Synthèse de contrôleur simplement valide dans le cadre de la programmation par contraintes

Michel Lemaître, Gérard Verfaillie, Cédric Pralet, Guillaume Infantès

ONERA/DCSD/CD
B.P. 74025 - 2, Av. E. Belin
F-31055 TOULOUSE CEDEX 4
Michel.Lemaitre@onera.fr

Résumé :
Les données d'entrée du problème de la synthèse de contrôleur sont : (1) les transitions T du système à contrôler, (2) les exigences R sur le système à contrôler. La solution du problème, si elle existe, consiste informellement en une politique (ou stratégie) Π valide qui, contrôlant T, satisfait R. On se restreint dans cet article à la synthèse de contrôleurs discrets «continuels», c'est-à-dire avec des exigences sur le comportement du système contrôlé mais sans état but à atteindre. La plupart des systèmes temps-réel correspondent à ce cadre.

Informellement, une politique est dite *totalement valide* si, en tout état atteignable, elle est définie, applicable et conforme aux exigences. En substituant la notion d'état faisable à celle d'état atteignable, on introduit la notion de *validité simple*, qui, moyennant quelques hypothèses naturelles, implique celle de validité totale. En se restreignant à cette notion de validité simple, qui «oublie» le problème de l'atteignabilité, et qui en pratique suffit souvent pour exprimer la validité d'un contrôle continuel, on exprime la politique cherchée sous forme d'un prédicat Π en logique du premier ordre, fonction de T et R. On décrit une méthode de calcul de la politique basée sur le cadre logique proposé, utilisant la programmation par contraintes.

1 Introduction

Un contrôleur est un dispositif qui, en fonction d'observations en provenance du système à contrôler (capteurs, commandes en provenance des opérateurs humains) et de son état interne (une certaine mémorisation du passé), prend continuellement les décisions réactives permettant le contrôle constant du système, de sorte que le comportement du système contrôlé soit conforme à celui désiré.

On s'intéresse ici aux contrôleurs logiciels implantés sur calculateurs digitaux. Le contrôleur réagit continuellement en temps-réel à ses entrées par des sorties appropriées. Dû au mode d'implémentation (sur un calculateur digital), la réaction ne peut pas être continue mais consiste en une suite discrète de calculs et de réactions successifs, aussi rapprochés qu'il est nécessaire ou possible.

On assimile un contrôleur discret à une *politique* (ou stratégie) : une fonction qui à tout *état* du système à contrôler lui fait correspondre une *commande* [1].

On se restreint dans cet article à la synthèse de contrôleurs discrets «continuels», c'est-à-dire avec des exigences sur le comportement du système contrôlé mais sans état but à atteindre. La plupart des systèmes temps-réel correspondent à ce cadre.

La manière courante de construire un contrôleur logiciel est de le décrire directement dans un langage de programmation de plus ou moins haut niveau. On doit ensuite valider le contrôleur, soit par simulation, soit par validation formelle, ou par ces deux moyens. La démarche habituelle en validation formelle de contrôleur consiste à confronter une définition formelle du contrôleur avec les propriétés qu'on désire voir vérifiées. La validation se fait directement sur le code source du contrôleur et non pas par exécution du code. La *synthèse de contrôleur* est une alternative à la fois à la construction et à la validation : il s'agit de synthétiser des contrôleurs valides par construction. La synthèse de contrôleurs consiste en quelque sorte à inverser la démarche de validation : partant d'un modèle du système à contrôler et des propriétés que l'on souhaite lui imposer, on en dérive automatiquement une implémentation du contrôleur.

1. On considère donc ici des contrôleurs sans mémoire. Toutefois, le cas des contrôleurs à mémoire finie s'y ramène, en considérant que la mémoire du contrôleur fait partie du système à contrôler.

De très nombreux travaux existent en synthèse de contrôleur réactifs pour des systèmes à événements discrets. Nous évoquons rapidement ceux qui nous semblent les principaux.

Les pionniers en la matière furent Ramadge et Wonham (Ramadge & Wonham, 1989). Leur approche est basée sur une distinction claire entre le système à contrôler et son contrôleur. Fondée sur la théorie des langages, elle peut être résumée (de manière quelque peu abusive) ainsi : un langage (ensemble de séquences possibles d'événements) L décrit le comportement possible du système à contrôler ; un autre langage K décrit le comportement admissible (désiré, légal) du système une fois contrôlé. Un contrôleur, s'il existe, est un automate produisant un langage C tel que $L \cap C \subseteq K$.

L'article de Pnueli et Rosner (Pnueli & Rosner, 1989), souvent cité, contient les premiers travaux réellement significatifs sur la synthèse complète de contrôleurs réactifs à partir de spécifications en logique temporelle. Techniquement, le comportement du module est spécifié en logique LTL (*Linear Time Logic*) par une formule du type $\forall s \exists c \mathbf{A} \phi(s, c)$ dans laquelle s représente les entrées du contrôleur en provenance du système à contrôler, c représente les sorties du contrôleur, et $\mathbf{A}$ est l'opérateur *always* de la logique temporelle. La méthode de synthèse consiste en la traduction de la formule en un automate.

Maler, Pnueli et Sifakis (Maler *et al.*, 1995), ainsi que Asarin, Maler et Pnueli (Asarin *et al.*, 1995) et Cassez (Cassez & Markey, 2007) présentent des algorithmes pour la synthèse automatique de contrôleurs temps-réel, pour des systèmes décrits par des automates temporisés (*timed-automata*) c'est-à-dire des automates dont les transitions peuvent dépendre explicitement du temps qui passe. Le comportement souhaité du système est décrit par une logique temporelle dans laquelle on exprime que la trajectoire du système (la suite des états qu'il traverse) satisfait une formule de logique temporelle. Cependant les systèmes à contrôler sont des systèmes temps-réel : les décisions de contrôle interagissent avec le passage continu du temps, elles sont discrètes et datées.

Arnold, Vincent et Walukiewick (Arnold *et al.*, 2003) utilisent le μ-calcul pour spécifier le système à contrôler ainsi que les propriétés temporelles à satisfaire. Ils montrent que le problème du contrôle peut se réduire à un problème de satisfiabilité d'une formule du μ-calcul.

L'approche *planning as model-checking*, fruit des travaux de Cimatti, Pistore, Roveri et Traverso Cimatti *et al.* (2003, 2004) (voir aussi (Ghallab *et al.*, 2004, chapitre 17)) concerne a priori les problèmes de planification en horizon fini, mais ils permettent dans une certaine mesure de traiter le problème de la synthèse de contrôleur. Dans cette approche, le système à contrôler est modélisé par un *planning domain* constitué d'un ensemble de variables propositionnelles P servant à coder les états possibles, d'un ensemble d'états $S \subseteq 2^P$, d'un ensemble fini d'actions A, et d'une relation de transition $R \subseteq S \times A \times S$. Étant donnés un ensemble d'états initiaux I et un ensemble d'états finaux G, le problème de base est de construire une table état-action (donc une politique ou stratégie) permettant, à partir d'un état de I, d'atteindre un état de G. Les travaux cités proposent une série d'algorithmes pour calculer une politique, basés sur la représentation de la relation R sous forme de BDD. Il existe des extensions à ce cadre, qui le rendent plus proche de la problématique de la synthèse de contrôleur réactif sans but à atteindre : la planification sur des buts étendus (*extended goals* (Ghallab *et al.*, 2004, section 17.3)). Ces extensions permettent d'exprimer des buts comme des contraintes sur le chemin d'exécution et non pas seulement sur l'état final, par des formules de logique temporelle CTL.

Quelques outils de synthèse de contrôleurs existent. Citons UPPAAL-TiGA (Behrmann *et al.*, 2004) pour la synthèse d'automates temporisés, et Anzu (Jobstmann *et al.*, 2007) pour la synthèse à partir de spécifications en logique temporelle linéaire.

Toutes les approches citées se heurtent au problème de la grande complexité algorithmique des algorithmes de synthèse proposés. Notre contribution vise une méthode de synthèse plus économe en ressources de calcul, basée sur des outils existants de programmation par contraintes, au prix évidemment d'objectifs plus réduits, mais qui dans la pratique s'avèrent suffisants dans beaucoup de cas. Voici une esquisse de cette contribution, développée ensuite dans l'article.

Les données d'entrée du problème de la synthèse de contrôleur sont : (1) les transitions T du système à contrôler, (2) les exigences R sur le système à contrôler. La solution du problème, si elle existe, est une politique Π valide qui, contrôlant T, satisfait R. Informellement, une politique est dite *totalement valide* si, en tout état atteignable, elle est définie, applicable et conforme aux exigences. En substituant la notion d'état faisable à celle d'état atteignable, on introduit la notion de *validité simple*, qui, moyennant quelques hypothèses naturelles, implique celle de validité totale. En se restreignant à cette notion de validité simple, qui «oublie» le problème de l'atteignabilité, et qui en pratique suffit souvent pour le contrôle continuel, on exprime la politique cherchée sous forme d'un prédicat Π en logique du premier ordre, fonction de T et R. On décrit une méthode de calcul de la politique basée sur le cadre logique proposé, utilisant la programmation par contraintes.

La section 2 définit le cadre formel que nous proposons pour le problème de la synthèse de contrôleurs, ainsi que les notions de politiques totalement et simplement valides. Les sections 3 et 4 présentent deux exemples de spécifications de contrôleurs dans le cadre proposé. La section 5 décrit une méthode de synthèse de politique simplement valide, basée sur l'utilisation de la programmation par contraintes. Enfin, nous concluons et indiquons les perspectives de travaux futurs.

2 Le problème de la synthèse de contrôleur

2.1 Systèmes dynamiques à évènements discrets, synchrones

Nous décrivons tout d'abord le cadre dans lequel on place le problème de la synthèse.

Un *système dynamique à événements discrets* est caractérisé par le fait que sa dynamique est régie par des événements discrets (instantanés). À chaque instant, le système possède un état, et des transitions entre ces états sont déclenchées par des événements discrets. Cela n'empêche nullement qu'entre deux événements, le système puisse évoluer de manière continue (mais hors modèle). Les événements peuvent amener des discontinuités dans la dynamique continue du système.

On dira qu'un système est *synchrone* si c'est un système dynamique à événements discrets, et si tous les événements discrets sont datés en bijection avec $\mathbb{N}$. Des événements distincts peuvent se produire à la même date : ce sont des événements simultanés. En d'autres termes, les dates induisent une structure de préordre sur tous les événements susceptibles de se produire. Dans un système synchrone, on s'intéressera uniquement aux états du système lors des événements. On ne s'intéresse pas en tant que tels aux états possibles du système entre deux événements successifs. Noter que tous les systèmes réels ne peuvent pas être modélisés de cette façon. Ainsi, les systèmes distribués, ou les systèmes asynchrones sortent en général du modèle synchrone, parce qu'on ne peut pas dater tous les événements sur une échelle commune, ou parce qu'on est incapable de prendre en compte la simultanéité d'événements. Le modèle synchrone (Benveniste *et al.*, 2003; Halbwachs, 1993) convient bien à la conception de contrôleurs implémentés sur un simple calculateur. En effet, les calculateurs fonctionnent en cycles et chaque cycle prend un temps non nul (pendant lequel le système à contrôler continue à évoluer).

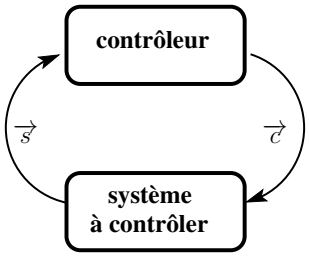

FIGURE 1 – Système contrôlé en boucle fermée

Un *système contrôlé en boucle fermée* (voir la figure 1), se compose d'une part du *système à contrôler* (une machine à laver, un ascenseur, un drone, un satellite ...) et d'autre part d'un *contrôleur* (contrôleur de machine à laver, pilote automatique, ...). Les entrées du contrôleur sont les sorties du système à contrôler. On les nomme également *observations*. Dans la figure 1, elles sont notées $\overrightarrow{s}$. Les entrées du système à contrôler sont les sorties du contrôleur. On les nomme également *actions*, *décisions*, ou *commandes*. Dans la figure 1, elles sont notées $\overrightarrow{c}$. Les observations parviennent au contrôleur par des *capteurs*, et les actions parviennent au système contrôlé par des *actionneurs*. S'il y a des opérateurs ou des agents dans la boucle (par exemple l'usager d'un ascenseur, l'opérateur d'un drone), on les considèrera comme faisant partie du système à contrôler. Leurs actions pourront apparaître dans notre modèle comme des entrées du contrôleur (des sorties du système à contrôler).

Le système a des évolutions continues (température, pression, position, vitesse ...) et discrètes (appui sur un bouton, détecteur d'alarme, sorties de calculateurs ...). Toutefois, dans notre modèle synchrone, les observations et les actions ne sont transmises que lors d'événements. En général les événements sont déclenchés périodiquement à l'initiative du contrôleur (modèle *clock-driven*). Il peuvent être aussi déclenchés par le système (modèle *event-driven*). Le contrôleur est lui aussi synchrone. Il évolue par étapes (ou cycles), et à chaque étape le contrôleur effectue une *réaction* : les capteurs sont lus et les actionneurs sont commandés pour l'étape courante. Le contrôle est «continuel» : s'il peut exister un état

initial, il n'y a pas à proprement parler d'état but à atteindre : le système ne cesse jamais de fonctionner. La plupart des systèmes temps-réel correspondent à ce type de fonctionnement.

Dans la figure 1, $\overrightarrow{s} : \mathbb{N} \to \mathcal{S}$ est le flot de sortie du système, et $\overrightarrow{c} : \mathbb{N} \to \mathcal{C}$ est le flot de sortie du contrôleur. $\mathcal{S}$ représente le domaine des valeurs que peut prendre le flot $\overrightarrow{s}$ à chaque instant, et $\mathcal{C}$ représente le domaine des valeurs que peut prendre le flot $\overrightarrow{c}$ à chaque instant.

On pourra utiliser une représentation factorisée à base de variables des flots considérés. Ainsi, $\mathcal{S}$ et $\mathcal{C}$ peuvent être des produits cartésiens de domaines simples (booléens, entiers ou flottants).

Le cadre formel que nous utilisons pour exprimer le problème de la synthèse est celui de la logique des prédicats du premier ordre, limitée à des symboles de fonctions constantes et en nombre fini, parfois nommée «logique relationnelle». Nous ferons usage parfois implicitement de la correspondance entre un prédicat et la relation associée (ensemble des tuples vérifiant le prédicat). Enfin, dans toute la suite, on lira $[A \implies B \implies C]$ comme $[A \implies [B \implies C]\,]$.

2.2 Données

Les données du problème de la synthèse de contrôleur consistent en un quintuplet $(\mathcal{S}, \mathcal{C}, T, R, I)$.

— $\mathcal{S}, \mathcal{C}$ sont, comme on l'a vu, les domaines de valeurs décrivant respectivement les états et les commandes possibles ou au moins représentables

— T, R et I sont des prédicats décrivant respectivement les transitions, les exigences et les états initiaux.

Pour alléger l'écriture, nous emploierons deux conventions :

— nous écrirons s pour s_i, c pour c_i, s' pour s_{i+1}, avec une quantification universelle $\forall i \in \mathbb{N}$ implicite

— la formule $Qs \in \mathcal{S} : F$, où F est un formule et Q un quantificateur $\forall$ ou $\exists$ sera notée simplement $Qs : F$ (le domaine est implicite), et de même $Qc \in \mathcal{C} : F$ sera noté simplement $Qc : F$.

État interne du contrôleur. Le contrôleur peut (et doit souvent) mémoriser des informations sur le passé. Ces informations constituent une forme d'état interne du contrôleur. Dans la mesure où elles sont réutilisées par le contrôleur, et sont donc des entrées de celui-ci, il est commode de les inclure dans l'état du système à contrôler s. Cette mémoire doit être bornée. C'est une façon d'inclure dans le modèle les contrôleurs à mémoire finie.

Transitions. Le modèle du système à contrôler (en «boucle ouverte») est décrit par le prédicat T :

$$\begin{aligned} T : \mathcal{S} \times \mathcal{C} \times \mathcal{S} \quad &\to \quad \mathbb{B} \\ \forall s, c, s' : T(s, c, s') \quad &= \quad \text{vrai si et seulement si une transition du système à contrôler est faisable} \\ &\qquad\quad \text{à partir de } s \text{ vers } s' \text{ avec la commande } c \end{aligned} \tag{1}$$

Ce prédicat condense toute la dynamique du système à contrôler. Il permet d'exprimer

— les états s ou s', ou les commandes c faisables physiquement

— les commandes applicables dans chaque état (couples (s, c) faisables)

— les transitions (s, c, s') proprement dites du système à contrôler.

Pour être tout à fait général, on n'interdit pas le non-déterminisme : à partir d'un état s, une commande c peut conduire à des états distincts.

Par commodité nous introduisons les notations suivantes :

$$\begin{aligned} \forall s : T(s, c) \quad &= \quad \text{vrai si et seulement si une transition du système à contrôler est faisable} \\ &\qquad\quad \text{depuis } s \text{ avec la commande } c \\ \forall s, c : [T(s, c) \quad &=_{\text{def}} \quad \exists s' : T(s, c, s')] \end{aligned} \tag{2}$$

et

$$\begin{aligned} \forall s : T(s) \quad &= \quad \text{vrai si et seulement si une transition du système à contrôler} \\ &\qquad\quad \text{est faisable depuis } s \\ \forall s : [T(s) \quad &=_{\text{def}} \quad \exists c, s' : T(s, c, s')] \end{aligned} \tag{3}$$

$T(s)$ exprime que s est à la fois

— un état faisable (appartenant à $\mathcal{S}$),

— un état non bloquant (car une commande et une transition sont possibles en cet état).
Par la suite, nous résumerons cela en disant que s est contrôlable.

Exigences. Le prédicat R (*requirements*) décrit les exigences, c'est-à-dire le comportement souhaité du système contrôlé :

$$\begin{aligned} R : \mathcal{S} \times \mathcal{C} \times \mathcal{S} &\rightarrow \mathbb{B} \\ \forall s : R(s, c, s') &= \text{vrai si et seulement si } (s, c, s') \text{ est une transition qui respecte les exigences.} \end{aligned} \tag{4}$$

R peut contenir les éléments d'une politique partielle.

Initialisation. Le prédicat I décrit les états possibles du système à l'instant initial.

$$\begin{aligned} I : \mathcal{S} &\rightarrow \mathbb{B} \\ \forall s : I(s) &= \text{vrai si et seulement si } s \text{ est un état initial possible du système à contrôler.} \end{aligned} \tag{5}$$

Hypothèses. Nous ferons les hypothèses suivantes sur le système à contrôler. *Hypothèse 1 : tout état atteint par une transition est contrôlable*. Autrement dit, toute transition du système à contrôler aboutissant à l'état s' peut être suivie d'une autre transition à partir de s'. Cette hypothèse correspond au fait que dans le monde réel, un système physique n'est jamais bloqué. Avec nos notations :

$$\forall s, c, s' : [T(s, c, s') \implies T(s')] \tag{6}$$

Cette hypothèse est naturellement complétée par celle-ci. *Hypothèse 2 : il existe au moins un état initial, et tout état initial est contrôlable* :

$$\exists s : I(s) \tag{7}$$

$$\forall s : [I(s) \implies T(s)] \tag{8}$$

2.3 Solution du problème de la synthèse

Politique. La solution d'un problème de synthèse $(\mathcal{S}, \mathcal{C}, T, R, I)$ est une *politique* Π *valide*. Une politique représente le comportement d'un contrôleur.

$$\begin{aligned} \Pi : \mathcal{S} \times \mathcal{C} &\rightarrow \mathbb{B} \\ \forall s, c : \Pi(s, c) &= \text{vrai si et seulement si la commande } c \text{ est possible} \\ &\quad \text{lorsque l'état du système est } s \text{ avec la politique } \Pi \end{aligned} \tag{9}$$

Noter le non-déterminisme possible. L'imbrication entre la politique et les transitions est représentée figure 2.

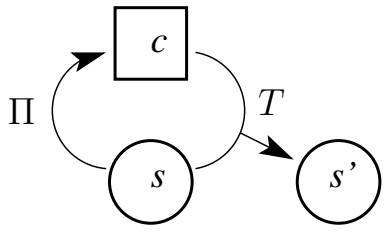

FIGURE 2 – Politique et transition dans un système dynamique à événements discret synchrone.

Par commodité nous introduisons la notation suivante :

$$\begin{aligned} \forall s : \Pi(s) &= \text{vrai si et seulement si une commande est définie par } \Pi \text{ dans l'état } s \\ \forall s : [\Pi(s) &=_{\text{def}} \exists c : \Pi(s, c)] \end{aligned} \tag{10}$$

Pour définir ce qu'est une politique valide, nous introduisons les prédicats suivants : *transition contrôlée* et *atteignabilité*.

Transitions du système contrôlé. Étant donné Π une politique potentielle, on définit le prédicat T_Π représentant les transitions possibles du système contrôlé par Π :

$$\begin{array}{rcl} T_\Pi : \mathcal{S} \times \mathcal{S} & \rightarrow & \mathbb{B} \\ \forall s, s' : T_\Pi(s, s') & = & \text{vrai si et seulement si une transition du système contrôlé est possible de } s \text{ vers } s' \\ \forall s, s' : [T_\Pi(s, s') & =_{\text{def}} & \exists c : [\Pi(s, c) \wedge T(s, c, s')] \,] \end{array} \tag{11}$$

Atteignabilité. L'atteignabilité est définie de manière inductive.

$$\begin{array}{rcl} A_\Pi : \mathcal{S} & \rightarrow & \mathbb{B} \\ \forall s : A_\Pi(s) & = & \text{vrai si et seulement si } s \text{ est atteignable depuis un état initial} \\ & & \text{par des transitions contrôlées} \\ \forall s : [I(s) & \Longrightarrow & A_\Pi(s)] \\ \forall\, 's, s : [A_\Pi('s) \wedge T_\Pi('s, s) & \Longrightarrow & A_\Pi(s)] \end{array} \tag{12}$$

On définit A_Π comme la relation minimale qui satisfait les deux formules précédentes. Autrement dit, $A_\Pi(s)$ est vrai si et seulement si $I(s)$ ou si dans le graphe dirigé $(\mathcal{S}, T_\Pi)$, il existe un chemin depuis un élément de I jusqu'à s.

2.4 Validité totale d'une politique

Le contrôleur doit être capable de gérer les états atteignables (existence d'une commande) et d'y répondre par une transition faisable (applicabilité) tout en respectant les exigences (conformité).
Une politique *totalement valide* est une politique satisfaisant les trois propriétés suivantes.

Existence. En tout état atteignable, la politique existe (une commande est définie).

$$\forall s : [A_\Pi(s) \implies \Pi(s)] \tag{13}$$

Applicabilité. En tout état atteignable, la politique est applicable : (la commande c d'un état s doit correspondre à une transition faisable du système à contrôler).

$$\forall s, c : [A_\Pi(s) \implies \Pi(s, c) \implies T(s, c)] \tag{14}$$

Noter que l'équation (14) peut être vérifiée sans que l'équation (13) le soit. Par exemple, une politique vide satisfait (14) mais pas (13) en général.

Conformité. En tout état atteignable, la politique respecte les exigences (toute transition contrôlée respecte les exigences).

$$\forall s, c, s' : [A_\Pi(s) \implies \Pi(s, c) \implies T(s, c, s') \implies R(s, c, s')] \tag{15}$$

2.5 Validité simple d'une politique

La propriété d'atteignabilité pose des problèmes de calculabilité. En effet, on ne peut pas simplement la vérifier. Il faut la construire inductivement (équation (12)) à partir des états initiaux. L'idée principale de cet article est de la remplacer par une propriété plus faible (plus souvent vérifiée) et de raisonner sur des transitions à partir des états vérifiant cette propriété plus faible. De la sorte, on caractérise des politiques plus définies que strictement nécessaire, mais plus faciles à calculer. L'inconvénient, on le verra, est qu'une politique basée sur une propriété plus faible peut ne pas exister, alors qu'il en existerait avec la propriété d'atteignabilité proprement dite. Plusieurs choix possibles s'offrent à nous pour cette propriété plus faible $F(s)$, pourvu que $A_\Pi(s) \implies F(s)$. Nous avons choisi ici de remplacer l'atteignabilité par la contrôlabilité $T(s)$.
La définition d'une politique *simplement* valide est la même que celle d'une politique totalement valide, à ceci près qu'on remplace donc l'atteignabilité $A_\Pi(s)$ par la contrôlabilité $T(s)$. L'ensemble des états s qui satisfont $T(s)$ sont des états contrôlables, des entrées possibles du contrôleur, hors atteignabilité. On demande que le contrôleur soit capable de gérer ces états (existence) et d'y répondre par une transition contrôlée (applicabilité) tout en respectant les exigences (conformité).
Une politique *simplement valide* est donc une politique satisfaisant les trois propriétés suivantes.

Existence. En tout état contrôlable, la politique existe (une commande est définie).

$$\forall s : [T(s) \implies \Pi(s)] \tag{16}$$

Applicabilité. En tout état contrôlable, la politique est applicable (la commande c d'un état s doit correspondre à une transition faisable du système à contrôler).

$$\forall s, c : [T(s) \implies \Pi(s, c) \implies T(s, c)] \tag{17}$$

Conformité. En tout état contrôlable, la politique respecte les exigences (toute transition contrôlée respecte les exigences).

$$\forall s, c, s' : [T(s) \implies \Pi(s, c) \implies T(s, c, s') \implies R(s, c, s')] \tag{18}$$

Toute politique simplement valide est totalement valide.

> Démonstration : du fait de la similitude entre les propriétés «totales» et «simples», il suffit de montrer que, pour toute politique simplement valide Π, $\forall s : [A_\Pi(s) \implies T(s)]$. Soit donc Π une politique simplement valide, et un état s vérifiant $A_\Pi(s)$. Si $I(s)$ alors $T(s)$ d'après l'équation (8). Sinon, c'est que s est obtenu par une transition (12), et alors $T(s)$ par l'hypothèse de non blocage du système à contrôler (équation (6)). Dans tous les cas, on a bien $T(s)$.

Il peut exister une politique totalement valide sans qu'il existe de politique simplement valide. Voir l'exemple du chauffe-eau section 3.

2.6 Résolution du problème de la synthèse avec validité simple

On cherche maintenant à définir de façon constructive une politique Π simplement valide.
On montre assez facilement que les propriétés d'applicabilité et de conformité simples (équations (17) et (18)) sont équivalentes à :

$$\forall s, c : [T(s) \implies \Pi(s, c) \implies [T(s, c) \wedge \forall s' : [T(s, c, s') \implies R(s, c, s')]]] \tag{19}$$

Intéressons nous aux états s contrôlables c'est-à-dire tels que $T(s)$. Pour les autres états, l'équation précédente montre que la politique peut ne pas exister ou être quelconque. La politique cherchée satisfait

$$\forall s, c : [\Pi(s, c) \implies [T(s, c) \wedge \forall s' : [T(s, c, s') \implies R(s, c, s')]]] \tag{20}$$

Cette forme exprime de manière compacte que si la commande en s est c, alors c est applicable à s, et toutes les transitions du système par (s, c) respectent les exigences.
La plus large relation Π (celle qui contient le plus de tuples) qui satisfait applicabilité et conformité est donc définie par

$$\forall s, c : [\Pi(s, c) = [T(s, c) \wedge \forall s' : [T(s, c, s') \implies R(s, c, s')]]] \tag{21}$$

ou, sous une forme équivalente :

$$\forall s, c : [\Pi(s, c) = [T(s, c) \wedge \neg\exists s' : [T(s, c, s') \wedge \neg R(s, c, s')]]] \tag{22}$$

Le prédicat $\exists s' : [T(s, c, s') \wedge \neg R(s, c, s')]$ représente les tuples interdits de toute politique valide.
Ayant obtenu une politique Π satisfaisant l'équation précédente (voir la section 5), il restera à vérifier la propriété d'existence (équation (16)).

Cas particulier. Lorsque R ne dépend pas de s' mais seulement de s et c, alors l'expression
$\forall s' : [T(s, c, s') \implies R(s, c, s')]$ est équivalente à
$[\exists s' : T(s, c, s')] \implies R(s, c, s')$ (car s' n'a pas d'occurence libre dans R).
L'équation (21) devient alors
$\forall s, c : [\Pi(s, c) = T(s, c) \wedge [\, [\exists s' : T(s, c, s')] \implies R(s, c, s')] \,]$
soit, par définition de $T(s, c)$ et par $[A \wedge [A \implies B]] \equiv [A \wedge B]$:
$\forall s, c : [\Pi(s, c) = [\exists s' : T(s, c, s')] \wedge R(s, c, s')]$ soit simplement

$$\forall s, c : [\Pi(s, c) = \exists s' : [T(s, c, s') \wedge R(s, c, s')]] \tag{23}$$

3 Exemple 1 : le contrôleur d'un chauffe-eau

Cet exemple fictif simple quoique non complètement trivial illustre le cadre et la méthode proposés. Il montre aussi la différence entre politique totalement et simplement valide.

Il s'agit de contrôler le brûleur d'un chauffe-eau, de manière à maintenir la température de l'eau dans une plage correcte. Les états possibles du système à contrôler sont

$$\mathcal{S} = \{\texttt{Init}, \texttt{Frozen}, \texttt{Cold}, \texttt{HotInf}, \texttt{HotSup}, \texttt{Boiling}\}.$$

L'état `Init` est l'état initial. Les états `Boiling` et `Frozen` ne sont pas autorisés (on désire que le chauffe-eau n'entre jamais dans ces états). Les commandes possibles sont $\mathcal{C} = \{\texttt{On}, \texttt{Off}\}$ selon que le brûleur est activé ou non pendant le cycle de commande. Les transitions T (comportement du système à contrôler) sont décrites par la relation suivante, représentée graphiquement figure 3.

s	c	s'
Init	Off	Init
Init	Off	Cold
Init	On	Init
Init	On	HotInf
Frozen	Off	Frozen
Frozen	On	Frozen
Cold	Off	Cold
Cold	Off	Frozen
Cold	On	Cold
Cold	On	Frozen
HotInf	Off	Cold
HotInf	Off	HotInf
HotInf	On	HotInf
HotInf	On	HotSup
HotSup	Off	HotSup
HotSup	Off	HotInf
HotSup	On	Boiling
HotSup	On	HotSup
Boiling	Off	HotSup
Boiling	Off	Boiling
Boiling	On	Boiling

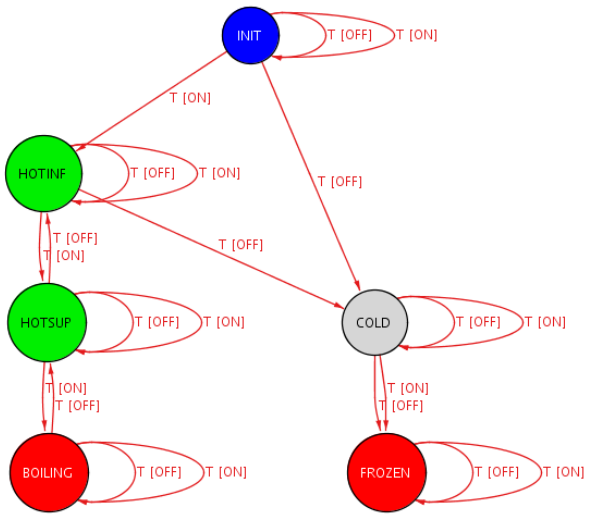

FIGURE 3 – Exemple du chauffe-eau : la relation T (transitions du système à contrôler).

L'idée est que, à partir des états `Init`, `HotInf` et `HotSup`, en chauffant, on peut soit rester dans le même état, soit parvenir à un état plus chaud. Sans chauffer, on reste dans le même état ou on refroidit. On ne peut pas être plus chaud que `Boiling`, et on ne peut pas être plus froid que `Frozen`. L'état

`Cold` est particulier : a priori il est autorisé, mais si on arrive dans cet état, on ne peut qu'y rester ou tomber dans `Frozen` quelle que soit la commande.
Les exigences s'expriment ainsi :

$$\forall s, c, s' : [R(s, c, s') = [OK(s) \implies OK(s')]\,] \tag{24}$$

$$\forall s : OK(s) = s \in \{\texttt{Init}, \texttt{Cold}, \texttt{HotInf}, \texttt{HotSup}\} \tag{25}$$

Dans cet exemple, il existe une politique totalement valide :

$$\Pi = \{(\texttt{Init}, \texttt{On}), (\texttt{HotInf}, \texttt{On}), (\texttt{HotSup}, \texttt{Off})\} \tag{26}$$

mais pas de politique simplement valide, pour la raison que `Cold` est contrôlable, mais qu'il existe des transitions inévitables de `Cold` vers `Frozen`, et donc aucune commande n'est applicable à `Cold` satisfaisant les exigences. `Cold` est un état «bloquant». Si on interdit cet état par la définition alternative

$$\forall s : OK(s) = s \in \{\texttt{Init}, \texttt{HotInf}, \texttt{HotSup}\} \tag{27}$$

alors il existe une politique simplement valide, qui inclut la précédente, par exemple celle-ci :

$$\Pi = \{(\texttt{Init}, \texttt{On}), (\texttt{HotInf}, \texttt{On}), (\texttt{HotSup}, \texttt{Off}), (\texttt{Cold}, \texttt{On}), (\texttt{Boiling}, \texttt{On}), (\texttt{Frozen}, \texttt{Off})\} \tag{28}$$

Voir l'annexe A pour un traitement complet de l'exemple avec le système Alloy.

4 Exemple 2 : le contrôleur d'un robot explorateur

Pour illuster la méthode de synthèse proposée dans cet article, nous l'appliquons à un système imaginaire mais nettement plus complexe et plus réaliste que l'exemple précédent du chaufe-eau, et plus proche des cibles applicatives pressenties pour la méthode. Il illustre de plus le cas où les états et les commandes sont décrits par des domaines combinatoires, et le cas où les états ne sont pas totalement observables.

4.1 Description informelle

Le système à contrôler est un robot équipé de roues dont les moteurs peuvent être commandés séparément, de capteurs d'obstacles et de capteurs de température. Le contrôleur du robot doit le diriger vers des zones froides tout en évitant des obstacles éventuels. Plus précisément, le robot possède 7 capteurs :
— deux capteurs de toucher ("*whiskers*"), avant gauche et avant droit, un capteur de toucher à l'arrière
— deux capteurs de roue, permettant de savoir dans quel sens tourne chacune des roues effectivement
— deux capteurs de température situés à l'avant de chaque côté.
Le comportement requis du robot est le suivant : en absence d'obstacles, si la température d'un côté est inférieure à celle de l'autre, le contrôleur doit diriger le robot vers le côté le plus froid. Si un obstacle est détecté, le robot doit être dirigé de manière à l'éviter. Il ne doit pas rester bloqué. Enfin, le contrôleur doit détecter le fait qu'une commande des moteurs puisse ne pas être suivi d'effet, et signaler ce cas.

4.2 États et commandes

Les états du robot (ensemble $\mathcal{S}$) sont décrits par des combinaisons de valeurs des variables suivantes :
— $leftTouch, rightTouch, rearTouch \in \mathbb{B}$: touchers des capteurs droit, gauche et arrière
— $leftWheel, rightWheel \in \{-1, 0, 1\}$: direction du mouvement des roues (-1 : vers l'arrière ; 0 : stoppé ; 1 : vers l'avant).
— $deltaTemp \in \{-1, 0, 1\}$: le signe de la différence de température entre les capteurs de température gauche et droit.
— $failLeftMotor, failRightMotor \in \mathbb{B}$: le moteur droit ou gauche est défaillant. Ces variables ne sont pas observables (le contrôleur ne les voit pas).
— $pCmdLeftMotor, pCmdRightMotor \in \{-1, 0, 1\}$: mémoires des commandes du cycle précédent.
Les commandes du robot sont décrits par des combinaisons de valeurs des variables suivantes :
— $cmdLeftMotor, cmdRightMotor \in \{-1, 0, 1\}$: commandes des aux moteurs ; (-1 : vers l'arrière ; 0 : stopper ; 1 : vers l'avant).
— $outOfService \in \mathbb{B}$: un mauvais fonctionnement des moteurs est détecté.
Note : $cmdLeftMotor$ et $cmdRightMotor$ sont des commandes proprement dites, tandis que $outOfService$ est un signal destiné à un contrôleur de niveau supérieur ou à un opérateur humain.

4.3 Contraintes modélisant la dynamique propre du robot : la relation T

Les contraintes suivantes constituent la définition de la relation de transition T.
Si un moteur n'est pas défaillant, alors le mouvement de la roue correspondante suivra, à l'instant suivant, la commande émise à l'instant présent. Si le moteur est défaillant, la roue sera bloquée.

$$\forall s, c, s' : \tag{29}$$
$$(\neg s.failLeftMotor) \implies (s'.leftWheel = c.cmdLeftMotor) \tag{30}$$
$$(s.failLeftMotor) \implies (s'.leftWheel = 0) \tag{31}$$
$$(\neg s.failRightMotor) \implies (s'.rightWheel = c.cmdRightMotor) \tag{32}$$
$$(s.failRightMotor) \implies (s'.rightWheel = 0) \tag{33}$$

Il est physiquement impossible de toucher les trois capteurs simultanément.

$$\forall s : \neg(s.rightTouch \wedge s.leftTouch \wedge s.rearTouch) \tag{34}$$
$$\forall s' : \neg(s'.rightTouch \wedge s'.leftTouch \wedge s'.rearTouch) \tag{35}$$

Pour faciliter l'expression de T, nous utiliserons des variables auxiliaires, considérées comme des variables de commande : $c.doTurnLeft, c.doTurnRight, c.doForw, c.doBack, c.doStop \in \mathbb{B}$: le contrôleur commande de tourner à gauche, à droite, d'aller en avant, en arrière ou de stopper. Ces variables sont définies par les équations suivantes :

$$\forall c :$$
$$(c.cmdRightMotor < c.cmdLeftMotor) \equiv c.doTurnRight \tag{36}$$
$$(c.cmdLeftMotor < c.cmdRightMotor) \equiv c.doTurnLeft \tag{37}$$
$$(c.cmdLeftMotor + c.cmdRightMotor = 2) \equiv c.doForw \tag{38}$$
$$(c.cmdLeftMotor + c.cmdRightMotor = -2) \equiv c.doBack \tag{39}$$
$$(c.cmdLeftMotor = 0 \wedge c.cmdRightMotor = 0) \equiv c.doStop \tag{40}$$

Enfin, deux dernières contraintes correspondent à la mémorisation de la commande précédente.

$$\forall c, s' :$$
$$s'.pCmdLeftMotor = c.cmdLeftMotor \tag{41}$$
$$s'.pCmdRightMotor = c.cmdRightMotor \tag{42}$$

4.4 Contraintes modélisant le comportement requis du robot : la relation R

Si $deltaTemp = 0$ et aucun toucher droit ni gauche, aller en avant. Si $deltaTemp = 1$ et aucun toucher droit ni gauche, aller à droite. Si $deltaTemp = -1$ et aucun toucher droit ni gauche, aller à gauche. Si le robot touche d'un côté et pas de l'autre, alors tourner du côté libre. Si le robot touche des deux côtés, reculer. Si le robot touche à l'arrière, ne pas reculer ni stopper.

$$\forall s, c :$$
$$(s.deltaTemp = 0 \wedge \neg s.rightTouch \wedge \neg s.leftTouch) \implies c.doForw \tag{43}$$
$$(s.deltaTemp = 1 \wedge \neg s.rightTouch \wedge \neg s.leftTouch) \implies c.doTurnRight \tag{44}$$
$$(s.deltaTemp = -1 \wedge \neg s.rightTouch \wedge \neg s.leftTouch) \implies c.doTurnLeft \tag{45}$$
$$(s.rightTouch \wedge \neg s.leftTouch) \implies c.doTurnLeft \tag{46}$$
$$(\neg s.rightTouch \wedge s.leftTouch) \implies c.doTurnRight \tag{47}$$
$$(s.rightTouch \wedge s.leftTouch) \implies c.doBack \tag{48}$$
$$s.rearTouch \implies \neg c.doBack \tag{49}$$
$$s.rearTouch \implies \neg c.doStop \tag{50}$$

Détection de panne :

$$\forall s, c : c.outOfService \equiv \tag{51}$$
$$(s.pCmdLeftMotor \neq s.leftWheel) \vee (s.pCmdRightMotor \neq s.rightWheel)$$

5 Synthèse d'une politique simplement valide dans le cadre des problèmes de satisfaction de contraintes

5.1 Représentation combinatoire des états et des commandes

Comme dans l'exemple du robot explorateur, il est naturel de représenter chaque état $s \in \mathcal{S}$ par N_s attributs $s.\mathrm{a}_1, ...s.\mathrm{a}_{N_s}$. Chaque attribut peut prendre un nombre fini de valeurs dans un domaine noté $\mathcal{D}_s$ (pour rester simple dans cet exposé, on a supposé que les domaines des attributs sont identiques ; la généralisation est aisée). Le domaine $\mathcal{S}$ est ainsi représenté de manière combinatoire par $\mathcal{D}_s^{N_s}$ valeurs possibles. Toutes ces valeurs ne sont pas nécessairement faisables.

De manière similaire, chaque commande $c \in \mathcal{C}$ est représentée par N_c attributs $c.\mathrm{a}_1, ...c.\mathrm{a}_{N_c}$. Chaque attribut peut prendre un nombre fini de valeurs dans un domaine noté $\mathcal{D}_c$. Le domaine $\mathcal{C}$ est ainsi représenté de manière combinatoire par $\mathcal{D}_c^{N_c}$ valeurs possibles. Toutes ces valeurs ne sont pas non plus nécessairement faisables.

5.2 Réseaux de contraintes et programmation par contraintes

Un réseau de contraintes (Rossi *et al.*, 2006) est un triplet $(\mathrm{X}, \mathrm{D}, \mathrm{C})$, où $\mathrm{X} = \{\mathrm{X}_1, ..., \mathrm{X}_n\}$ est un ensemble de variables de domaines respectifs $\mathrm{D} = \{\mathrm{Dom}(\mathrm{X}_1), ...\mathrm{Dom}(\mathrm{X}_n)\}$, et C est un ensemble de contraintes. Chaque contrainte est définie par la séquence des variables de X sur laquelle elle porte, et par une relation sur ces variables donnant l'ensemble des tuples autorisés par la contrainte. Une solution d'un réseau de contraintes est un tuple de valeurs des variables satisfaisant toutes les contraintes. Le problème de satisfaction de contraintes (CSP) est le suivant : étant donné un réseau de contraintes, existe-t-il une solution ? En général on s'intéresse non seulement au problème de l'existence d'une solution, mais aussi à celui d'en obtenir une ou même toutes.

La programmation par contrainte est un cadre applicatif qui permet de résoudre ce type de problèmes. De nombreux outils de programmation par contraintes performants existent aujourd'hui, par exemple OPL (Van Hentenryck, 2000), Choco (the `choco` team, 2008). Les avantages de la programmation par contraintes (par rapport à des moteurs de résolution basés sur le problème SAT par exemple) sont :

— la possibilité d'utiliser directement des domaines autres que booléens (entiers, flottants)

— l'existence d'algorithmes dédiés à des sous-problèmes particuliers (contraintes globales).

5.3 Méthode de synthèse

Pour construire une politique, nous utiliserons le principe général suivant. Soit $P(X, Y)$ un prédicat portant sur des variables des ensembles X et Y, avec X et Y disjoints. Supposons que la relation support de $P(X, Y)$ s'exprime comme l'ensemble des solutions d'un réseau de contraintes **P** sur X et Y. Alors l'ensemble des tuples sur les variables de Y qui vérifient $\exists X : P(X, Y)$, peut être calculé en collectant les solutions de **P** projetées sur Y. Une table de hachage est une structure bien adaptée pour cette collecte.

Le processus de construction de la politique Π fera intervenir la construction de relations intermédiaires obtenues comme on vient de l'expliquer comme ensembles de solutions de réseaux de contraintes bien choisis, ou par projections d'autres relations, ou par des différences d'ensembles. Les projections et différences d'ensembles peuvent se faire par opérations sur des tables de hachage.

Les réseaux de contraintes dont nous collecterons les solutions sont construits sur les variables $\mathrm{X} = \{s.\mathrm{a}_1 \cdots s.\mathrm{a}_{N_s}, c.\mathrm{a}_1 \cdots c.\mathrm{a}_{N_c}, s'.\mathrm{a}_1 \cdots s'.\mathrm{a}_{N_s}\}$ où les $s'.\mathrm{a}_i$ représentent les attributs de l'état successeur s', soit $2N_s + N_c$ variables. Ces variables ont pour domaines $\mathrm{Dom}(s.\mathrm{a}_k) = \mathrm{Dom}(s.\mathrm{a}'_k) = \mathcal{D}_s$ et $\mathrm{Dom}(c.\mathrm{a}_k) = \mathcal{D}_c$.

Comme dans les exemples précédents, nous supposerons les prédicats T et R donnés sous forme explicite comme des relations, ou implicitement sous forme de combinaisons de contraintes sur les variables X.

On cherche à construire une politique non déterministe d'après l'équation (22). La construction s'effectue en plusieurs étapes.

1. Construire le réseau de contraintes sur X avec pour contraintes celles qui correspondent aux prédicats $T(s, c, s')$ et $\neg R(s, c, s')$. Collecter les solutions de ce CSP et les projeter sur (s, c). On obtient l'ensemble des tuples interdits de toute politique valide.
2. Calculer l'ensemble des solutions du CSP dont la seule contrainte est T, projeter cet ensemble sur (s, c) pour obtenir les tuples de $T(s, c)$, en ôter les tuples obtenus à l'étape précédente (différence d'ensemble). On obtient les tuples d'une politique Π potentiellement valide.

3. Vérifier la propriété d'existence de la politique (équation (16)) : calculer les ensembles $T(s)$ et $\Pi(s)$ par projections, et vérifier que le premier est inclus dans le second. L'état de $T(s)$ qui ne serait pas inclus dans $\Pi(s)$ est un état «bloquant».

Si la dernière vérification échoue, alors la synthèse d'une politique simplement valide échoue. On peut tenter de modifier T de manière à ôter les états bloquants de $T(s)$, comme dans l'exemple du chauffe-eau, si toutefois cela est compatible avec la sémantique de l'application.

Remarque : il est intéressant d'exprimer les exigences directement sous la forme d'une contrainte $\neg R$ conjonctive pour faciliter le calcul des tuples interdits.

Cas particulier : lorsque R ne dépend pas de s', la politique vérifie l'équation (23). En ce cas, les tuples de la politique s'obtiennent très simplement comme l'ensemble des solutions d'un seul réseau de contraintes contenant les contraintes de T et de R, projetés sur (s, c). Il reste encore à vérifier l'existence (équation (16)).

La méthode de synthèse produit une politique non déterministe. On peut donc se poser la question du choix d'une commande lorsque plusieurs sont possibles. Il existe au moins deux façons de choisir :

— à l'exécution, choisir aléatoirement une des commandes possibles ; cette façon de faire conviendrait bien à l'exemple du robot exploreur

— au moment de l'implantation du contrôleur, choisir les commandes qui dans leur ensemble simplifient la logique du contrôleur, satisfont ou optimisent des critères non encore pris en compte dans la spécification.

Nous avons appliqué la méthode aux deux exemples précédents, avec les systèmes de programmation par contrainte OPL (Van Hentenryck, 2000) et Choco (the `choco` team, 2008). Les calculs sont très rapides (mais les exemples sont petits).

6 Conclusion et perspectives

En synthèse de contrôleur «classique», la construction d'un contrôleur est basée, implicitement ou non, sur la propriété d'atteignabilité : on exige la conformité aux spécifications réduite aux seuls états atteignables. Or la propriété d'atteignabilité pose des problèmes de calculabilité importants.

L'idée principale de cet article est de remplacer la propriété d'atteignabilité par une propriété plus faible (plus souvent vérifiée) mais facile à vérifier, et de raisonner sur des transitions à partir des états vérifiant cette propriété plus faible. On caractérise ainsi ce que nous avons nommé une politique *simplement valide*, par opposition à *totalement valide* lorsque la stricte atteignabilité est conservée. Toute politique simplement valide l'est aussi totalement. Les politiques simplement valides sont plus définies que strictement nécessaire, mais plus faciles à calculer. L'inconvénient est qu'une politique simplement valide peut ne pas exister, alors qu'il existerait une politique totalement valide. Il nous semble toutefois que, dans la plupart des cas réels, il existe souvent une politique simplement valide. C'est une hypothèse à consolider par l'expérience.

Nous avons proposé une méthode de synthèse de contrôleur décrite dans le cadre d'une logique relationnelle. La programmation par contraintes nous semble un outil de choix pour implémenter la méthode et calculer une politique (comportement du contrôleur) sous forme d'une relation.

Ce travail est préliminaire et doit être consolidé. Il devrait se poursuivre dans plusieurs directions, esquissées maintenant.

En premier lieu, des expérimentations doivent être menées sur des applications plus importantes et plus proches de systèmes réels, afin de valider l'approche.

Le lien avec la logique temporelle, cadre formel dans lequel la synthèse «classique» exprime les propriétés requises, doit être établi. La démarche proposée revient apparemment à considérer uniquement des propriétés de sûreté (*safety*) dans les spécifications.

Nous mentionnons dans l'article l'usage de tables de hachage pour construire et accéder rapidement aux relations manipulées. D'autres structures de données sont utilisables. On pense naturellement aux *Ordered Binary Decision Diagrams* (Bryant, 1986) avec toutefois l'inconvénient d'être fondamentalement restreints aux valeurs binaires.

D'autres cadres de résolution pourraient aussi bien supporter la méthode. Nous pensons bien sûr aux moteurs de résolution basés sur SAT [2]. Toutefois encore, ces systèmes sont fondamentalement basés sur la manipulation de valeurs binaires.

2. Par exemple SAT4J `http://www.sat4j.org/` ou MiniSAT `http://minisat.se/`

Nous avons proposé dans l'article d'utiliser la propriété de contrôlabilité $T(s)$ pour remplacer celle d'atteignabilité $A_\Pi(s)$. Mais d'autres choix sont tout à fait possibles. Par exemple une forme d'atteignabilité réduite pourrait être exploitée de la même façon, définie formellement par

$$\forall s : [Ar(s) = I(s) \vee \exists c, {}'s : T({}'s, c, s)] \tag{52}$$

($Ar(s)$ si s est initial ou si une transition est possible vers s).

Une autre question intéressante est celle-ci : comment poursuivre lorsqu'il n'existe pas de politique simplement valide ? Quelles modifications acceptables des exigences pourraient supprimer les états bloquants et donner des chances de restaurer l'existence d'une politique simplement valide ?

Enfin, reste le problème concret de l'exploitation en ligne de la politique. La méthode proposée fournit une politique non déterministe sous forme d'une relation. Les contrôleurs sont en pratique soumis à de fortes contraintes temps-réel et à des capacités limitées en mémoire. Il faut donc transformer cette relation en une collection de fonctions, une pour chaque actionneur (attribut de commande), très rapidement calculables en ligne et représentées de manière compacte. Une possibilité est, à la manière de Bloem *et al.* (2007) d'exploiter le non déterminisme de la politique calculée et le fait qu'elle peut être quelconque pour les états non faisables ou non contrôlables.

A Modélisation du problème du chauffe-eau avec Alloy

Nous avons utilisé le langage de modélisation Alloy (`alloy.mit.edu`) pour formaliser l'exemple du chauffe-eau. Alloy permet de traiter les deux versions : avec validités totale et simple.

Voici d'abord le modèle avec validité totale. Remarquer que la première partie est totalement générique (ne dépend pas de l'exemple).

```
1  module synthese
2
3  abstract sig Control {}
4
5  abstract sig State {
6      T : Control →State, // Transitions of the non controlled system
7      R : Control →State, // Required transitions
8      P : set Control, // Politic (controller)
9      C : set State // transitions of the Controlled system (T_\Pi in the article)
10 }
11
12 // definition of the transitions of the controlled system
13 fact { C = {s,s':State | some c: Control | s→c in P and s→c→s' in T} }
14
15 // definition of accessibility
16 pred A [s:State] { s in *C[INIT] }
17
18 // property 1 : existence of the policy
19 fact { all s:State |
20           A[s] implies some P[s] }
21
22 // property 2 : applicability of the policy
23 fact { all s:State, c:Control |
24          A[s] implies { s→c in P implies { some s':State | s→c→s' in T } } }
25
26 // property 3 : respect of requirements (conformity)
27 fact { all s,s':State, c:Control |
28          A[s] implies { s→c in P implies { s→c→s' in T implies s→c→s' in R } } }
29
30
31 // hypothesis : init states respect requirements
32 check { some c:Control, s':State | INIT→c→s' in R }
33
34
35 // ------------------------------
36
```

```
37 // water−heater
38
39 one sig ON, OFF extends Control {}
40 one sig INIT, FROZEN, COLD, HOTINF, HOTSUP, BOILING extends State {}
41
42 // transitions of the non controlled system
43 fact {
44     INIT·T[OFF] = { INIT + COLD }
45     INIT·T[ON] = { INIT + HOTINF }
46
47     FROZEN·T[OFF] = FROZEN
48     FROZEN·T[ON] = FROZEN
49
50     // COLD is contyrolable but it can be followed by FROZEN which is not controlable
51     COLD·T[OFF] = { COLD + FROZEN }
52     COLD·T[ON] = { COLD + FROZEN }
53
54     HOTINF·T[OFF] = { COLD + HOTINF }
55     HOTINF·T[ON] = { HOTINF + HOTSUP }
56
57     HOTSUP·T[OFF] = { HOTINF + HOTSUP }
58     HOTSUP·T[ON] = { HOTSUP + BOILING }
59
60   BOILING·T[OFF] = { HOTSUP + BOILING }
61   BOILING·T[ON] = BOILING
62 }
63
64
65 // requirements
66
67 pred OK[s:State] { s in INIT + COLD + HOTINF + HOTSUP }
68
69 fact { all s,s':State, c:Control | s→c→s' in R iff ( OK[s] implies OK[s'] ) }
70
71 run {}
```

Alloy Analyzer permet de construire une ou plusieurs instances satisfaisant le modèle. Ici il produit une politique totalement valide représentée graphiquement figure 4.

Voici maintenant un modèle pour la synthèse de politique simplement valide, appliquée à l'exemple du chauffe-eau. Remarquer que COLD ne fait pas partie des états autorisés (c'est à cette condition que la synthèse simplement valide est possible pour cet exemple).

```
1  module synthese
2
3  abstract sig Control {}
4
5  abstract sig State {
6      T : Control →State, // Transitions of the non controlled system
7      R : Control →State, // Required transitions
8      P : set Control // Politic (controller)
9  }
10
11
12 // definition of controlability ( T(s) dans l'article, C(s) ici )
13 pred C [s:State] { some c:Control, s':State | s→c→s' in T }
14
15 // property 1 : existence of the policy
16 fact { all s:State |
17           C[s] implies some P[s] }
18
19 // property 2 : applicability of the policy
20 fact { all s:State, c:Control |
```

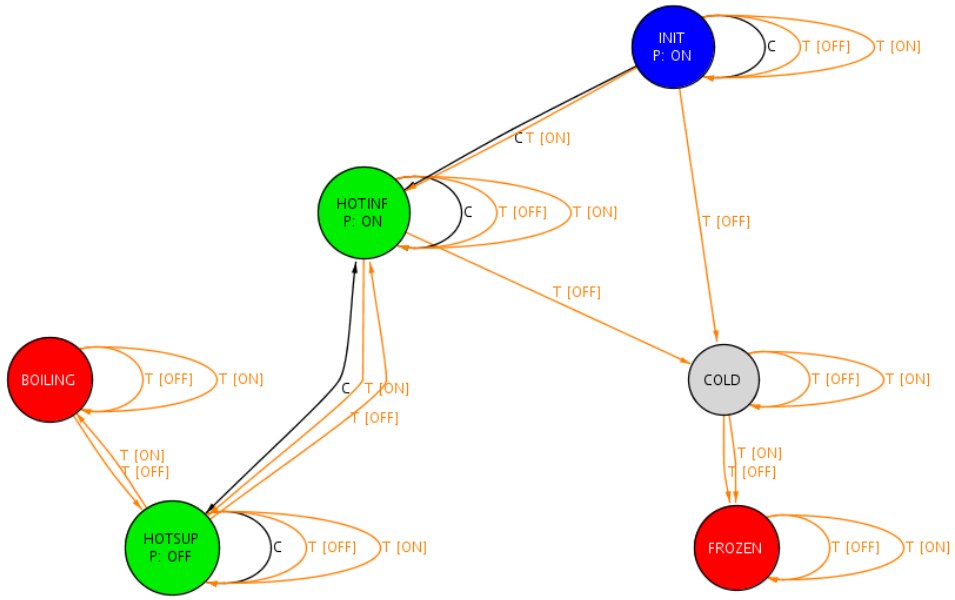

FIGURE 4 – Exemple du chauffe-eau : une politique totalement valide.

```
21          C[s] implies { s→c in P implies { some s':State | s→c→s' in T } } }
22
23  // property 3 : respect of requirements (conformity)
24  fact { all s,s':State, c:Control |
25          C[s] implies { s→c in P implies { s→c→s' in T implies s→c→s' in R } } }
26
27
28  // hypothesis : init states respect requirements
29  check { some c:Control, s':State | INIT→c→s' in R } expect 0
30
31
32  // ----------------------------------------
33
34  // water-heater
35
36  one sig ON, OFF extends Control {}
37  one sig INIT, FROZEN, COLD, HOTINF, HOTSUP, BOILING extends State {}
38
39  // transitions of the non controlled system
40  fact {
41      INIT·T[OFF] = { INIT + COLD }
42      INIT·T[ON] = { INIT + HOTINF }
43
44      FROZEN·T[OFF] = FROZEN
45      FROZEN·T[ON] = FROZEN
46
47      // COLD is controlable but it can be followed by FROZEN which is not controlable
48      COLD·T[OFF] = { COLD + FROZEN }
49      COLD·T[ON] = { COLD + FROZEN }
50
51      HOTINF·T[OFF] = { COLD + HOTINF }
52      HOTINF·T[ON] = { HOTINF + HOTSUP }
53
54      HOTSUP·T[OFF] = { HOTINF + HOTSUP }
55      HOTSUP·T[ON] = { HOTSUP + BOILING }
```

```
56
57    BOILING·T[OFF] = { HOTSUP + BOILING }
58    BOILING·T[ON] = BOILING
59  }
60
61
62  // requirements
63
64  pred OK[s:State] { s in INIT + HOTINF + HOTSUP }
65  // pred OK[s:State] { s in INIT + COLD + HOTINF + HOTSUP }
66
67  fact { all s,s':State, c:Control | s→c→s' in R iff ( OK[s] implies OK[s'] ) }
68
69  run {}
```

La figure 5 représente la politique simplement valide obtenue.

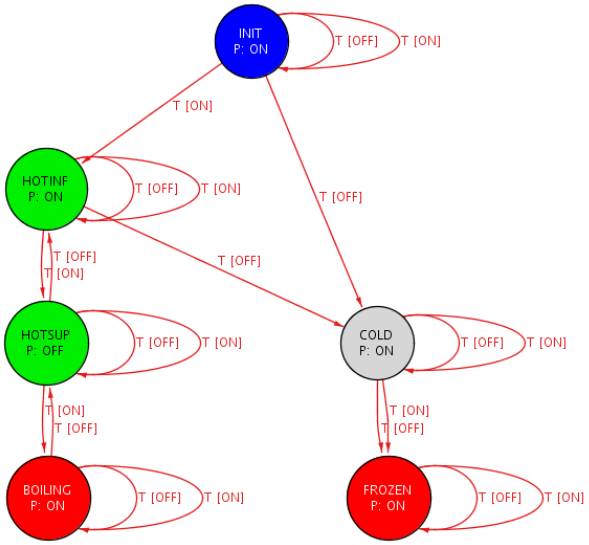

FIGURE 5 – Une politique simplement valide pour l'exemple du chauffe-eau.

Noter que l'exemple du robot exploreur n'a pu être traité avec Alloy du fait de sa plus grande complexité (impossibilité d'obtenir des instances en temps raisonnable).

Références

ARNOLD A., VINCENT A. & WALUKIEWICZ I. (2003). Games for synthesis of controllers with partial observation. *Theoretical Computer Science*, **303**(1), 7–34.

ASARIN E., MALER O. & PNUELI A. (1995). Symbolic controller synthesis for discrete and timed systems. In *Hybrid Systems II, LNCS 999*, p. 1–20 : Springer.

BEHRMANN G., DAVID A., & LARSEN K. G. (2004). A Tutorial on UPPAAL. In *LNCS 3185*, p. 200–236. Springer-Verlag. `http://www.cs.aau.dk/~adavid/tiga`

BENVENISTE A., CASPI P., EDWARDS S., HALBWACHS N., LE GUERNIC P. & DE SIMONE R. (2003). The Synchronous Languages Twelve Years Later. *Proceedings of the IEEE*, **91**(1), 64–83.

BLOEM R., GALLER S., JOBSTMANN B., PITERMAN N., PNUELI A. & WEIGLHOFER M. (2007). Specify, Compile, Run : Hardware from PSL. *Electronic Notes in Theoretical Computer Science*, **190**(4), 3 – 16. Proceedings of the Workshop on Compiler Optimization meets Compiler Verification (COCV 2007).

BRYANT R. E. (1986). Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, **C-35**(8), 677–691.

CASSEZ F. & MARKEY N. (2007). Contrôle et implémentation des systèmes temporisés. In *Actes de la 5ème École Temps-Réel (ETR'07)*, p. 111–123, Nantes, France.

CIMATTI A., PISTORE M., ROVERI M. & TRAVERSO P. (2003). Weak, strong, and strong cyclic planning via symbolic model checking. *Artificial Intelligence*, **147**, 35–84. Also available as ITC-irst technical report 0104-11.

CIMATTI A., ROVERI M. & BERTOLI P. (2004). Conformant planning via symbolic model checking and heuristic search. *Artificial Intelligence*, **159**, 127–206. Also available as ITC-irst technical report T03-12-24.

GHALLAB M., NAU D. & TRAVERSO P. (2004). *Automated Planning – Theory and Practice*. Morgan Kaufmann.

HALBWACHS N. (1993). *Synchronous Programming of Reactive Systems*. Kluwer.

JOBSTMANN B., GALLER S., WEIGLHOFER M. & BLOEM R. (2007). Anzu : A tool for property synthesis. In *Computer Aided Verification (CAV)*, p. 258–262.

MALER O., PNUELI A. & SIFAKIS J. (1995). On the synthesis of discrete controllers for timed systems. In *STACS 95, LNCS 900*, p. 229–242 : Springer Verlag.

PNUELI A. & ROSNER R. (1989). On the Synthesis of a Reactive Module. In *Conference Record of the 16th ACM Symp. Principles of Programming Languages*, p. 179–190.

RAMADGE P. J. & WONHAM W. M. (1989). The control of discrete event systems. *Proceedings of the IEEE*, **77**(1), 81–98.

F. ROSSI, P. VAN BECK & T. WALSH, Eds. (2006). *Handbook of Constraint Programming*. Foundations of Artificial Intelligence. Elsevier.

THE CHOCO TEAM (2008). *choco : an Open Source Java Constraint Programming Library*. Rapport interne, École des Mines de Nantes, France. http ://www.emn.fr/z-info/choco-solver/pdf/choco-presentation.pdf.

VAN HENTENRYCK P. (2000). *ILOG OPL, Optimization Programming Language, Reference Manual*. ILOG.