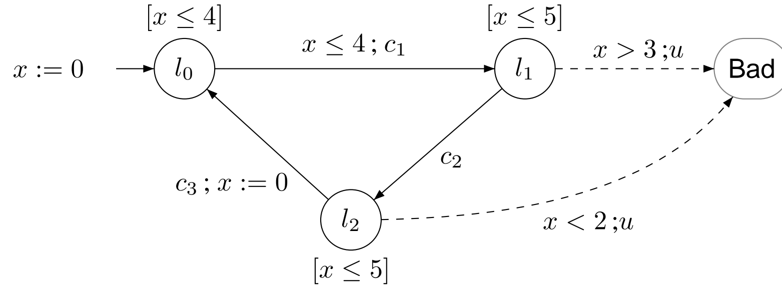

Figure 1. Un exemple de jeu temporisé – le système ouvert S.

par exemple[1] ρ_1 et ρ_2 :

$$\begin{aligned}\rho_1 : \quad & (\ell_0, 0) \xrightarrow{1.55} (\ell_0, 1.55) \xrightarrow{c_1} (\ell_1, 1.55) \\ & \xrightarrow{1.67} (\ell_1, 3.22) \xrightarrow{u} (\mathsf{Bad}, 3.22) \\ \rho_2 : \quad & (\ell_0, 0) \xrightarrow{1.1} (\ell_0, 1.1) \xrightarrow{c_1} (\ell_1, 1.1) \xrightarrow{2.1} (\ell_1, 3.2) \\ & \xrightarrow{c_2} (\ell_2, 3.2) \xrightarrow{0.1} (\ell_2, 3.3) \xrightarrow{c_3} (\ell_0, 0) \cdots\end{aligned}$$

Dans un système de transitions temporisé il y a tout à la fois une infinité de transitions temporisées (d'étiquette $\delta \in \mathbb{R}_{\geq 0}$) et une infinité d'états. Cependant, il existe une abstraction finie de ces systèmes (*automate des régions* de Alur & Dill [8]) qui permet de prouver des propriétés de correction. Les états *symboliques* d'une telle abstraction sont des *zones* définies par une localité et des contraintes sur les horloges. Dans l'exemple de la Figure 1, on obtient des états symboliques du type $(\ell_0, 1 \leq x < 3)$. Bien entendu dans le cas où il y plus d'une horloge les zones sont plus compliquées [8, 13].

Pour l'exemple de la Figure 1, si le contrôleur veut éviter l'état Bad, il doit imposer des restrictions sur les dates de tir des transitions contrôlables : par exemple, il ne doit pas autoriser l'action c_1 à partir de ℓ_0 si $x > 3$; il ne doit pas non plus attendre trop longtemps dans ℓ_1 s'il arrive dans cette localité avec $x \leq 3$, car dès que $x > 3$ l'environnement peut tirer une transition incontrôlable amenant le système dans Bad.

2 Sémantique des jeux temporisés

Lors du déroulement du jeu, il est nécessaire de définir la marche à suivre, et aussi les coups possibles pour chaque joueur. Dans le cas des jeux *discrets*, les coups possibles sont des *actions discrètes*. Chacun des joueurs possède un ensemble d'actions qu'il peut jouer, et qui peuvent varier suivant l'état dans lequel se trouve le système. Dans l'exemple de la Figure 1, c_1 n'est tirable que dans l'état $(\ell_0, x \leq 4)$, u dans $(\ell_1, x > 3), (\ell_2, x < 2)$. Dans le cas des jeux temporisés [38, 12], les joueurs disposent d'une action supplémentaire qui est l'« attente » : un joueur peut décider d'attendre avant de faire une action discrète si celle-ci ne doit pas être jouée trop tôt. Dans les deux cas (attente ou action discrète), la légalité et le résultat d'une action sont déterminés par les *règles du jeu*.

[1]On note (ℓ, v) un état du système au lieu de $(\ell, x = v)$.

2.1 Règles du jeu

Dans la théorie des jeux discrets [44] (automates finis, à pile, *etc.*), les règles du jeu définissent l'espace d'états du système : par exemple un jeu *à tour* à deux joueurs impose que les coups des joueurs alternent ; le prochain état du système est déterminé par un joueur à la fois, celui dont c'est le tour de jouer. Un jeu discret à deux joueurs est dit *concurrent* s'il impose que les deux joueurs choisissent simultanément et indépendamment leurs actions : le prochain état du système est alors la résultante de ces choix simultanés. Dans le cas des jeux temporisés, la notion de jeu à tour n'a pas beaucoup de sens[2] : l'intérêt du temps (quantitatif) dans le modèle est de décrire des joueurs dont les comportements dépendent du temps, et leurs actions sont donc fonctions des délais écoulés. On distingue deux *sémantiques* classiques pour les jeux temporisés : les jeux à *observation continue* [38, 12] et les jeux avec *surprise* [25].

Les jeux à *observation continue* [38, 12] sont tels que chaque joueur (et donc le contrôleur) observe continûment l'état du système pour déterminer son action, soit attendre, soit faire une action discrète ; dans l'exemple de la Figure 1, dans la localité ℓ_1, le contrôleur peut attendre tant qu'il veut (jusqu'à $x = 5$) ou tirer la transition c_2 : son choix se fait en fonction de l'état courant du système dont il a une connaissance exacte à tout moment. Il faut comprendre que si par exemple, le système est encore dans ℓ_1 à la date $x = 4$, il se peut que l'environnement tire u conduisant le système à Bad. Si, à cette même date, le contrôleur choisit aussi de tirer c_2, le système va de façon non-déterministe dans ℓ_2 ou dans Bad. Par conséquent, si le contrôleur veut à tout prix empêcher le système d'atteindre Bad, le contrôleur doit tirer c_2 avant la date $x = 3$ (incluse). En ce sens, le contrôleur peut anticiper les actions de l'environnement : l'environnement ne peut pas le *surprendre* car s'il peut faire une action incontrôlable (mauvaise) dans δ unités de temps, le contrôleur peut observer tous les états intermédiaires et décider de faire une action discrète avant δ.

La sémantique des jeux avec *surprise* [25] est *symétrique* : chaque joueur doit proposer indépendamment un coup qui est un couple (durée, action), où l'action peut

[2]Sauf dans le cas du contrôle d'un environnement continu par un contrôleur digital comme dans [34].

être une action de contrôle ou l'action « ne rien faire ». Le couple (δ, a) correspond au coup « je propose de faire l'action a dans δ unités de temps » ; une fois les propositions faites, par exemple (δ_1, a_1) pour le contrôleur et (δ_2, a_2) pour l'environnement, c'est celle de durée la plus courte qui est sélectionnée. Si $\delta_1 < \delta_2$ le prochain état du jeu est déterminé par le résultat de a_1, et si $\delta_1 = \delta_2$ il y a deux prochains états possibles : celui obtenu après a_1 et celui obtenu après a_2. Contrairement à la sémantique de l'*observation continue*, le contrôleur ne peut plus intervenir une fois qu'il a choisi de laisser passer δ_1 unités de temps et il peut se faire surprendre par l'environnement qui fait une action avant que δ_1 unités de temps se soient écoulées.

2.2 Stratégies

Dans le cadre de la théorie des jeux, le problème de contrôle (CP) devient : « Existe-t-il une *stratégie* pour que le contrôleur gagne le jeu ? » (la condition de gain est discutée dans la partie 2.3). On confond contrôleur et stratégie car ils représentent la même chose. La notion de stratégie est celle communément utilisée dans les jeux (comme les échecs par exemple). Une stratégie indique le coup à jouer, en fonction de l'historique des coups joués par les joueurs et des états rencontrés depuis le début du jeu. Dans le cas des jeux temporisés sous la sémantique à *observation continue*, une stratégie est une fonction partielle[3] de l'historique des coups (incluant les temps écoulés) dans l'ensemble $\{\lambda\} \cup \Sigma_c$ où λ est l'action spéciale « ne rien faire ». Dans le cas de la Figure 1, on peut définir partiellement la stratégie f par exemple par : $f(\rho.\ell_0, x < 2) = \lambda$, $f(\rho.\ell_0, x = 2) = c_1$, où la notation $\rho.\ell_i$ signifie « pour tous les historiques se terminant en ℓ_i ». Cette stratégie indique d'attendre à partir de tout historique se terminant dans l'état $(\ell_0, x < 2)$, et de faire l'action c_1 quand l'historique se termine par $(\ell_0, x = 2)$. Dans le système contrôlé par f, il est impossible d'obtenir des trajectoires se terminant en $(\ell_0, x > 2)$ car pour cela, il faudrait que la stratégie autorise l'action λ d'attente pour un historique se terminant en $(\ell_0, x = 2)$. Une stratégie restreint l'espace d'états atteigables du système de départ. Une stratégie f qui ne dépend pas de toute l'histoire mais seulement du dernier état du système est appelée *stratégie positionnelle* ou encore *sans mémoire*. Bien entendu, une stratégie doit prescrire un coup valide. Ainsi la fonction $f'(\rho.\ell_0, x > 4) = c_1$ n'est pas une stratégie car elle prescrit un coup qui n'est pas dans le jeu donné par l'exemple de la Figure 1. Dans le cas des jeux *avec surprise*, une stratégie est une fonction de l'historique dans l'ensemble des couples de $\mathbb{R}_{\geq 0} \times (\{\lambda\} \cup \Sigma_c)$. Si, dans un jeu *avec surprise*, on impose que les coups soient de la forme $(0, c)$ avec $c \in \Sigma_c$ ou (δ, λ) avec $\delta \in \mathbb{R}_{\geq 0}$, on obtient des stratégies « sans surprise » où l'état du système est observable après tout délai écoulé et avant toute action discrète.

[3] Il se peut en effet qu'un historique se termine par un état où aucun coup du contrôleur n'est possible ; dans ce cas c'est l'environnement seul qui peut jouer et on ne peut pas définir de stratégie pour le contrôleur.

2.3 Objectifs de contrôle

Dans la section 1, le problème de contrôle (CP) a comme paramètre la propriété souhaitée φ. Les propriétés les plus simples sont les propriétés (définies sur les états) de *sûreté* (*safety*) et d'*atteignabilité* (*reachability*). Dans le premier cas, φ désigne un ensemble d'états sûrs du système, et le problème de contrôle consiste alors à trouver une stratégie de manière à maintenir le jeu dans les états de φ : on parle alors de *safety control problem* (SCP). Dans le second cas, φ correspond à un ensemble d'états dans lequel on veut amener le système : le problème de contrôle consiste à trouver une stratégie de manière à forcer le jeu à aller dans un état de φ et on parle de *reachability control problem* (RCP). De façon plus générale, on peut définir des objectifs de contrôle qui sont des formules de logiques temporelles comme LTL [40], MTL [14] ou TCTL [31]. On peut ainsi spécifier des objectifs comme « amener infiniment souvent le jeu dans φ_1 sans jamais passer par φ_2 ». Plus généralement on peut définir des objectifs ω-réguliers [44].

2.4 Propriétés des stratégies

On définit maintenant de manière plus formelle (Cf. [38, 12]) les problèmes de contrôle donnés sous la forme de jeux temporisés ayant une sémantique à *observation continue* et pour des objectifs de sûreté. Soit G un TGA (par exemple l'automate temporisé S de la Figure 1), et tel que l'ensemble des états atteignables dans G (sans aucun contrôle) est $\mathit{Reach}(G)$. Soit φ l'ensemble des états sûrs. On considère le SCP pour G et φ. Une stratégie, pour être gagnante, va devoir restreindre l'ensemble des états atteignables du système pour que seuls des états sûrs soit atteignables. Remarquons tout d'abord qu'une stratégie, même si elle est déterministe, n'engendre pas une unique exécution. En effet, une stratégie ne donne que des restrictions sur les actions contrôlables (et ne restreint donc pas les actions incontrôlables). Par exemple la stratégie $f(\rho.\ell_2, x < 1) = \lambda$, et $f(\rho.\ell_2, x = 1) = c_3$ engendre les exécutions $(\ell_2, x = \nu < 1) \xrightarrow{\delta} (\ell_2, x = \nu + \delta < 1)$ et aussi $(\ell_2, x = \nu < 1) \xrightarrow{u} (\mathsf{Bad}, x = \nu)$. De même à partir de l'état $(\ell_2, x = 1)$, il n'y a pas de priorité à l'action contrôlable c_2, et les deux exécutions $(\ell_2, x = 1) \xrightarrow{c_3} (\ell_0, x = 1)$ et $(\ell_2, x = 1) \xrightarrow{u} (\mathsf{Bad}, x = 1)$ sont possibles. Le jeu G contrôlé avec la stratégie f, noté $f(G)$, produit comme résultat un ensemble d'exécutions qui forme un sous-ensemble de l'ensemble des exécutions du système ouvert G. En terme d'états atteignables, on a $\mathit{Reach}(f(G)) \subseteq \mathit{Reach}(G)$. Le SCP devient dans ce cas : « Existe-t-il une stratégie f, telle que $\mathit{Reach}(f(G)) \subseteq \varphi$? ». Une stratégie qui satisfait cette inclusion est dite *gagnante*. La stratégie triviale f_λ qui consiste pour le contrôleur à ne rien faire (toujours dire λ) peut être une stratégie gagnante : il suffit de vérifier par model-checking que $\mathit{Reach}(f_\lambda(G)) \subseteq \varphi$. Mais cette solution n'est pas satisfaisante car le jeu risque de se bloquer du fait de l'immobilisme du contrôleur. On sou-

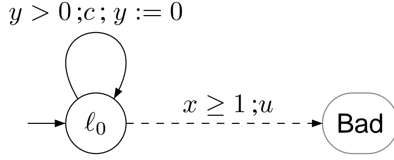

Figure 2. Contrôle Zénon

haite évidemment éviter cela et que le contrôleur soit non-bloquant. On cherche donc des stratégies qui ont cette propriété de non-blocage [38, 24, 30]. De la même manière, il se peut qu'une stratégie soit gagnante parce qu'elle induit des comportements dits *Zénon*, c'est-à-dire des comportements dans lesquels un nombre infini d'actions discrètes sont effectués en un temps fini. Sur l'exemple de la Figure 2, si l'on veut éviter Bad et donc rester dans $\varphi = \{(\ell_0, x, y \geq 0)\}$, on peut définir la stratégie suivante : soit ρ_n l'historique $(\ell_0, x = 0) \xrightarrow{(\delta_0,c)} \cdots \xrightarrow{(\delta_n,c)} (\ell_0, x = \sum_{i=0}^{n} \delta_i)$, où la transition $\xrightarrow{(\delta_i,c)}$ consiste à attendre δ_i unités de temps et ensuite à faire c. On définit la stratégie $f(\rho_n) = (\frac{1}{2} \cdot (1 - \sum_{i=0}^{n} \delta_i), c)$. Cette stratégie est bien non bloquante si on part de $0 < \delta_0 < 1$. Elle produit une seule exécution limite (infinie) qui ne passe jamais par Bad. Néanmoins elle empêche le temps de dépasser la valeur 1, produisant ainsi une exécution *Zénon*. En pratique, cela impose que le contrôleur réagisse de plus en plus vite et fasse des actions c séparées par une durée de plus en plus courte, tendant vers zéro. Ceci n'est pas réaliste et des méthodes ont été définies pour construire des stratégies *non-Zénon* [21, 25].

2.5 États gagnants

Le problème de contrôle se réduit au problème du calcul des états *gagnants* du jeu. Ces états sont définis sémantiquement comme étant ceux à partir desquels il existe une stratégie gagnante. On note $\mathcal{W}$ les états gagnants du jeu. Si on sait calculer l'ensemble de ces états, on peut déterminer facilement s'il existe une stratégie permettant de gagner le jeu : il suffit de déterminer si l'état initial fait partie de $\mathcal{W}$. Une bonne propriété des jeux de sûreté à *observation continue* est qu'ils sont *déterminés*, ce qui signifie que tout état s du système est soit gagnant pour le contrôleur, soit perdant (gagnant pour l'environnement). De plus, dans le cas où l'état initial est gagnant, il existe une stratégie globale f, sans mémoire, gagnante (*cf.* théorème 2 dans la partie 3.3). Cette propriété de jeu déterminé, permet par exemple de réduire le calcul des états gagnants pour un jeu d'atteignabilité « le contrôleur doit forcer φ » à un jeu *inversé* où on calcule les états gagnants du jeu de sûreté « l'environnement doit maintenir le système dans $\neg\varphi$ ».

Dans le cas général des jeux *avec surprise*, on n'a plus ces propriétés car les jeux *avec surprise* ne sont pas déterminés. D'autre part, les jeux avec surprise sont plus compliqués que les jeux à observation continue : par exemple, il existe des jeux temporisés *avec surprise* pour lesquels il faut des stratégies à mémoire non bornée pour gagner [25], alors que les stratégies sans mémoire suffisent pour gagner les jeux *à observation continue* (Théorème 2).

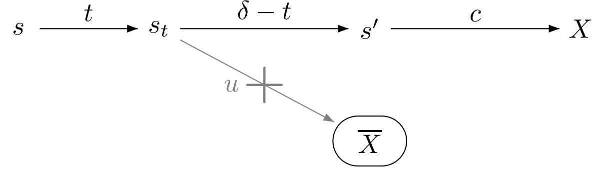

Figure 3. Prédécesseurs contrôlables

3 Algorithmes de synthèse de contrôleur

Les algorithmes pour la synthèse de contrôleur pour les automates temporisés de jeu sont donnés dans [38, 12, 23, 25]. On donne ici l'exemple d'un algorithme permettant de calculer les états gagnants d'un jeu de sûreté dans le cadre défini dans [38]. Le résultat de cet algorithme sur l'exemple de la Figure 1 est donné à la fin de la section.

3.1 Prédécesseurs contrôlables

Soit $\mathit{CPre}(X) = \{s \,|\, \exists c \in \Sigma_c \,|\, s \xrightarrow{c} s' \wedge s' \in X\}$ et $\mathit{UPre}(X) = \{s \,|\, \exists u \in \Sigma_u \,|\, s \xrightarrow{u} s' \wedge s' \in X\}$. Ces deux ensembles correspondent aux états à partir desquels il existe une transition contrôlable (resp. incontrôlable) dont le but est dans X.

La Figure 3 décrit les conditions pour qu'un état s soit un *prédécesseur contrôlable* d'un ensemble d'états X : il faut que le contrôleur puisse *forcer* le système à aller dans un état de X, en choisissant de laisser passer du temps (δ) puis de faire une action contrôlable c, et ce sans que l'environnent puisse l'amener à l'extérieur de X (noté $\overline{X}$) à partir d'un état intermédiaire s_t rencontré en allant de s à s'.

Un état s est donc un prédecesseur contrôlable de X si, et seulement si :

1. $\exists \delta \geq 0$ tel que $s \xrightarrow{\delta} s' \wedge s' \in \mathit{CPre}(X)$
2. pour tout $0 \leq t \leq \delta$ tel que $s \xrightarrow{t} s_t$, on a $s_t \notin \mathit{UPre}(\overline{X})$

L'ensemble des prédécesseurs contrôlables de X est noté $\pi(X)$.

Dans l'exemple de la Figure 1, l'état $(\ell_1, x = 1)$ est un prédécesseur contrôlable de $(\ell_2, x = 2)$: on peut laisser passer du temps de $(\ell_1, x = 1)$ jusqu'à $(\ell_1, x = 2)$ puis tirer c_2 de ℓ_1 à ℓ_2. Lors du passage du temps aucune transition incontrôlable n'est tirable.

3.2 Opérateurs symboliques

Comme nous l'avons indiqué précédemment, la sémantique d'un automate temporisé est un système de transitions temporisé contenant une infinité de transitions et d'états. Pour analyser de tels automates, il est nécessaire de regrouper les états en *états symboliques* qui sont des (unions finies de) couples (ℓ, Z) où ℓ est une localité et

Z une contrainte convexe sur les horloges. Les opérateurs *CPre*, *UPre* et π ont deux propriétés importantes. Si X est une union de zones, alors :

$\mathbf{P_1}$: $CPre(X), UPre(X)$ et $\pi(X)$ sont des unions de zones,

$\mathbf{P_2}$: $CPre(X)$, $UPre(X)$ et $\pi(X)$ sont effectivement calculables[4].

Pour l'automate temporisé de la Figure 1, on a $CPre(\ell_1, x \leq 3) = (\ell_0, x \leq 3)$ et si $Z = (\ell_0, x \leq 4) \cup (\ell_1, x \geq 0) \cup (\ell_2, x \geq 0)$ alors $\pi(Z) = Z'$ avec $Z' = (\ell_0, x \leq 3) \cup (\ell_1, 0 \leq x \leq 3) \cup (\ell_2, x \geq 2)$.

3.3 Calcul symbolique des états gagnants

On peut montrer [12] que dans le cas d'un TGA, et d'un objectif de sûreté φ convexe, l'ensemble des états gagnants $\mathcal{W}$, défini dans la partie 2.5, est le plus grand[5] point fixe de la fonction $h(X) = \varphi \cap \pi(X)$. Il faut bien noter que $\mathcal{W}$ est l'ensemble des états gagnants, donc maximal ; donc un état s est gagnant ssi $s \in \mathcal{W}$. Si φ est défini comme une union de zones, alors le calcul itératif défini par $X_0 = \varphi$, $X_{i+1} = h(X_i) = \varphi \cap \pi(X_i)$ est tel que tous les X_i sont des unions de zones[6], et s'arrête en un nombre fini d'itérations. Lorsque $X_{i_0+1} = X_{i_0}$, le plus grand point fixe $W^* = X_{i_0}$ de h est atteint et $W^* = \mathcal{W}$. Comme on peut décider si l'état initial d'un automate appartient à un état symbolique, on obtient donc le thèorème suivant :

Théorème 1 ([12, 35]) *Les problèmes de contrôle SCP et RCP sont décidables pour les automates temporisés et EXPTIME-complets.*

Dans le cas de notre exemple de la Figure 1, on obtient itérativement les ensembles[7] du Tableau 1. Pour les propriétés plus compliquées que des propriétés de sûreté, le calcul des états gagnants est plus complexe [38, 12, 23, 25].

3.4 Synthèse de stratégies gagnantes

Pour ce qui est du problème de synthèse de stratégie, dans le cas d'un objectif de sûreté, on peut définir la stratégie *maximale* ou *la plus permissive*. En toute rigueur, cette stratégie la plus permissive n'est pas une stratégie au sens de la partie 2.2 car il faudrait qu'elle associe une action de $\Sigma_c \cup \{\lambda\}$ à une exécution. La stratégie la plus permissive f^* associe à chaque exécution ρ un sous-ensemble non vide de $\Sigma_c \cup \{\lambda\}$. Toute (sous-)stratégie f (au sens de la partie 2.2) non bloquante telle que $f(\rho) \in f^*(\rho)$ est une stratégie gagnante, et f^* est maximale pour cette propriété (c'est pourquoi on la qualifie de stratégie la plus permissive). Concernant la synthèse de stratégies on a le théorème suivant :

[4]Pour calculer $\pi(X)$, il est nécessaire d'introduire un nouvel opérateur qui n'est pas décrit ici mais qui est facilement calculable.

[5]Dans le cas d'un jeu de *reachability* c'est le plus petit point fixe de la fonction $h(X) = \varphi \cup \pi(X)$.

[6]L'intersection de zones est une zone.

[7]Le domaine gagnant est donné pour chaque localité ℓ_i dans la colonne correspondante.

Théorème 2 ([12]) *Si G est un automate temporisé tel que l'état initial de G est gagnant pour l'objectif de sûreté φ, alors il existe une stratégie (maximale) f^* gagnante et sans mémoire.*

Pour calculer cette stratégie on commence par renforcer les gardes des transitions de telle manière qu'en partant d'un état gagnant s, si on franchit une transition contrôlable dans l'automate de jeu avec les gardes renforcées, alors on atteint un état gagnant. Les renforcements des gardes sont donc les plus grandes *préconditions* permettant d'atteindre des états gagnants. Dans notre cas on doit renforcer la garde de c_1 en ajoutant la contrainte $x \leq 3$, celle de c_2 en ajoutant la contrainte $x \geq 2$ (qui indique qu'on ne doit pas tirer trop tôt c_2). La stratégie la plus permissive est ensuite donnée par : attendre pour tous les états gagnants s tels qu'il existe $\delta > 0$ et $s \xrightarrow{\delta} s'$ et s' est gagnant ; faire une transition contrôlable c_i si la garde renforcée le permet. Dans le cas de l'exemple de la Figure 1 la stratégie la plus permissive est donnée dans le Tableau 1. Il faut bien noter que pour obtenir une stratégie *non-Zénon*, il ne faut pas choisir indéfiniment λ dans ℓ_0 (ou ℓ_1, ℓ_2), mais bien tirer c_1 à un certain point. De cette manière, on construit un automate temporisé C (donné Figure 4) qui représente la stratégie la plus permissive f^*, et tel que $G \parallel C$ satisfait la propriété de sûreté φ. Dans le cas de jeux d'atteignabilité, le calcul d'une stratégie gagnante peut nécessiter plus de travail [15].

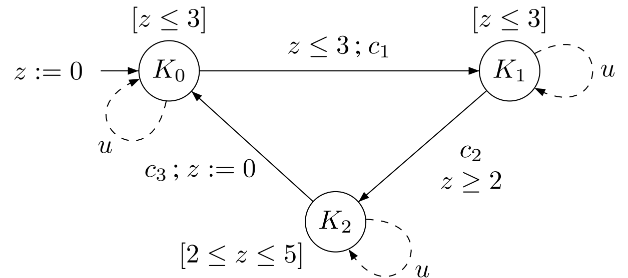

Figure 4. Contrôleur le plus permissif pour l'exemple de la Figure 1.

4 Pour aller plus loin

Observation partielle. Nous avons ici supposé que le système à contrôler était totalement observable : à tout moment le contrôleur peut connaître l'état du système, les valeurs des différentes horloges et les actions. Dans un système plus réaliste, le contrôleur accède aux variables du système par des capteurs et le système peut posséder ses propres variables, non communiquées au contrôleur. Largement étudié pour les systèmes à événements discrets, le problème du contrôle sous observation partielle se pose également dans un cadre temporisé. Différents niveaux d'observation partielle ont été introduits, allant

Calcul des états gagnants W^*

X_i	ℓ_0	ℓ_1	ℓ_2
0	$0 \le x \le 4$	$0 \le x \le 5$	$0 \le x \le 5$
1	$0 \le x \le 4$	$0 \le x \le 3$	$2 \le x \le 5$
2	$0 \le x \le 3$	–	–

Stratégie la plus permissive f^*

	ℓ_0	ℓ_1	ℓ_2
λ	$x < 3$	$x < 3$	$x < 5$
c	$x \le 3$	$2 \le x$	$x \le 5$

TAB. 1. Calculs symboliques pour l'exemple de la Figure 1.

de la non-observabilité d'actions à la non-observabilité d'horloges ou d'états [17].

Algorithmes, modèles et propriétés. Concernant les algorithmes de synthèse de contrôleurs, des travaux récents ont proposé des implémentations efficaces de l'algorithme à point fixe exposé dans cet article [3, 4, 20]. Les algorithmes de synthèse de contrôleurs ont été étendus au cas des systèmes *hybrides* [34, 35, 45], qui sont une généralisation des automates temporisés où les horloges peuvent évoluer à différentes vitesses. Finalement, dans le cas de jeux d'atteignabilité pour des automates temporisés se pose le problème du temps optimal pour atteindre un état gagnant. Ce problème a été résolu dans [11]. Des critères plus généraux d'« optimalité » sont considérés dans [7, 15, 19]. Pour considérer des propriétés plus compliquées, il est possible d'exprimer les objectifs de contrôle ω-réguliers [38, 23] ou encore à l'aide de logiques modales temporisés [16, 14] .

Deuxième partie

Implémentabilité des contrôleurs temporisés

Nous venons de voir comment il est possible de synthétiser des stratégies gagnantes pour contrôler des systèmes réactifs. Ces stratégies peuvent par exemple être représentées sous la forme d'un automate temporisé, qui s'exécutera en parallèle avec le système à contrôler.

On peut cependant se poser la question de la pertinence de l'utilisation 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 par l'implémentation.

Nous exposons ici 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.

4.1 Motivations

Rappelons qu'un automate temporisé est un automate fini étendu avec des variables mesurant le temps. Un tel automate possède deux types de transitions : les transitions de temps, qui consistent à laisser un certain délai s'écouler et donc à augmenter toutes les horloges de l'automate de ce délai, et les transitions d'action, à condition que les valeurs des horloges le permettent, changent l'état de l'automate et remettent éventuellement à zéro certaines horloges.

Prenons l'exemple du protocole d'exclusion mutuelle de Fischer [36] : 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 5 représente un système et son contrôleur.

On peut facilement montrer, grâce à un outil de vérification d'automates temporisés comme UPPAAL, KRONOS ou HyTech, que l'exclusion mutuelle est bien réalisée par ces contrôleurs.

4.2 Des automates temporisés aux plates-formes d'exécution

La propriété d'exclusion mutuelle de notre exemple est cependant étroitement liée au caractère infiniment précis du modèle mathématique 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 $x_i \le 2$ de l'état Requête, que lorsqu'un processus P_i peut atteindre sa section critique depuis l'état Attente, aucun autre processus P_j ne peut être dans l'état Requête. Ainsi, relâcher la contrainte $x_i > 2$ en la contrainte $x_i \ge 2$ 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 :

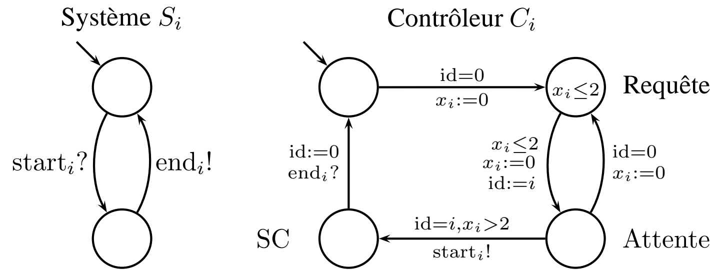

Figure 5. Le protocole d'exclusion mutuelle de Fischer

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 plate-formes 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.

4.3 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 ».

4.4 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 de

a) récupérer et transmettre les entrées,

b) mettre à jour l'horloge du processeur,

c) 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 [6]. 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 [5], 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 5.

2. *Adaptation de la sémantique de l'automate temporisé :* cette approche, introduite dans [27], 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 6.

5 Modélisation de la plate-forme d'exécution

L'approche de [5] 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 $\text{Prog}(\mathcal{A})$ 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 $\text{Prog}(\mathcal{A})$ modélise l'exécution du programme sur la plate-forme. Le but est alors de comparer cette exécution avec la sémantique de $\mathcal{A}$ et d'en déduire la préservation éventuelle de propriétés.

5.1 Le programme

Dans le modèle, l'automate temporisé $\mathcal{A}$ est directement en contact avec l'environnement physique Env (ce que nous notons $\mathcal{A} \leftrightarrow \text{Env}$). 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 $\mathcal{A} \leftrightarrow \mathcal{P} \leftrightarrow \text{Env}$. La transmission du temps physique comme des entrées/sorties de $\mathcal{A}$ sont filtrées et/ou déformées par le passage *via* la plate-forme. 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 plate-forme, 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 $\text{Prog}(\mathcal{A})$ est obtenu par transformation syntaxique de $\mathcal{A}$; un exemple est donné à la figure 6.

5.2 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 7 : les trois points identifiés au paragraphe 4.2 comme mettant en défaut les hypothèses idéales de la sémantique des automates temporisés sont isolés et chacun est modélisé par un automate temporisé.

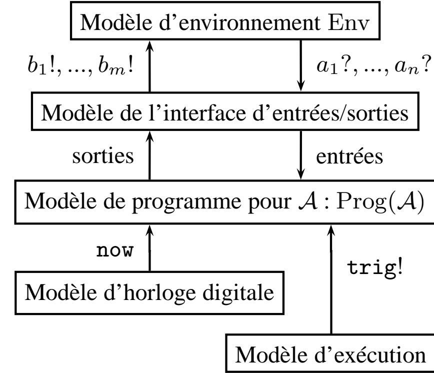

Figure 7. Modèle global d'exécution

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 8.
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.

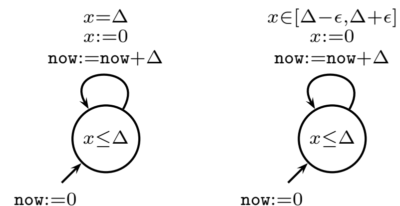

Figure 8. Deux modèles d'horloge digitale

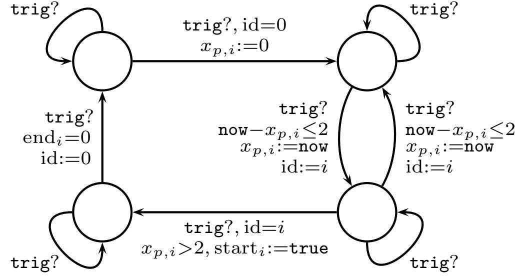

Figure 6. L'automate $\mathrm{Prog}(C_i)$, où C_i est un contrôleur du protocole de Fischer illustré à la figure 5.

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 : [5] 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.

6 Implémentabilité et robustesse

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.

6.1 La plate-forme

Nous considérons le modèle de plate-forme proposé dans [27] : 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.

6.2 De l'implémentabilité à la robustesse

[27] étudie alors les propriétés théoriques de ce type de plate-formes 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-formes, 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, c'est-à-dire à remplacer une garde $x \in [a, b]$ par la nouvelle garde $x \in [a - \Delta, b + \Delta]$.

Les auteurs de [27] 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 4.2 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-formes.

6.3 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 [26] 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 [8].

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 [28].

Dans [26], 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 ») [18].

7 Pour aller plus loin

Algorithmes efficaces pour la vérification robuste. Il est bien connu que les algorithmes basés sur les régions (comme celui que nous avons présenté ici pour l'approche sémantique) *ne passent pas à l'échelle*, c'est-à-dire qu'ils ne termineront, en pratique, que de petits exemples. Dans le cadre de la sémantique classique des automates temporisés, des algorithmes symboliques et des structures de données efficaces ont été définis, et les outils correspondant ont montré leur puissance (UPPAAL, KRONOS, ...) Plusieurs travaux ont été initiés sur l'extension de ces algorithmes symboliques au model-checking robuste [22, 43, 29].

Synthèse de contrôleurs robustes. Bien entendu, les deux parties de cet articles se rejoignent et ouvrent un champs vaste de travaux futurs : un défi important est de mettre au point des techniques de synthèse de contrôleurs implémentables, sans passer par la phase de vérification de l'implémentabilité.

Pour ce problème, l'approche de l'implémentabilité par modélisation semble très prometteuse, puisqu'elle permet de modéliser de façon très précise les interactions entre le contrôleur et le système qu'il supervise. Elle permet par ailleurs d'utiliser directement les algorithmes déjà existants.

Références

[1] K. Altisen, P. Bouyer, T. Cachat, F. Cassez, and G. Gardey. Introduction au contrôle des systèmes temps-réel. In H. Alla and É. Rutten, editors, *Actes du 5ème Colloque sur la Modélisation des Systèmes Réactifs (MSR'05)*, pages 367–380, Autrans, France, Oct. 2005. Hermès. Invited paper.

[2] K. Altisen, N. Markey, P.-A. Reynier, and S. Tripakis. Implémentabilité des automates temporisés. In H. Alla and É. Rutten, editors, *Actes du 5ème Colloque sur la Modélisation des Systèmes Réactifs (MSR'05)*, pages 395–406, Autrans, France, Oct. 2005. Hermès. Invited paper.

[3] K. Altisen and S. Tripakis. On-the-fly controller synthesis for discrete and dense-time systems. In *World Congress on Formal Methods (FM'99)*, volume 1708 of *Lecture Notes in Computer Science*, pages 233–252. Springer, 1999.

[4] K. Altisen and S. Tripakis. Tools for controller synthesis of timed systems. In *Proc. 2nd Workshop on Real-Time Tools (RT-TOOLS'02)*, 2002. Proc. published as Technical Report 2002-025, Uppsala University, Sweden.

[5] K. Altisen and S. Tripakis. Implementation of timed automata: An issue of semantics or modeling? Technical Report TR-2005-12, Verimag, Centre Équation, 38610 Gières, France, June 2005.

[6] R. Alur. *Techniques for Automatic Verification of Real-Time Systems*. PhD thesis, Stanford University, Stanford, CA, USA, 1991.

[7] R. Alur, M. Bernadsky, and P. Madhusudan. Optimal reachability in weighted timed games. In *Proc. 31st International Colloquium on Automata, Languages and Programming (ICALP'04)*, volume 3142 of *Lecture Notes in Computer Science*, pages 122–133. Springer, 2004.

[8] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[9] T. Amnell, G. Behrmann, J. Bengtsson, P. R. D'Argenio, A. David, A. Fehnker, T. Hune, B. Jeannet, K. G. Larsen, O. Möller, P. Pettersson, C. Weise, and W. Yi. UPPAAL – now, next, and future. In *Proc. Modelling and Verification*

of Parallel Processes (MOVEP2k), volume 2067 of *Lecture Notes in Computer Science*, pages 99–124. Springer, 2001.

[10] A. Arnold, A. Vincent, and I. Walukiewicz. Games for synthesis of controllers with partial observation. *Theoretical Computer Science*, 1(303):7–34, 2003.

[11] E. Asarin and O. Maler. As soon as possible: Time optimal control for timed automata. In *Proc. 2nd International Workshop on Hybrid Systems: Computation and Control (HSCC'99)*, volume 1569 of *Lecture Notes in Computer Science*, pages 19–30. Springer, 1999.

[12] E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *Proc. IFAC Symposium on System Structure and Control*, pages 469–474. Elsevier Science, 1998.

[13] P. Bouyer. Forward analysis of updatable timed automata. *Formal Methods in System Design*, 24(3):281–320, 2004.

[14] P. Bouyer, L. Bozzelli, and F. Chevalier. Controller synthesis for MTL specifications. In C. Baier and H. Hermanns, editors, *Proceedings of the 17th International Conference on Concurrency Theory (CONCUR'06)*, volume 4137 of *Lecture Notes in Computer Science*, pages 450–464, Bonn, Germany, Aug. 2006. Springer.

[15] P. Bouyer, F. Cassez, E. Fleury, and K. G. Larsen. Optimal strategies in priced timed game automata. In *Proc. 24th Conference on Foundations of Software Technology and Theoretical Computer Science (FST&TCS'2004)*, volume 3328 of *Lecture Notes in Computer Science*, pages 148–160. Springer, 2004.

[16] P. Bouyer, F. Cassez, and F. Laroussinie. Modal logics for timed control. In *Proc. 16th International Conference on Concurrency Theory (CONCUR'2005)*, Lecture Notes in Computer Science. Springer, 2005. À paraître.

[17] P. Bouyer, D. D'Souza, P. Madhusudan, and A. Petit. Timed control with partial observability. In W. A. Hunt, Jr and F. Somenzi, editors, *Proceedings of the 15th International Conference on Computer Aided Verification (CAV'03)*, volume 2725 of *Lecture Notes in Computer Science*, pages 180–192, Boulder, Colorado, USA, July 2003. Springer.

[18] P. Bouyer, N. Markey, and P.-A. Reynier. Robust model-checking of timed automata. In *Proc. 7th Latin American Symposium on Theoretical Informatics (LATIN'06)*, volume 3887 of *Lecture Notes in Computer Science*, pages 238–249. Springer, 2006.

[19] T. Brihaye, V. Bruyère, and J.-F. Raskin. On optimal timed strategies. Technical Report 2005.48, Centre Fédéré en Vérification, Belgique, 2005.

[20] F. Cassez, A. David, E. Fleury, K. G. Larsen, and D. Lime. Efficient on-the-fly algorithms for the analysis of timed games. In *Proc. 16th International Conference on Concurrency Theory (CONCUR'2005)*, Lecture Notes in Computer Science. Springer, 2005. À paraître.

[21] F. Cassez, T. A. Henzinger, and J.-F. Raskin. A comparison of control problems for timed and hybrid systems. In *Proc. 5th International Workshop on Hybrid Systems: Computation and Control (HSCC'02)*, volume 2289 of *LNCS*, pages 134–148. Springer, 2002.

[22] C. Daws and P. Kordy. Symbolic robustness analysis of timed automata. In *Proc. 4th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS'06)*, volume 4202 of *Lecture Notes in Computer Science*, pages 143–155, Paris, France, 2006. Springer.

[23] L. de Alfaro, T. A. Henzinger, and R. Majumdar. Symbolic algorithms for infinite-state games. In *Proc. 12th International Conference on Concurrency Theory (CONCUR'01)*, volume 2154 of *Lecture Notes in Computer Science*, pages 536–550. Springer, 2001.

[24] L. de Alfaro, T. A. Henzinger, and M. Stoelinga. Timed interfaces. In *Proc. 2nd International Workshop on Embedded Software (EMSOFT'02)*, volume 2491 of *Lecture Notes in Computer Science*, pages 108–122. Springer, 2002.

[25] L. d. de Alfaro, M. Faëlla, T. A. Henzinger, R. Majumdar, and M. Stoelinga. The element of surprise in timed games. In *Proc. 14th International Conference on Concurrency Theory (CONCUR'2003)*, volume 2761 of *Lecture Notes in Computer Science*, pages 142–156. Springer, 2003.

[26] M. De Wulf, L. Doyen, N. Markey, and J.-F. Raskin. Robustness and implementability of timed automata. In Y. Lakhnech and S. Yovine, editors, *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, 2004.

[27] M. De Wulf, L. Doyen, and J.-F. Raskin. Almost ASAP semantics: From timed models to timed implementations. In R. Alur and G. J. Pappas, editors, *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, 2004.

[28] M. De Wulf, L. Doyen, and J.-F. Raskin. Systematic implementation of real-time models. In *Proceedings of the International Symposium of Formal Methods Europe (FM'05)*, Lecture Notes in Computer Science. Springer, 2005. À paraître.

[29] C. Dima. Dynamical properties of timed automata revisited. Technical report, LACL, Créteil, France, 2006.

[30] D. D'Souza and P. Madhusudan. Timed control synthesis for external specifications. In *Proc. 19th International Symposium on Theoretical Aspects of Computer Science (STACS'02)*, volume 2285 of *Lecture Notes in Computer Science*, pages 571–582. Springer, 2002.

[31] M. Faëlla, S. La Torre, and A. Murano. Dense real-time games. In *Proc. 17th IEEE Symposium on Logic in Computer Science (LICS'02)*, pages 167–176. IEEE Comp. Soc. Press, 2002.

[32] G. Frehse. Phaver: Algorithmic verification of hybrid systems past hytech. In *HSCC*, pages 258–273, 2005.

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

[34] T. A. Henzinger, B. Horowitz, and R. Majumdar. Rectangular hybrid games. In *Proc. 10th International Conference on Concurrency Theory (CONCUR'99)*, volume 1664 of *Lecture Notes in Computer Science*, pages 320–335. Springer, 1999.

[35] T. A. Henzinger and P. W. Kopke. Discrete-time control for rectangular hybrid automata. *Theoretical Computer Science*, 221:369–392, 1999.

[36] K. J. Kristoffersen, F. Laroussinie, K. G. Larsen, P. Pettersson, and W. Yi. A compositional proof of a real-time mutual exclusion protocol. In M. Bidoit and M. Dauchet, editors, *Proceedings of the 7th International Joint Conference*

CAAP/FASE on Theory and Practice of Software Development (TAPSOFT'97), volume 1214 of *Lecture Notes in Computer Science*, pages 565–579, Lille, France, Apr. 1997. Springer.

[37] F. Laroussinie and K. G. Larsen. CMC: A tool for compositional model-checking of real-time systems. In *Proc. IFIP Joint International Conference on Formal Description Techniques & Protocol Specification, Testing, and Verification (FORTE-PSTV'98)*, pages 439–456. Kluwer Academic, 1998.

[38] O. Maler, A. Pnueli, and J. Sifakis. On the synthesis of discrete controllers for timed systems. In *Proc. 12th Annual Symposium on Theoretical Aspects of Computer Science (STACS'95)*, volume 900 of *Lecture Notes in Computer Science*, pages 229–242. Springer, 1995.

[39] R. McNaughton. Infinite games played on finite graphs. *Annals of Pure and Applied Logic*, 65(2):149–184, 1993.

[40] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *Proc. 16th ACM Symposium on Principles of Programming Languages (POPL'89)*, pages 179–190. ACM, 1989.

[41] S. Riedweg and S. Pinchinat. Quantified mu-calculus for control synthesis. In *Proc. 28th International Symposium on Mathematical Foundations of Computer Science (MFCS'03)*, volume 2747 of *Lecture Notes in Computer Science*, pages 642–651. Springer, 2003.

[42] P. Schnoebelen, B. Bérard, M. Bidoit, F. Laroussinie, and A. Petit. *Vérification de logiciels : Techniques et outils de model-checking*. Vuibert, 1999.

[43] M. Swaminathan and M. Fränzle. A symbolic decision procedure for robust safety of timed systems. In *Proc. 14th International Symposium on Temporal Representation and Reasoning (TIME'07)*, page 11, Alicante, Spain, 2007. IEEE Comp. Soc. Press.

[44] W. Thomas. On the synthesis of strategies in infinite games. In *Proc. 12th Annual Symposium on Theoretical Aspects of Computer Science (STACS'95)*, volume 900, pages 1–13. Springer, 1995. Invited talk.

[45] H. Wong-Toi. The synthesis of controllers for linear hybrid automata. In *Proc. 36th IEEE Conference on Decision and Control*, pages 4607–4612. IEEE Comp. Soc. Press, 1997.

[46] S. Yovine. KRONOS: A verification tool for real-time systems. *Journal of Software Tools for Technology Transfer*, 1(1–2):123–133, 1997.