Aix-Marseille Université

École doctorale de mathématiques et informatique de Marseille (ED 184)

Thèse

présentée en vue d'obtenir le grade de

Docteur d'Aix-Marseille Université

spécialité : Informatique

par

Janusz Malinowski

sous la direction de Peter Niebert

Algorithmes pour la Synthèse et le Model Checking

Thèse soutenue le 10 décembre 2012 devant le jury composé de :

Oded Maler	Université Joseph Fourier	rapporteur
Eugene Asarin	Université Paris Diderot-Paris 7	rapporteur
Nicolas Markey	ENS Cachan	examinateur
Antoine Girard	Université Joseph Fourier	examinateur
Rémi Morin	Aix-Marseille Université	examinateur
Peter Niebert	Aix-Marseille Université	directeur de thèse

Remerciements

Je tiens à remercier Oded Maler, Eugene Asarin, Nicolas Markey, Antoine Girard et Rémi Morin d'avoir accepté de faire partie de mon jury et plus particulièrement Oded Marler et Eugene Asarin, les rapporteurs de ma thèse, qui ont relu mon manuscrit.

Je n'aurai pas de mot assez fort pour remercier mon directeur de thèse, Peter Niebert, qui a été un véritable guide dans cette épreuve difficile. Il a su faire preuve d'une patience infinie en supportant les aléas de ma vie privée, mes insomnies, mes hésitations, nos prises de bec. Ses idées se sont souvent révélées justes et pertinentes alors que je les remettais parfois en cause. Il n'a jamais cessé de me montrer la voie à suivre, d'être toujours présent. Il a été bien plus qu'un directeur de thèse pour moi. Merci Peter.

Cette thèse a été réalisée au Laboratoire d'Informatique Fondamentale, dans lequel j'ai pu trouver des conditions idéales pour travailler. Je remercie toute l'équipe du LIF et plus particulièrement Denis Lugiez, qui a cru en moi et m'a accordé sa confiance aux moments clés de mon parcours. J'espère avoir été (un peu) à la hauteur de ses attentes.

J'ai une pensée toute particulière pour Edward et Georgette Nabiel, amis depuis toujours, qui m'ont vu grandir et n'ont cessé d'exprimer leur amitié et leur soutien tout au long de ma vie.

Je remercie ma famille et plus particulièrement ma mère, qui a toujours été présente, dans les bons et les mauvais moments de ma vie, m'apportant un soutien sans condition. Cette thèse, je te la dédie.

Comment ne pas remercier celle qui m'a supporté pendant toutes ces années, pendant que je passais mes examens, qui s'est occupée de notre fille pour que je ne sois pas dérangé... Salima, merci pour tout.

J'embrasse mes filles, Saori et Hilda, témoins malgré elles de toute cette aventure commencée il y a bien des années.

Résumé

Cette thèse a pour objet l'étude d'algorithmes permettant l'exploration de systèmes de grande taille tant dans le domaine de la synthèse de contrôleur que de la vérification. Nous avons étudié une approche discrète de la synthèse de contrôleurs pour les systèmes hybrides permettant la manipulation de dynamiques non-linéaires. Dans cette approche, les états sont regroupés dans une partition finie au prix d'une sur-approximation non déterministe de la relation de transition. Nous avons développé des algorithmes basés sur une approche hiérarchique du problème de la synthèse en le résolvant pour des sous problèmes et en utilisant ces résultats pour réduire l'espace d'états du problème global. Nous avons aussi combiné des objectifs de contrôle de sécurité et de vivacité pour s'approcher d'une stabilisation. Nous avons complété notre étude pour les cas où les objectifs évoluent avec le temps. Des résultats implémentés sur un prototype viennent montrer l'intérêt de cette approche. Pour la vérification, nous avons étudié le problème du model checking d'automates temporisés basé sur la résolution SAT. Nous avons exploré des solutions alternatives pour le codage des réductions SAT basées sur des exécutions parallèles de transitions indépendantes et temporisées. Nous avons développé trois sémantiques différentes pour les séquences temporisées avec des transitions parallèles, dont nous prouvons la correction. Des résultats expérimentaux viennent valider notre approche.

Mots-clefs : Synthèse de contrôleur, Systèmes dynamiques, Systèmes EDO, Stabilisation, Hiérarchie, Automates temporisés, Multistep.

Abstract

The main goal of this thesis is the study of algorithms for big size system exploration in the field of controler synthesis or verification. We consider a discretization based approach to controller synthesis of hybrid systems that allows to handle non-linear dynamics. In such an approach, states are grouped together in a finite index partition at the price of a non-deterministic over approximation of the transition relation. We propose a hierarchical approach to the synthesis problem by solving it first for sub problems and using the results for state space reduction in the full problem. A secondary contribution concerns combined safety and liveness control objectives that approximate stabilization. We studied also the case where goals evolve with the time. Results implemented on a prototype show the benefit of this approach. For the verification, we study the model checking problem of timed automata based on SAT solving. Our work investigates alternative possibilities for coding the SAT reductions that are based on parallel executions of independent timed transitions. We define and analyse three different semantics of timed sequences with parallel transitions. We prove the correctness of the proposed semantics and report experimental results with a prototype implementation.

Keywords : Controler synthesis, Dynamic systems, ODE systems, Stabilization, Hierarchy, Timed automata, Multistep.

Table des matières

Introduction

Cette thèse présente les travaux que j'ai réalisés durant mon doctorat sous la direction de Peter Niebert. La première partie de mes travaux concerne la synthèse de contrôleur avec pour base les jeux [CDF+05]. Une publication [MNR11] ainsi que le développement d'un prototype accompagnent cette partie. La seconde partie concerne la vérification ou le Model Checking [BCC99] : il s'agit du prolongement de mon mémoire de Master Recherche et a aussi donné lieu à une publication [MN10] ainsi que l'extension de la plate-forme de prototypage POEM.

Dans cette introduction, nous allons présenter en détail ces deux parties afin de donner au lecteur le fil conducteur de cet ouvrage et de mieux en apprécier le contenu.

Synthèse de contrôleur

Etant donné un *système* (un objet, une machine) pouvant avoir des interactions avec son environnement (l'air, l'eau, des perturbations). Ce système est dynamique [Lyg04] si à chaque instant il possède un *état*, c'est à dire un ensemble de valeurs constitué des variables du système. Si l'on considère un *objectif* à atteindre pour ce système (par exemple maintenir l'altitude d'un avion), alors un *contrôleur* transmettra à ce dernier l'action à réaliser en fonction de son état courant et de l'objectif, voir Figure 1.

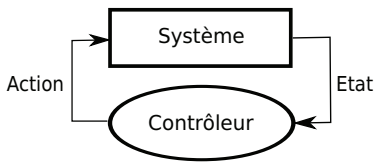


FIGURE 1 – Un système et son contrôleur

Observer un système dynamique revient à observer l'évolution de son état au

fil du temps. Un système dynamique peut être continu, discret ou bien hybride selon que son état prenne ses valeurs dans $\mathbb{R}^n$, dans un ensemble énumérable $\{q_1, q_2, ...\}$ ou alors un mélange des deux (certaines variables de l'état prennent leurs valeurs dans $\mathbb{R}^n$ et d'autres dans un ensemble énumérable $\{q_1, q_2, ...\}$). L'évolution du temps peut aussi être continue, discrète ou hybride selon que le temps évolue sur $A \subset \mathbb{R}$, sur $B \subset \mathbb{N}$ ou bien évolue de manière continue mais possède des instants discrets. Les systèmes à états continus peuvent être linéaires ou non-linéaires. Dans le premier cas l'évolution du système est régie par un système d'équations différentielles linéaires, et dans le second par un système d'équations non-linéaires. Les systèmes que nous avons étudiés peuvent être considérés comme continus et non-linéaires bien que les algorithmes utilisés manipulent des variables continues sous forme discrète. De même le temps évolue de manière continue mais est manipulé de façon discrète (voir la Section 2.4 consacrée à la discrétisation).

Le premier d'entre eux, le *pendule inversé* (voir figure 2) est un système continu assez simple à décrire avec un état constitué de quatre variables. Toutefois ce n'est pas un système trivial car sa dynamique est non-linéaire ce qui nous a permis de tester les différents algorithmes des sections suivantes. Un autre modèle étudié est un quadricoptère ou drone développé par la société Novadem (Figure 2) présentant des caractéristiques continues non-linéaires avec l'avantage de pouvoir valider la partie théorique sur un objet concret.

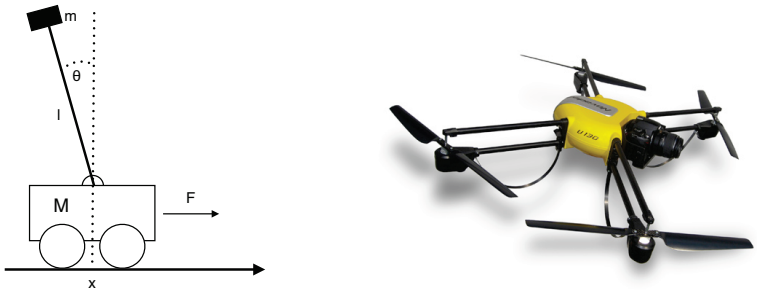


FIGURE 2 – Les modèles étudiés : le pendule inversé et le drone U130 développé par la société Novadem

Synthétiser un contrôleur peut se ramener à calculer des jeux à alternance (voir [GTW02]) : deux joueurs s'affrontent, l'un représentant le contrôleur l'autre un environnement hostile. Chacun joue à tour de rôle. On dit que le contrôleur possède une *stratégie gagnante* si quelques soient les actions de l'environnement, le contrôleur possède une suite d'actions pour atteindre l'objectif. On dit qu'un système est contrôlable si le contrôleur possède une stratégie gagnante. Les notions de jeux les plus répandues dans les applications de contrôle sont les jeux de sécurité (safety) (par exemple les jeux de Ramadge-Wonham) où le contrôleur est supposé éviter que quelque chose de « mauvais » se produise (en interdisant certaines transitions contrôlables) ainsi que les jeux d'accessiblité (reachability) où le contrôleur est supposé amener le système dans un bon état

en un temps fini (voir par exemple [GTW02], [AT02]).

Contribution. Nos deux principales contributions sont d'une part la formalisation et l'écriture d'algorithmes de *stabilisation* des systèmes contrôlés en se basant sur des jeux : on souhaite que le système *atteigne* un objectif et y *reste*. La discrétisation d'un système continu, c'est à dire la transformation de ses variables réelles en variables prenant leurs valeurs sur des ensembles finis, a pour conséquence une explosion du nombre d'états à explorer (par exemple plusieurs milliards pour un système simple comme le pendule inversé).

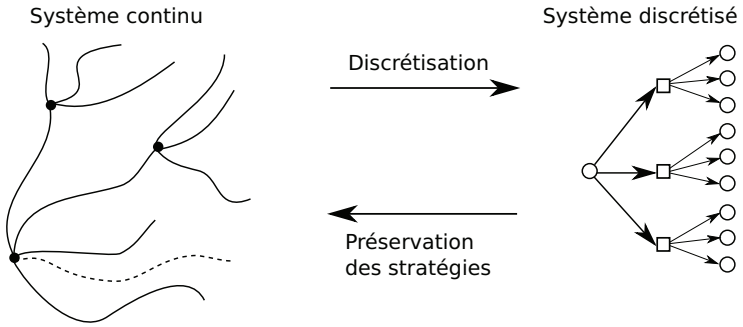


FIGURE 3 – Discrétisation d'un système continu

Notre seconde contribution consiste à utiliser une approche *hiérarchique* du système : le système est décomposé en sous-systèmes indépendants, comportant chacun moins de variables que le système complet. Les résultats des sous systèmes sont ensuite utilisés pour calculer le système complet en éliminant un certain nombre d'états. Il est possible d'utiliser cette méthode sur plusieurs niveaux. Plus précisément, on effectue une projection des états perdants des sous systèmes d'un niveau inférieur au niveau supérieur. Cette décomposition est rendue possible grâce à des particularités dans les systèmes d'équations différentielles régissant les systèmes dynamiques. Cette approche permet de diminuer significativement le nombre d'états à explorer.

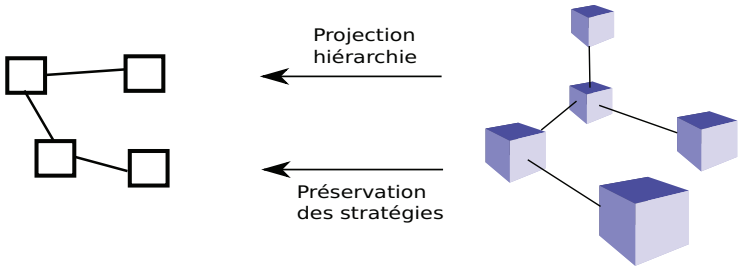


FIGURE 4 – Principe de hiérarchie

Plan. Nous allons dans un premier temps expliquer comment synthétiser un contrôleur en utilisant des jeux au Chapitre 1. Nous aborderons ensuite la notion de contrôleur stabilisant au Chapitre 2, c'est à dire des contrôleurs amenant le système vers un objectif et y demeurant pour toujours. Nous détaillerons l'algorithme implémenté à la Section 2.2. Cet algorithme ne traite que les états discrets à actions discrètes. Or les systèmes que nous traitons sont des systèmes dynamiques régis par des systèmes d'équations différentielles (EDO). Après une formalisation des systèmes EDO et des trajectoires à la Section 2.3, nous montrerons l'équivalence entre l'existence d'une stratégie pour des jeux discrets (donc d'un contrôleur) et l'existence d'une trajectoire d'un système continu à l'aide d'une simulation à la Section 2.3. Au Chapitre 3 nous discuterons de l'approche hiérarchique utilisée pour réduire l'espace d'états à explorer. Nous définirons différents « outils » comme la *bisimulation asymétrique* permettant d'établir une équivalence entre deux systèmes, la *projection* qui permet d'utiliser les résultats d'un sous système S_1 vers un système S_2 tel que $S_1 \prec S_2$ et nous montrerons que les deux systèmes sont équivalents. Nous donnerons des résultats expérimentaux obtenus à l'aide du principe de hiérarchie. Au Chapitre 4 nous donnons des détails sur l'implémentation des algorithmes précédents dans un prototype et nous précisons le concept de perturbations permettant d'obtenir des contrôleurs pour des systèmes plus généraux. Au Chapitre 5 nous aborderons la notion d'objectifs dynamiques, c'est à dire des objectifs évoluant avec le temps, et nous verrons comment les algorithmes précédents ont pu être utilisés pour parvenir à synthétiser des contrôleurs avec de tels objectifs.

Model checking

De nos jours la *vérification formelle* occupe une place prépondérante dans la vérification des systèmes informatiques critiques et en particulier, des systèmes embarqués. Alors que les tests et les simulations ne permettent d'explorer que *certains* comportements d'un système, laissant ainsi les exécutions non testées sans garantie sur l'absence de bugs aux conséquences imprévisibles, la vérification formelle permet de réaliser une *exploration exhaustive* de tous les comportements possibles.

Il existe plusieurs approches formelles comme par exemple la logique de Hoare [Hoa69], les assistants de preuve ou le model checking que nous allons étudier dans ce document.

Le model checking présente l'avantage d'être totalement automatique, ne nécessitant pas de paramétrage particulier ni d'expertise en mathématique ou en logique et est une technique très puissante pour l'étude de systèmes concurrents. De plus si l'analyse ne peut satisfaire une propriété donnée du système, un contre exemple est alors généré permettant de mieux comprendre les raisons de l'échec. Dans le model checking, un modèle est écrit dans un langage de haut niveau, et une propriété est saisie (par exemple, est ce qu'il existe une situation telle que le programme va se bloquer ?). Le model checker va ensuite transformer ce modèle en une représentation adaptée à l'informatique, typiquement un

automate, dans lequel chaque état représente un emplacement dans le modèle et chaque transition, une action du modèle se réalisant selon certaines conditions (par exemple avec un if-then-else). Par transformation, le model checking revient à vérifier si l'on peut atteindre des états d'un automate ayant une certaine propriété. Le problème est que le nombre d'états peut rapidement devenir ingérable, en particulier avec les systèmes concurrents pouvant induire de nombreux entrelacements.

La nécessité de créer des modèles ayant des contraintes temporelles est rapidement apparue entrainant l'utilisation d'*automates temporisés* [Alu98], ajoutant une complexité supplémentaire. Les automates temporisés sont des automates auxquels sont ajoutés des variables à valeurs réelles : *les horloges*. Ces horloges permettent de mesurer la progression du temps, et d'empêcher (ou de forcer) l'occurrence d'une transition avant un laps de temps. Les horloges ayant leurs valeurs dans $\mathbb{R}^+$, différentes approches ont été développées afin de pouvoir en étudier le comportement comme par exemple l'utilisation de zones [Alu92], de régions [Alu98]...

Dans notre travail nous nous sommes intéressés à deux méthodes ayant fait leurs preuves pour les systèmes discrets : *les méthodes d'ordre partiel* [Val90, Pel93, LNZ04, LNZ05] et *les réductions SAT* [Hel01, NMA$^+$02, BCC99].

Les méthodes d'ordre partiel consistent à construire un automate réduit, alors que l'automate complet, souvent trop grand pour tenir dans le mémoire, n'est jamais construit. Les comportements de l'automate réduit sont un sous ensemble des comportements de l'automate complet. La justification est que les comportements non présents dans l'automate réduit n'apportent aucune information complémentaire. Les méthodes d'ordre partiel se basent sur la *relation de dépendance* existante entre les transitions d'un système, et elles décident quelles transitions doivent être présentes dans l'automate réduit, et quelles transitions ne doivent pas l'être.

Les réductions SAT consistent à transformer le problème de l'accessibilité d'un état dans un automate en un problème de satisfaisabilité d'une formule SAT : les états, les transitions, les variables, les opérations, ainsi que la propriété recherchée sont alors exprimés à l'aide de littéraux (variables booléennes) et de clauses représentant une formule CNF. La satisfaisabilité de cette formule nous permet de déduire l'existence d'un chemin de longueur bornée vers un état ayant les propriétés recherchées.Bien que le problème de la satisfaisabilité d'une formule SAT soit un problème NP-Complet [Coo71], certaines heuristiques telles que l'algorithme de Davis-Putnam permettent en pratique de résoudre ce problème. De plus la montée en puissance des ordinateurs, ainsi que l'amélioration des SAT solvers, rendent l'utilisation de cette technique intéressante.

De nombreuses méthodes permettent l'analyse complète d'un système, explorant des chemins non bornés, mais atteignent rapidement leur limite face à l'explosion du nombre d'états, rendant l'analyse impossible dès que les modèles dépassent une certaine dimension. Dans [BCC99] est introduit le *Bounded Model Checking*, dont le principe est de chercher l'existence de chemins témoins de tailles bornées. La principale difficulté consiste alors à bien définir la borne k, sous peine de ne pas pouvoir vérifier une propriété alors qu'elle est peut être

vérifiable pour une borne $k+1$. Les réductions SAT sont une mise en application du Bounded Model Checking.

Contribution. Notre contribution a consisté à appliquer des méthodes d'ordre partiel aux réductions SAT, en permettant l'exécution parallèle (concurrente) de plusieurs transitions. Bien que cela ait déjà été réalisé pour les systèmes discrets, en particulier dans [Hel01], *cela n'a jamais été étudié* pour les systèmes temporisés. L'objectif est de compresser les chemins solutions ou bien de permettre l'analyse de modèles de plus grande taille. Nous avons par la suite réalisé une implémentation de nos différents algorithmes de gestion du temps afin de vérifier leur pertinence.

Plan. Cette partie est structurée de la manière suivante : au Chapitre 6 nous introduisons les notions de base des systèmes temporisés avec un niveau de spécification : les programmes multithreadés. Il est essentiel d'utiliser un tel modèle afin de comprendre le codage SAT. Nous allons aussi introduire des notions issues des automates temporisés : les *horloges*, les conditions sur les horloges et leur remise à zéro. Au Chapitre 7, nous rappelons les notions d'indépendance dans le contexte des automates temporisés et introduisons les sémantiques des multisteps. Les principaux outils formels sont développés ici, différentes notions de progrès du temps sont définies formellement et leur équivalence est démontrée. Au Chapitre 8, nous montrons comment intégrer ces concepts dans une réduction SAT pour les systèmes décrits au Chapitre 6. Cette description permet de comprendre comment une telle réduction est construite et comment les notions de la Section 6.2 sont intégrées dans les instances SAT. Au Chapitre 9, nous allons aborder la façon dont les *invariants d'état*, un concept important dans les automates temporisés ont été intégrés pour chacun des progrès du temps. Au chapitre 10.2, nous illustrons le potentiel de ces algorithmes à l'aide de résultats expérimentaux obtenus avec notre prototype implémenté.

Première partie

Synthèse de contrôleur pour systèmes hybrides

CHAPITRE 1

Jeux pour la synthèse

1.1 Présentation

Dans l'introduction, nous avons évoqué une correspondance entre la synthèse de contrôleur et les jeux à deux joueurs (voir [Gim06]). Nous allons dans un premier temps examiner un exemple de jeu à l'aide du graphe de la Figure 1.1.

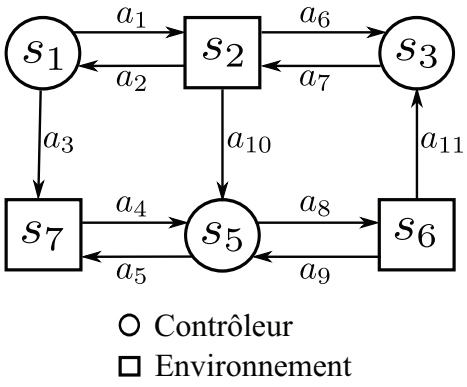


FIGURE 1.1 – Graphe d'un jeu

Dans ce graphe on peut observer un ensemble d'états $\{s1, s2, ..., s7\}$ reliés entre eux par des arêtes $\{a_1, a_2, ..., a_{11}\}$ aussi appelées *actions*. Pour plus de clarté dans les figures, nous omettrons par la suite le nom des arêtes. Chaque état appartient à un des joueurs : les états ronds au Contrôleur et les états carrés à l'Environnement. Le but est de se déplacer d'état en état en suivant

les arêtes. Le jeu démarre dans un état donné, par exemple l'état $s1$: il s'agit d'un état rond, appartenant au Contrôleur, c'est à son tour de jouer. Il dispose de deux possibilités : soit de se rendre dans l'état $s2$ en empruntant l'arête a_1 (nous noterons ce choix $s1 \xrightarrow{a_1} s2$) soit dans l'état $s7$ en empruntant l'arête a_3 (choix noté $s1 \xrightarrow{a_3} s_7$) (voir Figure 1.2) [1].

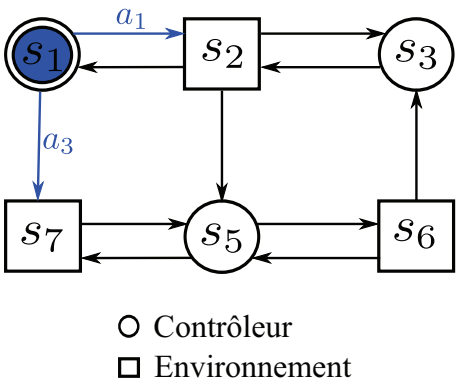


FIGURE 1.2 – Possibilités du Contrôleur depuis l'état $s1$

Si le Contrôleur choisit l'état $s2$, c'est au tour de l'Environnement de jouer qui dispose de trois possibilités : $s2 \xrightarrow{a_2} s1$, $s2 \xrightarrow{a_6} s3$ et $s2 \xrightarrow{a_{10}} s5$ (voir Figure 1.3) et ainsi de suite.

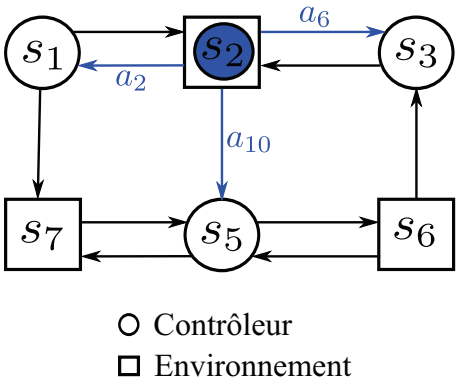


FIGURE 1.3 – Possibilités de l'Environnement depuis l'état $s2$

Admettons que l'*objectif de victoire* pour le Contrôleur est d'atteindre l'état

1. Il peut sembler inutile de nommer l'arête empruntée pour aller d'un état s vers un état s' car il n'y a à priori qu'une seule arête menant d'un état vers un autre. Nous rencontrerons par la suite des graphes dans lesquels il existe plusieurs arêtes entre deux états, et donc il sera indispensable de les nommer explicitement.

$s3$ en partant de $s1$ (voir Figure 1.4) quelques soient les choix effectués par l'environnement au cours du jeu. Si le Contrôleur gagne alors l'Environnement perdra, et donc le but de ce dernier est d'empêcher le Contrôleur de gagner. Examinons le jeu suivant : le Contrôleur choisit $s1 \xrightarrow{a_1} s2$, l'Environnement choisira par exemple $s2 \xrightarrow{a_{10}} s5$, puis le Contrôleur jouera par exemple $s5 \xrightarrow{a_8} s6$ et ensuite $s6 \xrightarrow{a_9} s5$. Le Contrôleur n'a aucun moyen d'arriver à coup sûr à l'état $s3$.

Prenons à présent comme objectif l'état $s5$ (Figure 1.5). Cette fois ci le Contrôleur peut forcer la victoire grâce à la suite de coups $s1 \xrightarrow{a_3} s7 \xrightarrow{a_4} s5$. On dit que le Contrôleur possède une *stratégie gagnante*.

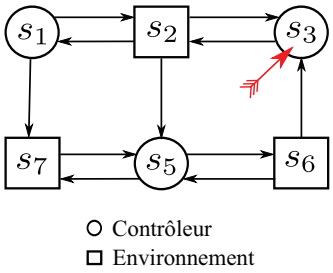


FIGURE 1.4 – Premier objectif pour le Contrôleur

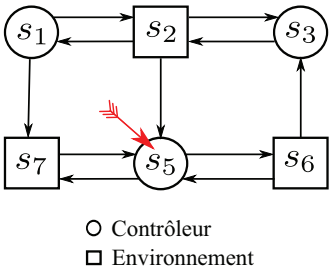


FIGURE 1.5 – Second objectif pour le Contrôleur

1.2 Formalisation

Soit Σ_C et Σ_E les ensembles d'actions réalisables (les ensembles de choix pouvant être effectués) par respectivement le contrôleur et l'environnement. Ces deux ensembles sont considérés comme fixés.

1.2.1 Structure de jeu

Définition 1. Control Game Structure (CGS). *Un* CGS *sur* (Σ_E, Σ_C) *est un tuple* $\mathcal{C} = \langle S_E, S_C, T, S_0 \rangle$ *tel que :*

- S_C *est l'ensemble des états du contrôleur*
- S_E *est l'ensemble des états de l'environnement*
- $S := S_C \cup S_E$ *et* $S_C \cap S_E = \varnothing$
- T *est une relation de transition* $T \subseteq (S_E \times \Sigma_E \times S_C) \cup (S_C \times \Sigma_C \times S_E)$
- S_0 *est un ensemble d'états initiaux avec* $S_0 \neq \varnothing$

La relation de transition T permet de relier dans un graphe deux états s et s' selon une arête étiquetée par l'action a comme précédemment : $s \xrightarrow{a} s'$. Nous ajoutons des restrictions à la relation T : la première est qu'il existe au moins une action pour chaque état de l'environnement. Plus formellement $\forall s \in S_E, \exists a \in \Sigma_E, \exists s' \in \Sigma_C | (s, a, s') \in T$. La seconde est que les actions du contrôleur soit déterministes, c'est à dire $\forall s \in S_C, \forall a \in \Sigma_C, \exists! s' \in \Sigma_E | (s, a, s') \in T$.

Soit $succ(s)$ l'ensemble des successeurs de l'état $s \in S$ définit par :
- Si $s \in S_C$ alors $succ(s) = \{s' | \exists a \in \Sigma_C \wedge s' \in S_E \wedge (s, a, s') \in T\}$
- Si $s \in S_E$ alors $succ(s) = \{s' | \exists a \in \Sigma_E \wedge s' \in S_C \wedge (s, a, s') \in T\}$

Définition 2. *Une* partie *de* $\mathcal{C} = \langle S_E, S_C, T, S_0 \rangle$ *est une séquence finie ou infinie* $w = (s_0, a_1, s_1, a_2, s_2, ...)$ *telle que :*
- $\forall i \geq 0, s_i \in S$
- $\forall i \geq 0, a_i \in (\Sigma_C \cup \Sigma_E)$
- $\forall i \geq 0, (s_i, a_{i+1}, s_{i+1}) \in T$

On notera qu'il n'est pas nécessaire que $s_0 \in S_0$, c'est à dire il n'est pas nécessaire que la séquence commence avec un état initial car nous nous focalisons sur les jeux sans mémoire (voir [GTW02]).

1.2.2 Stratégies et victoire

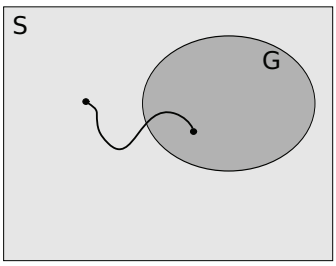


FIGURE 1.6 – Objectif W_{Reach} : Atteindre G

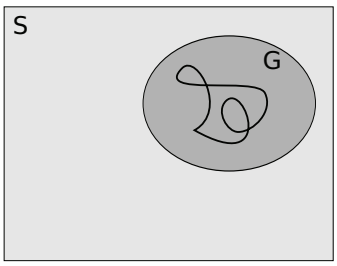


FIGURE 1.7 – Objectif W_{Safe} : Rester dans G

Définition 3. *Un objectif de* victoire *ou objectif de contrôle pour le contrôleur est un ensemble* W_C *de parties. Les objectifs de victoire les plus courants sont :*
- $W_{Reach}(\mathsf{G}) = \{(s_0, a_1, s_1...) | \exists i : s_i \in \mathsf{G}\}$ *: l'ensemble* G *est atteint éventuellement (Figure 1.6)*
- $W_{Safe}(\mathsf{G}) = \{(s_0, a_1, s_1...) | \forall i : s_i \in \mathsf{G}\}$ *: on évite toujours l'ensemble* $S \setminus \mathsf{G}$ *(Figure 1.7)*

Dans un jeu à deux joueurs, chacun choisit l'état successeur de l'état courant en utilisant une *stratégie* :

Définition 4. *Une stratégie pour un contrôleur est une fonction* $\lambda : S \times S_C \rightarrow S$ *telle que* $\lambda(s_1 s_2 ... s_n) \in succ(s_n)$ $\forall s_1, ..., s_{n-1} \in S \wedge s_n \in S_C$

Une stratégie λ est dite *gagnante* pour un contrôleur et pour un objectif de victoire W si pour toute séquence $\rho = (s_0, a_1, s_1, ...)$ dans laquelle les actions $a_k \in \Sigma_C$ ont été déterminées en fonction de λ alors $\rho \in W$. Un état s est gagnant pour un objectif de contrôle W s'il existe une stratégie gagnante à partir de s. Enfin un CGS $\mathcal{C}$ est gagnant pour un objectif de contrôle W si et seulement si tous les états $s \in S_0$ sont gagnants pour l'objectif de contrôle W.

1.3 Calcul des jeux

Afin de résoudre un jeu, donc de générer un contrôleur pour un système, nous allons introduire les opérateurs $\exists CPre$, $\forall CPre$ et $CPre$.

Définition 5. *Soit $X \subseteq S$, alors :*
- $\exists CPre(X) = \{s \in S_C | \exists s' \in succ(s) : s' \in X\}$
- $\forall CPre(X) = \{s \in S_E | \forall s' \in succ(s) : s' \in X\}$

La figure 1.8 illustre intuitivement cette définition : l'opérateur $\exists CPre(X)$ représente l'ensemble des états à partir desquels le contrôleur possède un choix lui permettant d'atteindre X en un coup tandis que l'opérateur $\forall CPre(X)$ représente l'ensemble des états à partir desquels tous les choix de l'environnement l'obligent à aller dans X.

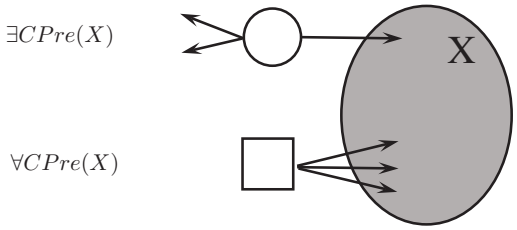


FIGURE 1.8 – Operateurs $\exists Pre$ et $\forall Pre$

Nous pouvons maintenant définir l'opérateur $CPre(X)$:

Définition 6. *Soit $X \subseteq S$, alors : $CPre(X) = (\exists Pre(X) \cap S_C) \cup (\forall Pre(X) \cap S_E)$*

$CPre(X)$ représente l'ensemble des états à partir desquels il est possible de forcer le prochain coup, que ce soit pour le contrôleur ou l'environnement, afin d'arriver dans X.

1.3.1 Jeux étendus

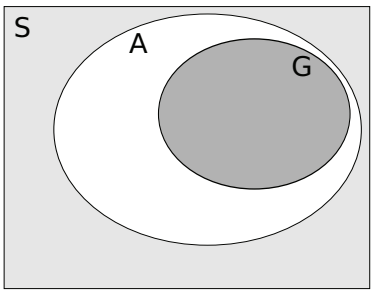


FIGURE 1.9 – Jeu étendu

Aux figures 1.6 et 1.7 nous avons illustrés des objectifs avec l'ensemble $S = S_E \cup S_C$ et un ensemble d'états G contenant les états objectifs tels que $\mathsf{G} \subseteq S$. Cette formulation sous entend qu'il n'existe pas d'états considérés comme *mauvais*, c'est à dire des états que l'on souhaite prendre en considération mais que l'on ne souhaite pas visiter (dans le cas d'un avion il pourrait s'agir des états dans lesquels la vitesse est trop élevée entraînant une disloquation de l'aéronef). Nous allons étendre l'aire de jeu précédente en rajoutant un ensemble d'états A contenant tous les états autorisés tel que $\mathsf{G} \subseteq \mathsf{A} \subseteq \mathsf{S}$, voir figure 1.9. Ainsi les états appartenant à $\mathsf{S} \setminus \mathsf{A}$ seront considérés comme mauvais, interdits.

Les définitions énoncées précédemment demeurent vraies pour notre nouvelle aire de jeu. Nous allons introduire deux nouveaux objectifs W_{Stay} et W_{Until} qui sont les pendants pour ce nouveau modèle des objectifs W_{Safe} et W_{Reach}.

1.3.2 Objectif W_{Stay}

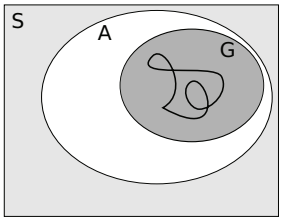


FIGURE 1.10 – Objectif W_{Stay}

L'objectif W_{Stay} (Figure 1.10) est similaire à l'objectif W_{Safe} vu précédemment : rester indéfiniment dans un ensemble, quelques soient les actions de l'environnement. Plus formellement :

$$W_{Stay}(\mathsf{G}) = \{\rho = (s_0, a_1, s_1, a_2, ...) | \forall i \geq 0, s_i \in \mathsf{G}\}$$

Un exemple d'objectif W_{Stay} est la stabilisation d'un avion en vol : on souhaite qu'il reste indéfiniment dans une plage correcte de vitesses et d'altitudes. Tous les états de G ne permettent pas de garantir un objectif $W_{Stay}(\mathsf{G})$: examinons le graphe de jeu de la Figure 1.11.

Dans cette figure $\mathsf{S} = \{s1, ..., s8\}$ et $\mathsf{G} = \{s1, ..., s7\} = S \setminus s8$. On constate qu'à partir des états $s4$ et $s7$ appartenant à G, l'environnement peut sortir de G en allant vers $s8$. La méthode permettant de calculer l'ensemble des états permettant de garantir l'objectif $W_{Stay}(\mathsf{G})$ est itérative : soit X_i l'ensemble d'états à partir duquel le contrôleur peut forcer le prochaine position à rester dans G après i coups joués :

#	Ensemble	Etats
0	$X_0 = \mathsf{G}$	$s1, s2, s3, s4, s5, s6, s7$
1	$X_1 = \mathsf{G} \cap CPre(X_0)$	$s1, s2, s3, s5, s6$
2	$X_2 = \mathsf{G} \cap CPre(X_1) = X_1$	$s1, s2, s3, s5, s6$

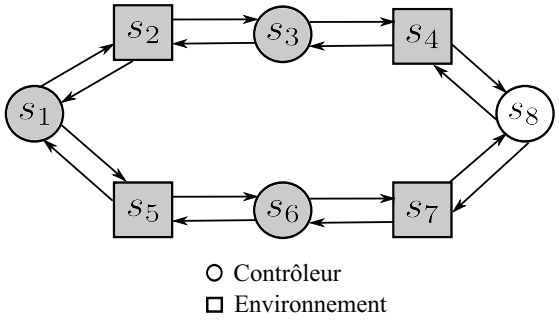


FIGURE 1.11 – Un nouveau jeu avec les états objectifs en gris

$X_0 = \mathsf{G}$ représente l'ensemble des états qui nous permettent de rester dans G sans jouer de coup. L'étape X_1 représente donc l'ensemble des états à partir desquels on a la garantie d'atteindre G en un seul coup c'est à dire l'ensemble $\{s1, s2, s3, s5, s6\}$. On constate que X_2, qui contient les états permettant d'atteindre G en deux coups (et donc d'atteindre X_1 en un seul coup) contient les mêmes éléments que X_1 soit $X_2 = X_1$. On en déduit alors que $\forall n \geq 2.X_n = X_{n-1}$: on a atteint un *point fixe*, l'ensemble n'évoluant plus. Il s'agit ici du *plus grand point fixe*.

Définition 7. *Soit l'opérateur* $O_{Stay(\mathsf{G})}(X) = \mathsf{G} \cap CPre(X)$ *alors on caractérise l'ensemble des états permettant d'assurer un objectif* $W_{Stay}(\mathsf{G})$ *pour un CGS* $\mathcal{C}$ *par*

$$Stay(\mathcal{C}, \mathsf{G}) = \bigcap_{n \geq 0} O^n_{Stay(\mathsf{G})}(S)$$

Dans cette définition $O^0(X) = X$ et $O^{n+1}(X) = O(O^n(X))$. Pour les lecteurs familiers avec le $\mu -$ calcul, on peut écrire cette définition sous la forme :

$$Stay(\mathcal{C}, \mathsf{G}) = \nu X.\mathsf{G} \wedge CPre(X)$$

On notera que pour un état $s \in Stay(\mathcal{C}, \mathsf{G}) \cap S_C$, le contrôleur doit choisir une action $a \in \Sigma_C$ telle que $s' \in Stay(\mathcal{C}, \mathsf{G})$ et $s \xrightarrow{a} s'$.

1.3.3 Objectif W_{Until}

L'objectif W_{Until} (Figure 1.12) présente une différence par rapport à W_{Reach} qui est l'existence d'états *interdits*. Partant d'un état de A il faut trouver un chemin jusqu'à un état de G en évitant les mauvais états de $\mathsf{S} \setminus \mathsf{A}$. Formellement :

$$W_{Until}(\mathsf{A}, \mathsf{G}) = \{\rho = (s_0, a_1, s_1, a_2, ...) | \exists k. \forall i, 0 \leq i < k, s_i \in \mathsf{A} \wedge s_k \in \mathsf{G}\}$$

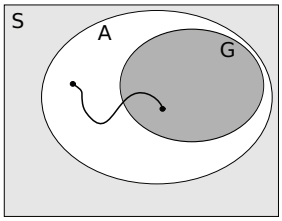


FIGURE 1.12 – Objectif W_{Until}

Un exemple d'objectif W_{Until} pour un véhicule serait d'atteindre une vitesse donnée. Examinons la Figure 1.13, nous avons $\mathsf{S} = \{s1, ..., s9\}, \mathsf{A} = \{s1, s2, s3, s4, s5, s6, s9\}$ et $\mathsf{G} = \{s9\}$. Les états à éviter sont donc $\mathsf{S} \setminus \mathsf{A} = \{s7, s8\}$. De même que pour W_{Stay}, le calcul de W_{Until} est itératif. Soit X_i l'ensemble des états à partir duquel le contrôleur peut forcer à atteindre G en *au plus* i coups. Il vient alors :

#	Ensemble	Etats
0	$X_0 = \mathsf{G}$	$s9$
1	$X_1 = \mathsf{G} \cup CPre(X_0)$	$s5, s9$
2	$X_2 = \mathsf{G} \cup CPre(X_1)$	$s4, s5, s9$
3	$X_3 = \mathsf{G} \cup CPre(X_2)$	$s2, s3, s4, s5, s9$
4	$X_4 = \mathsf{G} \cup CPre(X_3)$	$s1, s2, s3, s4, s5, s9$
5	$X_5 = \mathsf{G} \cup CPre(X_4) = X_4$	$s1, s2, s3, s4, s5, s9$

L'ensemble $X_0 = \{s9\}$ contient les états à atteindre. Concernant X_1, qui représente les états pouvant atteindre G en un seul coup, on constate que les candidats sont $s5, s6, s8$. Or $s8$ est un état interdit et depuis $s6$ le contrôleur peut décider d'aller dans l'état interdit $s7$. Il ne reste donc que $s5$ à ajouter. En continuant jusqu'à X_5, on constate que $X_5 = X_4$, on en déduit alors que $\forall n \geq 5, X_n = X_{n-1}$: on a atteint un *point fixe*, l'ensemble n'évoluant plus. Il s'agit ici du *plus petit point fixe*.

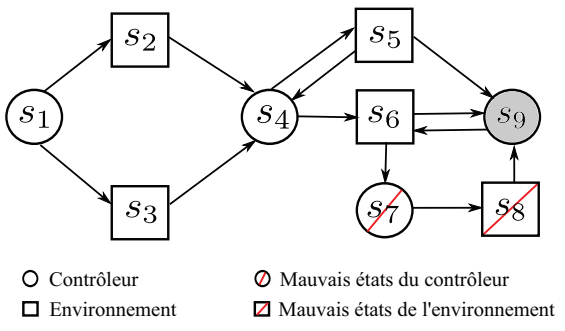


FIGURE 1.13 – Jeu pour objectif W_{Until} avec mauvais états

Définition 8. *Soit l'opérateur* $O_{Until(\mathsf{A},\mathsf{G})}(X) = \mathsf{G} \cup (\mathsf{A} \cap CPre(X))$ *alors on caractérise l'ensemble des états permettant d'assurer un objectif* $W_{Until}(\mathsf{A},\mathsf{G})$ *pour un CGS* $\mathcal{C}$ *par*

$$Until(\mathcal{C},\mathsf{A},\mathsf{G}) = \bigcup_{n\geq 0} O^n_{Until(\mathsf{A},\mathsf{G})}(\varnothing)$$

Dans cette définition $O^0(X) = X$ et $O^{n+1}(X) = O(O^n(X))$. Pour les lecteurs familiers avec le $\mu-$ calcul, on peut écrire cette définition sous la forme :

$$Until(\mathcal{C},\mathsf{A},\mathsf{G}) = \mu X.\mathsf{G} \vee (\mathsf{A} \wedge CPre(X))$$

Définition 9. *Soit* $s \in Until(\mathcal{C},\mathsf{A},\mathsf{G})$*, on appelle distance* $d(s,\mathsf{G})$ *le nombre minimal de coups pour atteindre* G *à partir de* s*. Formellement,* $d(s,\mathsf{G})$ *est le plus petit* n *tel que* $s \in O^{n+1}_{Until(\mathsf{A},\mathsf{G})}(\varnothing)$*.*

Afin de garantir un progrés vers G, le contrôleur doit choisir à partir d'un état $s \in S_C$ une action $a \in \Sigma_C$ telle que $(s,a,s') \in T$ vérifiant $d(s',\mathsf{G}) < d(s,\mathsf{G})$. Ceci est garanti par la caractérisation du point fixe de l'ensemble $Until(\mathcal{C},\mathsf{A},\mathsf{G})$.

CHAPITRE 2

Contrôleur stabilisant

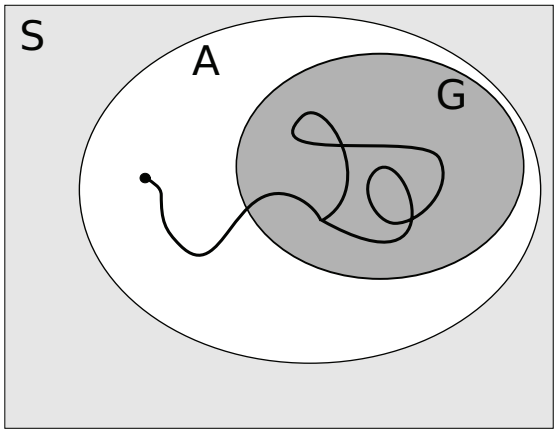


FIGURE 2.1 – Objectif $W_{Stabilize} = W_{Until}$ puis W_{Stay}

2.1 Stabilisation

Nous avons vu dans le chapitre 1 les objectifs de contrôle W_{Until} et W_{Stay} adaptés à un ensemble comportant des états interdits, autorisés et des états objectifs. Une de nos contributions dans cette partie dédiée à la synthèse de contrôleur a été de combiner ces deux objectifs de contrôle : progresser à travers un ensemble d'états autorisés en évitant des états interdits vers un ensemble d'états objectifs afin d'y rester (voir Figure 2.1). Afin de restreindre l'espace

d'états explorés et d'obtenir des algorithmes efficaces, les calculs seront démarrés depuis un des états initiaux de S_0. Formellement :

$$W_{Stabilize}(\mathsf{A},\mathsf{G}) = \{\rho = (s_0, a_1, s_1, a_2, ...) | \exists k, \forall i, 0 \leq i < k, s_i \in \mathsf{A} \wedge \forall j \geq k, s_j \in \mathsf{G} \wedge s_o \in S_0\}$$

Pour définir les états garantissant une stratégie $W_{Stabilize}$, nous allons introduire le sous ensemble de A des états accessibles depuis une condition initiale :

$$Acc(\mathcal{C},\mathsf{A}) = \{s \in S | \exists \rho = (s_0, a_1, ..., a_{n-1}, s_n) \text{ tel que } s_0 \in S_0 \wedge s_n = s \wedge \forall i \leq n, s_i \in \mathsf{A}\}$$

A présent nous pouvons formaliser l'ensemble des états permettant de garantir une stratégie $W_{Stabilize}$ à l'aide des opérateurs $Until$, $Allow$ et Acc :

$$Stabilize(\mathcal{C},\mathsf{A},\mathsf{G}) = Until(\mathcal{C}, Acc(\mathcal{C},\mathsf{A}), Stay(\mathcal{C}, \mathsf{G} \cap Acc(\mathcal{C},\mathsf{A})))$$

Intuitivement $Stabilize$ représente l'ensemble des états permettant de progresser dans $Acc(\mathcal{C},\mathsf{A})$ vers $\mathsf{G} \cap Acc(\mathcal{C},\mathsf{A})$ pour y rester (rappelons que $\mathsf{G} \subseteq \mathsf{A}$). La stratégie pour $Stabilize$ est sans mémoire, il s'agit bien d'une simple combinaison des objectifs de contrôle W_{Until} et W_{Stay}.

2.2 Algorithme pour contrôleurs stabilisants

Nous allons dans cette section présenter l'algorithme utilisé pour résoudre le problème de la synthèse de contrôleur. Cet algorithme est dérivé des algorithmes de model checking pour le $\mu-$ calcul sans alternance, utilisant des concepts publiés dans [CS93, LS98].

Dans [CDF+05], les auteurs observent que ces algorithmes locaux peuvent être utilisés pour décider des stratégies d'accessibilité dans des jeux temporisés ou non. Nous étendons ces spécifications dans notre cas plus complexe, car $Stabilize$ peut s'exprimer comme une formule de $\mu-calcul$ sans alternance. Il s'agit d'un algorithme *local* car comme nous le verrons l'exploration du système de transitions débute depuis des états initiaux et l'exploration d'états perdants est arrêtée.

Etant donné un CGS $\mathcal{C}$ et deux ensembles Allow et Goal tels que Goal $\subseteq$ Allow, l'algorithme se compose de trois phases :

- Exploration *en avant* du graphe
- Calcul de l'ensemble Stay par *propagation arrière*
- Calcul de l'ensemble Until (et donc de Stabilize) par *propagation arrière*

2.2.1 Exploration du graphe

La première partie de l'algorithme démarre par les états initiaux $s \in S_0$. Pour chaque transition $(s, a, s') \in T$, si s' n'est pas dans Goal, alors on le marque *Not in Goal* (Ligne 5), c'est à dire que cet état ne pourra pas faire partie de Stay quoiqu'il arrive. Cette information de non appartenance à Goal sera utilisée dans les parties suivantes de l'algorithme pour effectuer la *propagation arrière*. Si $s' \in$ Allow et si cet état n'a pas déjà été exploré alors on l'ajoute à la liste

Algorithme 1 : Algorithme pour l'exploration en avant

Data : $\mathcal{C} = \langle S_E, S_C, T, S_0, \lambda \rangle$,Allow,Goal
Result : NotInGoal, Passed
1 $Waiting \leftarrow S_0$; NotInGoal $\leftarrow \emptyset$; $Passed \leftarrow \emptyset$;
2 **while** $Waiting \neq \emptyset$ **do**
3 $s \leftarrow pop(Waiting)$; $Passed \leftarrow Passed \cup \{s\}$;
4 **foreach** $(s, a, s') \in T$ **do**
5 **if** $s' \notin$ Goal **then** NotInGoal $\leftarrow$ NotInGoal $\cup \{s'\}$;
6 $depend[s'] \leftarrow depend[s'] \cup \{s\}$;
7 **if** $s' \in$ Allow $\wedge\, s' \notin Passed$ **then** $Waiting \leftarrow Waiting \cup \{s'\}$;
8 **end**
9 **end**

des états à explorer (Ligne 7). Afin d'économiser du temps de calcul pour les parties suivantes, chaque prédécesseur d'un état s' est mémorisé (Ligne 6). On recommence cette partie jusqu'à ce qu'il n'y ait plus d'états à explorer.

2.2.2 Calcul de Stay

Algorithme 2 : Calcul de Stay

Data : $\mathcal{C} = \langle S_E, S_C, T, S_0, \lambda \rangle$,NotInGoal, Passed
Result : STAY, Passed
1 $\forall s \in Passed \cap S_C; counter_C(s) \leftarrow \#Succ(s)$;
2 $Waiting \leftarrow$ NotInGoal ; $STAY \leftarrow Passed$;
3 **while** $Waiting \neq \emptyset$ **do**
4 $s \leftarrow pop(Waiting)$;
5 $STAY \leftarrow STAY \setminus \{s\}$;
6 **if** $s \in S_C$ **then**
7 **foreach** $s' \in depend[s]$ **do**
8 **if** $s' \in STAY$ **then** $Waiting \leftarrow Waiting \cup \{s'\}$;
9 **end**
10 **else**
11 **foreach** $s' \in depend[s]$ **do**
12 $counter_C(s') \leftarrow counter_C(s') - 1$;
13 **if** $counter_C(s') = 0$ **then** $Waiting \leftarrow Waiting \cup \{s'\}$;
14 **end**
15 **end**
16 **end**

La seconde partie de l'algorithme a pour tâche de calculer l'ensemble des états de Stay et donc de calculer un *plus grand point fixe* (voir Section 1.3.2).

Initialement l'ensemble Stay contient tous les états visités par l'algorithme 1. Chaque état s marqué précédemment *Not in Goal* sera logiquement enlevé de Stay (ligne 5). Ensuite une propagation arrière est effectuée : si s est un état du contrôleur alors tous ses prédécesseurs (des états de l'environnement) doivent aussi être enlevés de Stay et une nouvelle propagation aura lieu à partir de ces derniers (lignes 6 à 9). Si s est un état de l'environnement alors la propagation arrière s'effectuera sur ses prédécesseurs (des états du contrôleur) uniquement si ces derniers n'ont plus d'autre choix que d'aller vers un état de l'environnement ne faisant pas partie de Stay. L'utilisation de $counter_c$ permet de comptabiliser les choix restant pour chaque état du contrôleur afin d'accélérer les calculs (lignes 11 à 15).

Si l'on se place d'un point de vue d'une analyse en avant plutôt qu'en arrière, nous dirions dans le premier cas que si un état de l'environnement a une possibilité de se rendre dans un état du contrôleur ne pouvant pas être dans Stay, alors l'environnement fera ce choix. Il faut donc considérer cet état de l'environnement comme ne faisant pas partie de Stay. Dans le second cas, tant qu'un état du contrôleur a une possibilité de se rendre vers un état de l'environnement encore dans Stay, alors le contrôleur choisira cette possibilité. Comme indiqué précédemment, il sera enlevé de Stay s'il ne possède plus cette liberté.

2.2.3 Calcul de Until

Algorithme 3 : Calcul de Until

Data : $\mathcal{C} = \langle S_E, S_C, T, S_0, \lambda \rangle$, STAY
Result : UNTIL, STAY

```
1  Waiting ← STAY ; UNTIL ← ∅ ;
2  while Waiting ≠ ∅ do
3      s ← pop(Waiting);
4      UNTIL ← UNTIL ∪ {s};
5      if s ∈ S_C then
6          foreach s' ∈ depend[s] do
7              counter_E(s') ← counter_E(s') − 1;
8              if counter_E(s') = 0 then Waiting ← Waiting ∪ {s'};
9          end
10     else
11         foreach s' ∈ depend[s] do
12             if s' ∉ UNTIL then Waiting ← Waiting ∪ {s'};
13         end
14     end
15 end
```

La troisième partie de l'algorithme va calculer l'ensemble Until et par conséquent l'ensemble Stabilize en calculant un plus petit point fixe (voir Section 1.3.3). Initialement l'ensemble Until est vide, et l'on va traiter les états encore dans Stay à l'étape précédente à l'aide encore une fois d'une propaga-

tion en arrière. Alors que dans l'algorithme 2 nous propagions des « mauvaises nouvelles » (l'impossibilité pour un état de demeurer dans Stay), nous allons à présent propager des « bonnes nouvelles » : l'ajout d'un état à l'ensemble Until. Le fonctionnement est en quelque sorte dual de l'algorithme précédent. Chaque état s de Stay est ajouté à Until. Si s est un état du contrôleur, alors la propagation est effectuée aux prédécesseurs s' (des états de l'environnement) si et seulement si tous les choix de ces derniers sont des états du contrôleur appartenant à Until. On utilise comme précédemment un compteur $counter_E$ permettant de comptabiliser, pour un état de l'environnement, le nombre de choix menant vers des états Until (lignes 6 à 10). Dans le cas où s est un état de l'environnement, alors on effectue une propagation arrière aux prédécesseurs s' (des états du contrôleur), à moins que ces derniers ne fassent déjà partie de Until (lignes 11 à 15).

2.3 Systèmes d'équations différentielles

Un système dynamique est représenté de manière statique par son état $x : \mathbb{R}^+ \to \mathscr{S}$ dont la valeur évolue au cours du temps $t \in \mathbb{R}^+$ et où $\mathscr{S}$ est l'espace d'état du système tel que $\mathscr{S} \subseteq \mathbb{R}^n$. On notera ainsi $x(t_i)$ la valeur de l'état au temps t_i. Nous avons dans ce travail restreint notre étude à des valeurs du temps positives.

Pour modéliser le fait que des facteurs externes peuvent l'influencer, nous utiliserons des paramètres en entrée $u \in U$ où U est un ensemble fini de valeurs de $\mathbb{R}$.

Les systèmes dynamiques évoluent continûment dans le temps et le modèle mathématique utilisé pour les représenter est celui des *équations différentielles ordinaires* (ou EDO) de la forme :

$$\dot{x} = f(t, x, u) \tag{2.1}$$

avec $f : \mathbb{R}^+ \times \mathscr{S} \times U \to \mathbb{R}^n$.

Bien que cette formulation soit concise, elle peut sembler inhabituelle car manipulant des espaces multidimentionnels sans que cela n'apparaisse immédiatement. Développons l'équation 2.1 en posant

$$x = \begin{bmatrix} x_1 \\ ... \\ x_n \end{bmatrix}$$

avec $\forall 1 \leq i \leq n, x_i : \mathbb{R}^+ \to \mathbb{R}$. Et de même avec la fonction f :

$$f(t, x, u) = \begin{bmatrix} f_1(t, x, u) \\ ... \\ f_n(t, x, u) \end{bmatrix}$$

avec $\forall 1 \leq i \leq n, f_i : \mathbb{R}^+ \times \mathbb{R}^n \times U \to \mathbb{R}$. On obtient alors un système EDO sous une forme plus « familière » :

$$\begin{cases} \dot{x_1} &= f_1(t, x_1, ..., x_n, u) \\ ... & \\ \dot{x_n} &= f_n(t, x_1, ..., x_n, u) \end{cases} \tag{2.2}$$

Les systèmes EDO formulés précédemment possèdent une infinité de solutions possibles. Afin de restreindre l'espace des solutions, nous allons ajouter une condition initiale (t_0, x_0, u_0). Toutefois, l'existence de cette condition ne garantit pas l'unicité de la solution. Dans la suite de l'exposé, nous admettrons que la fonction f est continue par partie et localement Lipshitz :

Définition 10. *La fonction $f : \mathbb{R}^+ \times \mathscr{S} \times U \to \mathbb{R}^n$ est localement Lipschitz par rapport à x au point x_0 s'il existe un voisinage $\mathcal{N}(x_0, r) = \{x \in \mathbb{R}^n | \parallel x - x_0 \parallel < r\}$ tel que $\forall (t, x_1, u), (t, x_2, u)$*

$$\parallel f(t, x_1, u) - f(t, x_2, u) \parallel \leq k \parallel x_1 - x_2 \parallel$$

Lemme 1. *Soit $f(t, x, u)$ une fonction continue en t et localement Lipschitz en x au point x_0 pour tout $t \in [t_0, t_1]$. Alors il existe $\delta > 0$ tel que l'EDO $\dot{x} = f(t, x, u)$, avec la condition initiale (t_0, x_0, u_0) telle que $x(t_0) = x_0$, a une unique solution sur $[t_0, t_0 + \delta]$.*

Ce lemme nous garantie l'existence d'une unique solution pour une condition initiale donnée. Nous pouvons à présent introduire la notion de trajectoire.

Définition 11 (Trajectoire)**.** *On appelle trajectoire du système 2.1 tout fonction $I \to \mathbb{R}^n$ qui satisfait identiquement sur un intervalle d'intérieur non vide I de $\mathbb{R}$ les équations de 2.1 avec la condition initiale (t_0, x_0, u_0)*

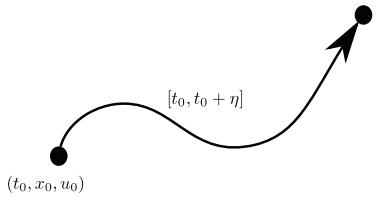


FIGURE 2.2 – Une trajectoire

Définition 12 (Séquence de trajectoires)**.** *Une séquence de trajectoires ζ de conditions initiales $(t_k, x_{k-1}(t_k), u_k)$ pour $0 \leq k \leq N$ est définie par :*

- *$\mathcal{I} = \{I_k | 0 \leq k \leq N\}$ une séquence d'intervalles I_k de la forme $[t_0; t_1], [t_1; t_2], ..., [t_N, t_{N'}]$*
- *$\sigma = \{u_k | 0 \leq k \leq N\}$ une séquence d'éléments $u_k \in U$ (chaque u_k est constant sur chaque intervalle I_k)*
- *$\mathcal{X} = \{x_k | 0 \leq k \leq N\}$ une séquence de fonctions $x_k : I_k \to \mathbb{R}^n$ continues dérivables par parties et solution du système EDO $\mathcal{O}$ de conditions initiales $(t_k, x_{k-1}(t_{k-1}), u_k)$*

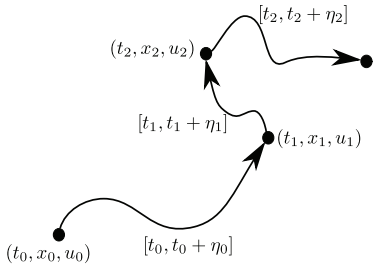


FIGURE 2.3 – Séquence de trajectoires

Exemple 1: *Nous allons illustrer les définitions précédentes à l'aide d'un exemple qui nous servira de fil conducteur tout au long de notre exposé : le pendule inversé (voir Figure 2.4).*

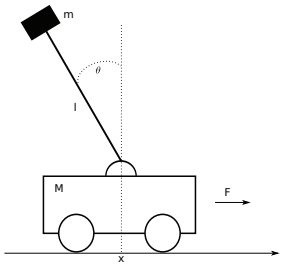


FIGURE 2.4 – Le pendule inversé

Le principe est simple : un véhicule de masse M muni de quatre roues peut se déplacer librement en avant et en arrière. Sa position x est repérée sur un axe horizontale. Une tige de longueur l et de masse négligeable est accrochée sur le dessus de ce véhicule. Cette tige peut tourner librement autour de son point d'attache d'un angle θ. Un poids de masse m est accroché à son sommet. L'objectif est de maintenir la tige aussi verticale que possible (c'est à dire avoir un angle θ aussi petit que possible). Ce système se décrit à l'aide d'un système d'équations différentielles non linéaires :

$$\begin{cases} (M+m)\ddot{x} - ml\ddot{\theta}\cos\theta + ml\dot{\theta}^2\sin\theta = F \\ l\ddot{\theta} - g\sin\theta = \ddot{x}\cos\theta \end{cases} \tag{2.3}$$

Dans ce système $\dot{\theta}$ représente la variation de l'angle par rapport au temps $\frac{d\theta}{dt}$ c'est à dire la vitesse angulaire, $\ddot{\theta}$ représente l'accélération angulaire, $\dot{x}$ la vitesse du véhicule, $\ddot{x}$ son accélération. F représente la force appliquée sur le mobile pour le faire bouger : il s'agit du paramètre d'entrée déterminé par le contrôleur . En posant $x_1 = x$, $x_2 = \dot{x_1}$, $x_3 = \theta$ et $x_4 = \dot{\theta}$, nous obtenons le système de quatre équations différentielles ordinaires suivant :

$$\begin{cases} \dot{x_1} &= x_2 \\ \dot{x_2} &= \frac{F+(l.x_4^2-g.\cos x_3).m.\sin x_3}{M+m.\sin^2 x_3} \\ \dot{x_3} &= x_4 \\ \dot{x_4} &= \frac{g.\sin x_3.M-\cos x_3.F+(g-l.x_4^2.\cos x_3).m.\sin x_3}{l.M+l.m.\sin^2 x_3} \end{cases} \tag{2.4}$$

Trouver une trajectoire de ce système revient, à déterminer une solution $(x_1, \dot{x_1}, \theta_1, \dot{\theta_1})$ *au problème du pendule de condition initiale* $(t_0, [x_0, \dot{x_0}, \theta_0, \dot{\theta_0}], F_0)$.

2.4 Discrétisation

Nous allons dans cette section établir un lien entre les EDO vu à la section 2.3 et un CGS. Ce lien établit, il sera possible d'utiliser les algorithmes décrits à la section 2.2 pour synthétiser un contrôleur d'un système à partir des équations décrivant sa dynamique. Les systèmes EDO décrits dans la section 2.3 sont en principe non linéaires et ne peuvent pas être résolus en général. Nous allons réaliser une approximation du système dynamique par un système à états finis.

Nous allons restreindre le comportement du contrôleur afin d'obtenir un conroleur à temps discret possédant une période $\eta \in \mathbb{Q}^+$. Chaque changement du paramètre d'entrée u ne pourra survenir qu'au temps $k.\eta$ avec $k \in \mathbb{N}$. De fait chaque trajectoire aura une durée de η unités de temps. On appelera η la période d'échantillonnage.

Soit un système EDO $\mathcal{O}$ et une condition initiale (t_0, x_0, u_0). L'espace d'états $\mathscr{S} \in \mathbb{R}^n$ est défini comme un hyper rectangle $I_1 \times ... \times I_n$ de $\mathbb{R}^n$ (chaque I_k correspondant à une dimension de S). Chaque ensemble $I_k \subseteq \mathbb{R}$ pour $1 \leq k \leq n$ est partitionné en un nombre fini d'intervalles comme représenté dans la Figure 2.5. Par définition du partitionnement, si l'on appelle p_i les partitions de l'intervalle I_k alors $\bigcap p_i = \varnothing$ et $\bigcup p_i = I_k$. Nous nommerons $\mathcal{P}_k$ le partitionnement de chaque intervalle I_k. Ce partitionnement permet d'obtenir une abstraction finie d'un système infini.

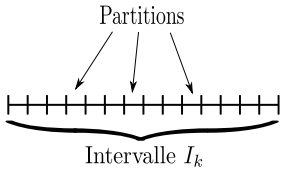


FIGURE 2.5 – Partitionnement des intervalles I_k

En reprenant les défintions de la Section 1.2, nous allons essayer d'obtenir un CGS $\mathcal{C}$ $\langle S_E, S_C, T, S_0 \rangle$ sur les alphabets (Σ_E, Σ_C). Dans cette défintion le contrôleur choisit la valeur du paramètre d'entrée en choisissant une lettre dans

Σ_C. D'un autre côté, l'environnement résout le non-déterminisme induit par le système EDO et il n'est donc pas nécessaire de mettre des labels sur les transitions de l'environnement (en d'autres termes l'ensemble Σ_E pourra être vide ou bien constitué d'une lettre quelconque). Nous obtenons les définitions suivantes :

- $S_C = \prod_{j=1}^{n} \mathcal{P}_j$
- $S_E = S_C \times U$
- $S_0 = (p_0, u_0)$ avec p_0 un intervalle tel que $x_0 \in p_0$
- $\Sigma_C = U$
- $\Sigma_E = \{e\}$, avec e une lettre quelconque

Les transitions du contrôleur sont :

$$T \cap (S_C \times \Sigma_C \times S_E) = \{s \xrightarrow{u} (s, u) | u \in U, s \in S_C\}$$

Concernant les transitions de l'environnement, étant donné une « cellule » (une partition de S) et une valeur d'entrée, nous souhaitons approximer l'ensemble des cellules accessibles après un délai de η unités de temps. Bien que nous supposons que f soit une fonction localement Lipschitz et que pour une condition initiale donnée il n'existe qu'une et une seule solution au système EDO, chaque cellule possède une infinité de points et donc une infinité de trajectoires possibles.

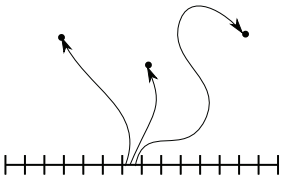


FIGURE 2.6 – Multiples trajectoires possibles depuis une cellule

Le problème du calcul de ces successeurs a déjà été étudié par plusieurs auteurs : méthodes pour intervalles numériques [HHMWT00], techniques mathématiques standards basées sur l'évaluation de la constante de Lipschitz [ADG07], simulation du système basée sur une analyse de sensibilité [DM07]. Il est aussi possible d'exploiter certaines particularités du système EDO et de f : nous donnerons certains détails dans la partie expérimentations.

Définition 13 (Sur-approximation)**.** *Soit un système EDO $\mathcal{O}$ défini par $(f, \mathscr{S}, U)$ et une condition initiale (x_0, t_0, u_0). Le CGS $\mathcal{C} = \langle S_E, S_C, T, S_0 \rangle$ est une sur-approximation de $\mathcal{O}$ muni de sa condition initiale s'il satisfait la propiété suivante : $\forall (s, u) \in S_E, \forall x_0 \in s$ soit $x(t)$ l'unique solution de l'EDO $\mathcal{O}$ ayant pour condition initiale (x_0, t_0, u). Alors pour tout $s \in S_C$ tel que $x(\eta) \in s'$, il vient $(s, u) \xrightarrow{e} s' \in T$.*

Soit Γ un ensemble de propositions atomiques caractérisant les états, c'est à dire : un état est-il dans Stay, dans Goal, est-il hors de Allow, etc. Nous allons

introduire deux applications : la première pour les états discrets, la seconde pour les états réels d'un système associé à un EDO. Soit $\lambda : S_C \to \Gamma^n$ qui a un état discret de S_C associe un ensemble de propositions atomiques ou dit autrement λ renvoie les propositions atomiques que possède un état discret s. Soit $M : \Gamma \to \mathscr{S}$ une application qui associe une proposition atomique à un ensemble d'états réels du système EDO. Posons qu'un état discret s possède une propriété γ si et seulement si tous ses points réels la possèdent (voir Figure 2.7). Plus formellement :

$$\forall \gamma \in \Gamma, \forall s \in S_C, \gamma \in \lambda(s) \Leftrightarrow s \cap M(\gamma) \neq \varnothing \text{ ssi } s \subseteq M(\gamma)$$

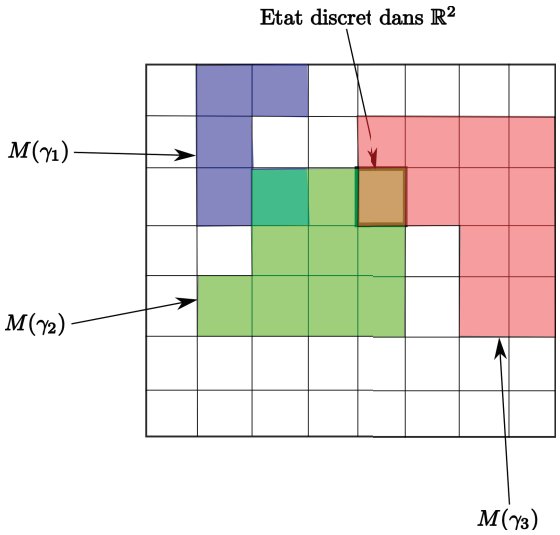


FIGURE 2.7 – Application de M sur des propositions $\gamma_1, \gamma_2, \gamma_3$

Proposition 1 (Simulation)**.** *Soit $\mathcal{O}$ un système EDO de condition initiale (t_0, x_0, u_0) et une CGS $\mathcal{C}$ qui en est une sur-approximation. Alors pour toute séquence de trajectoires ζ de $\mathcal{O}$ telle que chaque intervalle $I \in \mathcal{I}$ est de la forme $[k.\eta, (k+1).\eta]$ avec $k \in \mathbb{N}$ et η une constante représentant la période d'échantillonage, alors il existe un chemin $\rho = (s_0, a_1, s_1, a_2, ...)$ dans $\mathcal{C}$ tel que $\sigma = (a_i)_i$, et $x_i(t_i) \in s_i$ pour tout i.*

La propriété de la simulation entraîne que si l'on peut synthétiser un contrôleur pour une *CGS* $\mathcal{C}$ pour un objectif de contrôle, alors ce contrôleur peut être utilisé comme un contrôleur à temps discret pour un système EDO $\mathcal{O}$ pour le même objectif de contrôle. La seule différence étant que les propositions atomiques sont garanties uniquement à chaque période d'échantillonnage η.

Exemple 2: *Pour reprendre l'exemple du pendule inversé, nous pouvons voir à la Figure 2.8 la « discrétisation » (le partitionnement) adopté pour chaque*

variable (chaque intervalle) du système. L'expérimentation nous a permis de trouver des bornes acceptables pour chacunes d'entre elles. Par exemple, l'angle θ a été divisé en 160 parties pour des valeurs comprises entre $-\frac{\pi}{4}$ et $\frac{\pi}{4}$ radians.

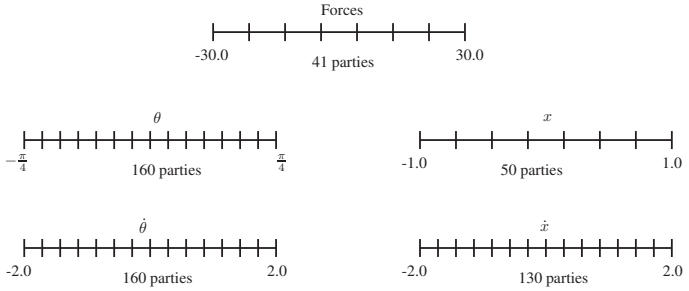


FIGURE 2.8 – Partitionnements utilisés pour le pendule inversé

CHAPITRE 3

Principe de hiérarchie

Dans ce chapitre nous allons présenter une approche originale pour l'analyse de systèmes EDO discrétisés. Comme nous pouvons le voir dans l'exemple 2 du Chapitre 2, le nombre d'états discrets explose avec le nombre de variables. Notre technique exploite les dépendances entre les variables du système afin de résoudre des sous-systèmes de petite taille et d'utiliser ces résultats pour réduire drastiquement la taille de l'espace d'états à explorer avec l'ensemble complet de variables.

3.1 Abstractions et préservation de la contrôlabilité

Nous allons définir une abstraction particulière utilisée par la suite : il s'agit d'une bisimulation en ce qui concerne les transitions possibles mais seulement une simulation pour ce qui est de la statisfaction des propriétés. Ainsi $\mathcal{C}_2$ peut être vu comme une abstraction du système $\mathcal{C}_1$:

Définition 14. *Soit deux CGS* $\mathcal{C}_i = \langle S^i_E, S^i_C, T^i, S^i_0 \rangle$ *et* $S^i = S^i_E \uplus S^i_C$ *pour* $i \in \{1, 2\}$. *Soit une application surjective* $\alpha : S^1 \to S^2$ *et la relation associée* $R \subseteq S^1 \times S^2$ *définie par* s_1 R s_2 *si et seulement si* $\alpha(s_1) = s_2$. *Nous disons que* α *possède une relation de bisimulation asymétrique* R *si et seulement si pour toute paire* (s_1, s_2) *telle que* $s_1 R s_2$ *:*

1. *Soit* $s_1 \in S^1_C \wedge s_2 \in S^2_C$ *ou* $s_1 \in S^1_E \wedge s_2 \in S^2_E$.
2. *Si* $s_1 \in S^1_0$ *alors* $s_2 \in S^2_0$
3. $\forall \gamma \in \Gamma$, *si* $\gamma \in \lambda_1(s_1)$, *alors* $\gamma \in \lambda_2(s_2)$
4. $\forall (s_1, a, s'_1) \in T^1$ *il existe* s'_2 *tel que* $(s_2, a, s'_2) \in T^2$ *et* s'_1 R s'_2

5. $\forall (s_2, a, s_2') \in T^2$ *il existe* s_1' *tel que* $(s_1, a, s_1') \in T^1$ *et* $s_1' \ R \ s_2'$

Cette propriété peut être vue comme une instance particulière des propriétés des bisimulations « zig-zag » (voir [BS06]), avec la particularité que les règles (2) et (3) ne sont pas symétriques.

Les relations « zig-zag » préservent des propriétés exprimées dans le μ-calcul. La proposition suivante permet de garantir que l'ensemble des états « gagnants » du système abstrait est un sur-ensemble des états gagnants du système concret.

Proposition 2. *Soient* $\mathcal{C}_1$ *et* $\mathcal{C}_2$ *deux CGS et* $\alpha : S^1 \to S^2$ *une application produisant une relation de bisimulation asymétrique. Soient* $X_1, X_2 \subseteq S_1$ *des ensembles d'états. On a alors :*

$$\alpha(\mathsf{Stay}(\mathcal{C}_1, X_1)) \subseteq \mathsf{Stay}(\mathcal{C}_2, \alpha(X_1))$$

$$\alpha(\mathsf{Until}(\mathcal{C}_1, X_1, X_2)) \subseteq \mathsf{Until}(\mathcal{C}_2, \alpha(X_1), \alpha(X_2))$$

Démonstration. Nous allons dans un premier temps montrer que

$$\forall X_1 \subseteq S_1, X_2 \subseteq S_2, \alpha(X_1) \subseteq X_2 \Rightarrow \left\{ \begin{array}{l} \alpha(Post_1(X)) \subseteq Post_2(\alpha(X)) \\ \alpha(CPre_1(X)) \subseteq CPre_2(\alpha(X)) \end{array} \right.$$

où $Post_i(X) = \bigcup_{s \in X} succ_i(s)$ et $CPre_i$ représente l'opérateur $CPre$ dans le CGS $\mathcal{C}_i$.

On donne l'argument pour la deuxième ligne. Soit $s_2 \in \alpha(CPre_1(X))$ alors il existe un $s_1 \in CPre_1(X) = (\exists CPre_1(X) \cap S_C^1) \cup (\forall CPre_1(X) \cap S_E^1)$. Supposons que $s_1 \in \exists CPre_1(X) \cap S_C^1$ avec $(s_1, a, s_1') \in T^1$ et $s_1' \in X$. Alors le fait que α engendre une bisimulation asymétrique implique que $x_2 \in S_C^2$ et qu'il existe une transition $(s_2, a, s_2') \in T^2$ avec $s_2' = \alpha(s_1') \in \alpha(X)$. On obtient alors $s_2 \in CPre_2(\alpha(X)$. Les autres cas sont similaires.

Pour montrer que $\alpha(\mathsf{Stay}(\mathcal{C}_1, X_1)) \subseteq \mathsf{Stay}(\mathcal{C}_2, \alpha(X_1))$ (et de même pour $\alpha(\mathsf{Until}(\mathcal{C}_1, X_1, X_2)) \subseteq \mathsf{Until}(\mathcal{C}_2, \alpha(X_1), \alpha(X_2))$), il suffit de faire une preuve inductive sur les approximations de Stay via sa définition en tant que point fixe. □

Nous appliquerons la proposition dans la section suivante pour montrer la correction de certaines abstractions.

3.2 Abstractions hiérarchiques dans les systèmes EDO

Soit un système EDO $\mathcal{O} = (f, \mathscr{S}, U)$ défini de manière similaire au système d'équations 2.2 :

$$\left\{ \begin{array}{rcl} \dot{x_1} & = & f_1(t, x_1, ..., x_n, u) \\ ... & & \\ \dot{x_n} & = & f_n(t, x_1, ..., x_n, u) \end{array} \right.$$

où pour chaque $1 \leq i \leq n$, $f_i : \mathbb{R}^+ \times \mathscr{S} \times U \to \mathbb{R}$ est supposé localement Lipschitz.

Définition 15 (Dépendance)**.** *Soient $i, j \in \{1, ..., n\}$. on dit que l'application f_i ne dépend pas de x_j si et seulement si $f_i(t, x_1, ..., x_j, ..., x_n, u) = f_i(t, x_1, ..., x'_j, ..., x_n, u)$ avec $x_j \neq x'_j$. Sinon l'application f_i est dépendante de x_j.*

Dans les systèmes EDO standards pour lesquels les applications sont décrites avec des expressions explicites telles que des polynômes, des sinus, ... , les applications f_i ne dépendent pas de la variable x_j si elle n'apparait pas dans l'expression.

Exemple 3: *En examinant l'équation 2.4 du pendule inversé, on peut observer que le calcul de $\dot{x_4}$ dépend uniquement de x_3 et x_4, les autres variables n'apparaissant pas. De même le calcul de $\dot{x_3}$ dépend de x_3 et de x_4 et donc on en déduit que x_3 et x_4 dépendent l'un de l'autre, mais pas des autres variables. De même on constate que x_2 dépend de x_3 et x_4. Enfin x_1 dépend de x_2. On a ainsi établi la hiérarchie des dépendances dans le cadre du pendule inversé (voir Figure 3.1).*

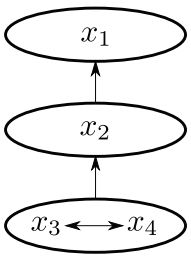


FIGURE 3.1 – Dépendances entre les variables du pendule inversé

Définition 16 (Ensembles indépendants de variables)**.** *Soit $J \subset \{1, ..., n\}$. On dit que le sous ensemble de variables J est indépendant si le sous système obtenu en se restraignant aux variables $\{x_j | j \in J\}$ constitue un sous système indépendant c'est à dire si pour $j \in J$, l'application f_j dépend seulement des variables de l'ensemble $\{x_j | j \in J\}$.*

Comme on peut le constater dans l'exemple précédent, le pendule inversé possède quatre sous ensembles indépendants : $\varnothing$, $\{x_3, x_4\}$, $\{x_2, x_3, x_4\}$, $\{x_1, x_2, x_3, x_4\}$.

Proposition 3. *Les ensembles indépendants de variables d'un système EDO forment un treillis complet.*

Définition 17. *Soit $\mathcal{O}$ un système EDO. On note $\mathcal{L}(\mathcal{O})$ le treillis complet de ses sous ensembles indépendants de variables. De plus soit $J, J' \in \mathcal{L}(\mathcal{O})$ on notera $J' \prec J$ si et seulement si $J' \subsetneq J$ et s'il n'existe pas d'ensemble $J'' \in \mathcal{L}(\mathcal{O})$ tel que $J' \subsetneq J''$ et $J'' \subsetneq J$.*

Nous allons à présent aborder la notion de projection π qui transforme un espace d'états de n dimensions en un espace d'états possédant k dimensions avec $k < n$. A la Figure 3.2 nous pouvons voir un espace d'états à 3 dimensions, c'est à dire que chaque état est décrit à l'aide de trois variables (x_1, x_2, x_3). A la suite de la projection π, chaque état perd une composante et se décrit donc à l'aide de deux variables (x_1, x_2) et donc plusieurs vecteurs auront la même projection, diminuant ainsi la taille de l'espace d'états.

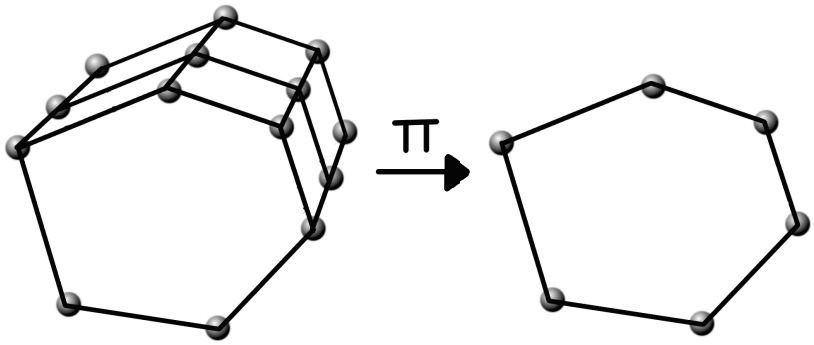


FIGURE 3.2 – Projection de 3 dimensions vers 2 dimensions

Il ne reste plus qu'à appliquer les algorithmes de la section 2.2 afin d'éliminer un certain nombre d'états. En effectuant la projection inverse π^{-1} des états de dimension k vers les états de dimension n avec $k < n$, un état (x, y) éliminé en dimension k éliminera de l'analyse tous les états en dimension n de la forme (x, y, z) avec $z \in \mathbb{R}$ (voir Figure 3.3).

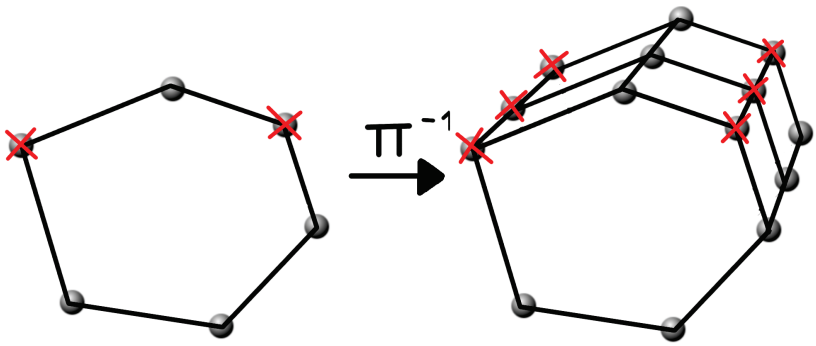


FIGURE 3.3 – Projection inverse et réutilisation des résultats des dimensions inférieures

Nous allons à présent définir formellement la notion de projection :

Définition 18. *Soit un système EDO $\mathcal{O}$ de condition initiale (t_0, x_0, u_0) et un ensemble de partitions $\mathcal{P}_j$ pour $1 \leq j \leq n$. Pour tout ensemble $J \in \mathcal{L}(\mathcal{O})$, on note $\mathcal{C}_J$ la discrétisation du sous système $\mathcal{O}$ restreint à J pour le partitionnement $\mathcal{P}_j$ avec $j \in J$. Soit $J, J' \in \mathcal{L}(\mathcal{O})$ tels que $J' \subseteq J$. On note $\pi_{J,J'}$ la projection des états de $\mathcal{C}_J$ vers les états de $\mathcal{C}'_J$ obtenue en effaçant les composants de J n'apparaissant pas dans J'. On écrira simplement π_j la projection $\pi_{\{1,\ldots,n\},J}$.*

Afin de pouvoir garantir la correction du principe de hiérarchie, nous allons introduire un lemme montrant l'équivalence entre deux systèmes l'un obtenu par la projection π à partir de l'autre :

Lemme 2. *Soit $J, J' \in \mathcal{L}(\mathcal{O})$ tels que $J' \subseteq J$. L'application $\pi_{J,J'}$ est une relation de bisimulation asymétrique entre $\mathcal{C}_J$ et $\mathcal{C}'_J$.*

Démonstration. Soit R la relation associée à $\pi_{J,J'}$. Cette relation suit aisément les points 1 à 4 de la Définition 14 si l'on considére la description de π. Le point 5 est aussi vérifié car J et J' sont des ensembles de variables indépendants. Ceci implique que toute séquence de trajectoires $(\mathcal{I}', \sigma', \mathcal{X}')$ du système ODE $\mathcal{O}$ restreinte à J' peut être étendue en une séquence de trajectoires $(\mathcal{I}, \sigma, \mathcal{X})$ de $\mathcal{O}$ restreinte à J pour laquelle la projection sur J' coïncide avec $(\mathcal{I}', \sigma', \mathcal{X}')$. □

Exemple 4: *Dans le cadre du pendule inversé, la hiérarchie des dépendances est celle de la Figure 3.1 : on commence par résoudre le problème en ne tenant compte que de l'angle et de la vitesse angulaire de la tige, sans tenir compte de la vitesse ni de la position du véhicule. Au cours du calcul, il apparait que certaines positions de la tige sont acceptables (en particulier celles proches de la verticale) et d'autres non (par exemple un angle de $\frac{\pi}{2}$ n'est sûrement pas souhaitable ou bien sort des bornes fixées, voir Figure 3.4).*

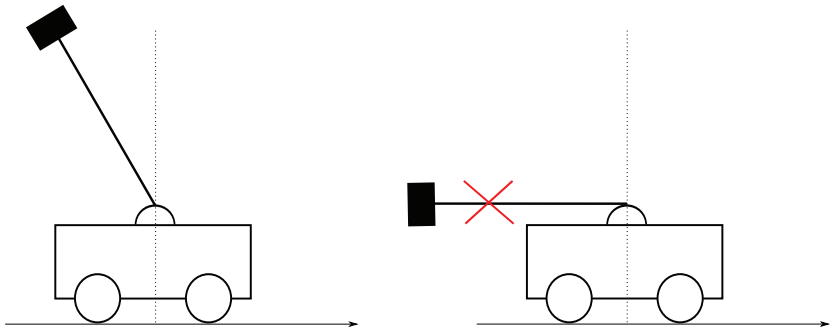


FIGURE 3.4 – Une position du pendule valide et une autre invalide

L'étape suivante consiste à reprendre les calculs en ajoutant la vitesse du véhicule ce qui donne des états à 3 dimensions (x_2, x_3, x_4). Si (x_3, x_4) est un état éliminé lors du calcul précedent en dimension 2, alors tous les états $\pi^{-1}(x_3, x_4)$

seront eux aussi éliminés en dimension 3. On poursuit le calcul en ajoutant la position x_1 du véhicule donnant ainsi des états à 4 dimensions (x_1, x_2, x_3, x_4)

3.3 Algorithme hiérarchique

Nous allons présenter dans cette section l'algorithme hiérarchique utilisant les algorithmes de synthèse des sections précédentes ainsi que le principe de hiérarchie.

Algorithme 4 : Algorithme hiérarchique pour des objectifs de stabilisation

Data : un EDO $\mathcal{O}$, une condition initiale (t_0, x_0, u_0), Allow, Goal
Result : Stabilize($\mathcal{C}_J$), Allow, Goal

1 Calculer le treillis $\mathcal{L}(\mathcal{O})$;
2 **foreach** $J \in \mathcal{L}(\mathcal{O})$ *triés par taille croissante* **do**
3 $\quad \mathsf{A} \leftarrow \pi_J(\mathsf{Allow}) \cap \bigcap_{J' \prec J} \pi^{-1}_{J,J'}(U(J'))$;
4 $\quad \mathsf{G} \leftarrow \pi_J(\mathsf{Goal}) \cap \bigcap_{J' \prec J} \pi^{-1}_{J,J'}(S(J'))$;
5 $\quad (U(J), S(J)) \leftarrow$ Stabilize$(C_J, \mathsf{A}, \mathsf{G})$;
6 **end**
7 **return** $U(\{1, ..., n\})$;

Le fonctionnement de l'algorithme consiste à réduire la taille des ensembles Allow et Goal donnés en paramètre à chaque niveau de hiérarchie, jusqu'au niveau final qui utilisera toutes les variables mais aura éliminé un nombre non négligeable d'états. Plus concrètement, on calcule les ensembles Stay et Until pour chaque sous problème indépendant par ordre croissant du nombre de variables et on exploite la propriété de bisimulation asymétrique pour éliminer des états de ces deux ensembles s'ils n'appartiennent pas à Stay ou à Until avec la projection π.

3.4 Résultats expérimentaux

Nous allons décrire dans cette section les résultats concernant le pendule inversé (ceux concernant le quadricoptère seront détaillés ultèrieurement).

Nous avons réalisé des tests avec plusieurs états initiaux. A la Figure 3.7, on peut voir un état initial avec le pendule proche de la verticale, et le second avec le pendule totalement à l'envers : l'objectif dans ce dernier cas est de réaliser un « swingup », c'est à dire emmener le pendule en position verticale, orienté vers le haut. Les contraintes dans ce second cas sont bien évidemment plus importantes que dans le premier.

Différentes discrétisations ont aussi été expérimentées en sachant que plus il y a d'états discrets, plus les calculs seront précis mais en contrepartie le temps et l'espace nécessaires risquent de rapidement exploser. Une des discrétisations mise en oeuvre est celle de la Figure 2.8 pour un état initial « pas trop éloigné » de la verticale. Un état est dans Goal si :

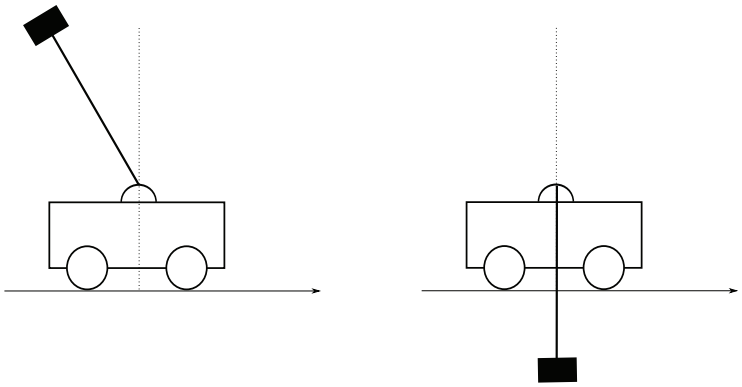


FIGURE 3.5 – Deux états initiaux

	$\{x_3, x_4\}$	$\{x_2, x_3, x_4\}$	$\{x_1, x_2, x_3, x_4\}$
$\lvert S_E \cap \mathsf{Allow}\rvert$	1.049.600	136.448.000	6.822.400.000
$\lvert S_C \cap \mathsf{Allow}\rvert$	25.600	3.328.000	166.400.000
partie explorée de $S_C \cap \mathsf{Allow}$	17.616	815.643	18.261.684
ratio exploration	68%	24%	11%
exploré sans hiérarchie	17.616	1.571.127	n/a
gain avec l'approche hiérarchique	0%	48%	n/a
$\lvert S_C \cap \mathsf{Stay}\rvert$	1.683	50.787	1.305.059
$\lvert S_C \cap \mathsf{Until}\rvert$	10.121	432.547	9.678.467

FIGURE 3.6 – Résultats pour la synthèse d'un contrôleur pour un pendule proche de la verticale en utilisant la discrétisation de la Figure 2.8

- $\theta, \dot{\theta}, x, \dot{x}$ sont dans Allow, c'est à dire si ces 4 variables sont bien contenues entre leurs bornes
- $-0.1 \leq \theta \leq 0.1$, les angles étant donnés en radians

Dans le tableau de la Figure 3.6, chaque colonne représente un niveau de la hiérarchie avec les variables utilisées. Ainsi dans le première colonne, seules x_3 et x_4 ont été utilisées (c'est à dire θ et $\dot{\theta}$), et ainsi de suite pour les autres colonnes. Le ratio d'exploration représente le pourcentage d'états explorés par rapport à la totalité de l'espace d'états. Tous les états ne sont pas explorés car un certain nombre d'entre eux sont éliminés lors des propagations arrières, d'autres le sont en utilisant les résultats obtenus aux niveaux précédents grâce à la projection π. La première colonne ne pouvant pas utiliser les résultats d'un niveau précédent, il n'y aura aucun gain avec l'approche hiérarchique et le nombre d'états explorés en utilisant ou non l'algorithme hiérarchique sera le même. Dés la seconde colonne, on constate que l'approche hiérarchique permet d'éliminer près de la moitié des états à explorer et à la troisième colonne, il n'a pas été possible de fournir de statistiques pour les calculs sans hiérarchie car

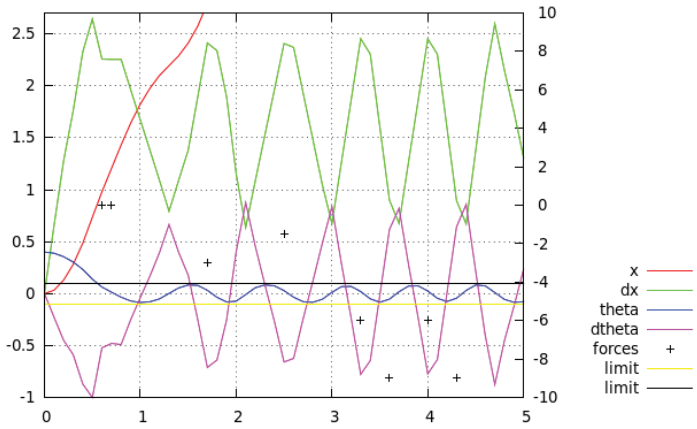


FIGURE 3.7 – Stratégie avec deux variables (x_3, x_4) soit $(\theta, \dot{\theta})$

les ressources nécessaires en mémoire et en temps dépassaient nos capacités de calcul.

CHAPITRE 4

Implémentation

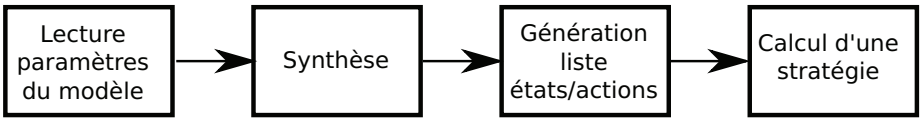


FIGURE 4.1 – Etapes de la synthèse

Nous avons implémenté les algorithmes précédents sous la forme d'un prototype écrit en langage C optimisé afin d'offrir les meilleurs performances possibles en vitesse et en consommation mémoire (bien que ce dernier point reste perfectible). Comme le montre la Figure 4.1, plusieurs étapes s'enchainent avant de produire un contrôleur.

4.1 Lecture du modèle

Le prototype lit la description du modèle sous forme d'un fichier texte comportant les éléments suivants :

- Les *variables* du système : bornes, discrétisation, valeur intial, pertubation et niveau de hiérarchie
- Les *entrées* du système : variables non contrôlables correspondant à l'environnement. Lors des calculs de la synthèse, ces variables ne peuvent sortir de leurs bornes. Elle incluent les mêmes élèments que les variables du système et sont utilisées en particulier pour les systèmes à objectifs dynamiques (voir Chapitre 5).
- Les *actions* du système : bornes et discrétisation

– Du *code* spécifique et en particulier une fonction `isgoal` prenant en entrée un état et renvoyant `vrai` si cet état est dans G, et une fonction `differential` contenant les relations entre les différentes variables (le système EDO décrivant le modèle) et qui sera appelée par le solveur d'équation différentielles.

```
[action]
lower = 20.0
upper = 40.0
intervals =  100

[parameter]
id=1 // Speed
initial = 0.3
lower = -0.2
upper = 2.2
intervals = 300
level = 0

[input]
id = 2 // Expected speed
lower = 0.0
upper = 2.0
initial = 0.0
intervals = 400
perturbation = 0.005
level = 0
link = 1
```

FIGURE 4.2 – Extrait d'un fichier de description de modèle

4.2 La synthèse

La partie synthèse commence le calcul depuis l'état initial avec une exploration en avant. Du fait de la discrétisation, depuis un état discret s, une action a peut aboutir à plusieurs états successeurs. Dans la terminologie utilisée précédemment, le contrôleur choisit une action et l'environnement choisit les successeurs (voir Figure 4.3). Pour chaque état discret s et chaque action a, les états successeurs sont calculés en résolvant le système EDO à partir de plusieurs points réels appartenant à s. Ces points sont choisis de façon à obtenir une couverture des successeurs possibles. Nous obtenons alors un ensemble P de points réels, chacun étant dans $\mathbb{R}^n$ où n est le nombre de variables.

4.2.1 Perturbations

Pour chaque point $p \in P$, nous définissons les points p_+ et p_- de la façon suivante : $p_+ = p + \epsilon$ et $p_- = p - \epsilon$, où ϵ est une valeur donnée en paramètre appelée *perturbation*. Ces points sont ajoutés à l'ensemble des successeurs : $P = P \cup \{p_+\} \cup \{p_-\}$ puis une clôture est réalisée sur cet ensemble P (voir ci dessous le détail d'implémentation de la clôture).

En connaissant la perturbation p et la période d'échantillonage η, on peut déterminer la pente maximale d'une variable (voir Figure 4.4).

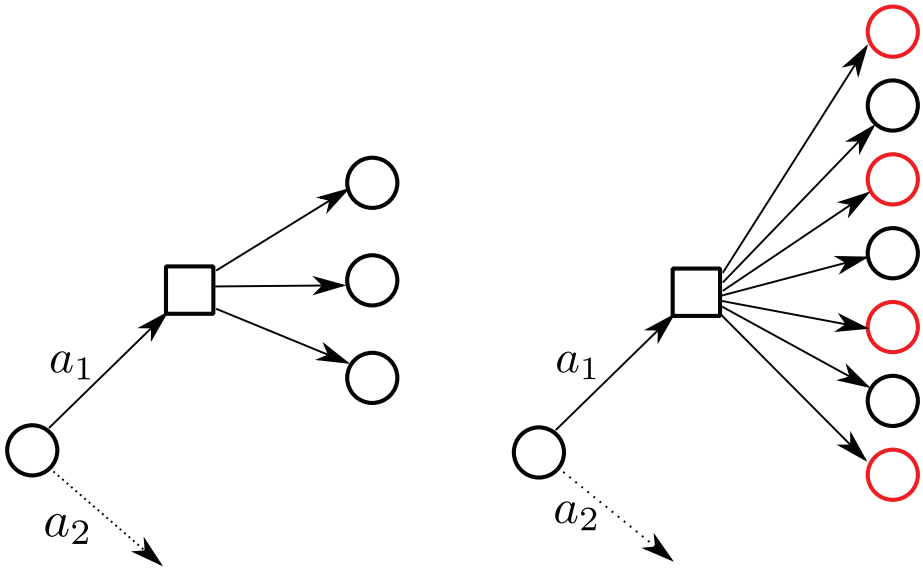


FIGURE 4.3 – Transitions sans et avec des perturbations

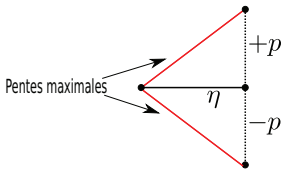


FIGURE 4.4 – Pentes maximales en fonction de p et η

4.2.2 Optimisations

Nous avons réalisé plusieurs optimsations afin de réduire les temps de calcul et la mémoire utilisée. La première optimisation concerne le stockage des prédécesseurs dans la fonction *depend* des algorithmes de la section 2.2. Une première approche naïve consisterait à associer à chaque état une liste chaînée de tous ses prédécesseurs, ce qui engendre un surcout en mémoire devenant au fil des calculs difficilement tenable. Notre approche consiste à réaliser une *couverture* des états plutôt que de stocker les états eux mêmes. En d'autres termes, nous ne stockons qu'un état *minimum* et un état *maximum*, de sorte que tous les états compris entre ces deux bornes sont considérés comme étant des prédécesseurs. A titre d'exemple prenons un état s quelconque en dimension 2. Si cet état possède 3 prédécesseurs $(1, 10)$, $(3, 4)$ et $(9, 2)$ alors tous les états compris entre $(1, 2)$ et $(9, 10)$ (bornes incluses) seront considérés comme des prédécesseurs. En procédant ainsi nous obtenons une sur-approximation de l'ensemble des états

prédécesseurs ce qui ne pose pas de problèmes car si une solution existe pour cette sur-approximation alors elle existera nécessairement pour les prédécesseurs réels. Une seconde optimisation consiste à réaliser simultanément l'exploration du graphe et le calcul de Stay (algorithmes 1 et 2) afin d'éliminer d'avantage d'états lors des calculs. A cette optimisation se rajoute la propagation en arrière des états ne pouvant rester dans Allow (en plus de ceux ne pouvant rester dans Goal), éliminant ainsi un nombre conséquent d'états supplémentaires.

4.3 Génération liste états/actions

Si les calculs aboutissent (c'est à dire si un chemin est trouvé depuis les états initiaux vers l'ensemble d'états Goal et que le chemin y demeure) alors un fichier texte est créé comportant la liste des états avec les actions associées (permettant d'obtenir une stratégie Stabilize).

4.4 Calcul d'une stratégie

Le calcul d'une stratégie est effectué indépendamment de la synthèse. Les paramètres du modèle sont récupérés ainsi que la liste des états avec les actions associées permettant d'obtenir une stratégie gagnante. La stratégie commence par l'état initial réel s_0. Cet état est discrétisé puis une action valide est sélectionnée dans la liste des états/actions. Le système EDO est résolu pour cet état réel et nous obtenons ainsi un nouvel état s'. Le processus est alors répété. A partir d'un état, il peut y avoir plusieurs actions acceptables et le choix de l'une ou de l'autre dépend si l'état est dans G ou pas. Si l'état est dans G alors toute action est possible mais d'un point de vue pratique moins la force appliquée à un système varie et moins ce dernier subit de contraintes pouvant accélérer son usure. Dans l'exemple $s \xrightarrow{a} s_1 \xrightarrow{a'} s_2$, l'action a' sera choisie telle que $|a' - a|$ soit minimal.

4.4.1 Objectifs dynamiques

Nous traiterons du calcul des stratégies pour les objectifs dynamiques au Chapitre 5.

CHAPITRE 5

Objectifs dynamiques

5.1 Contrôleurs classiques

5.1.1 Synthèse

Jusqu'à présent nous avons réalisé des synthèses de contrôleur pour des objectifs prédéfinis, fixés avant le lancement des calculs. Nous allons à présent étudier des objectifs évoluant dans le temps ou dit autrement des objectifs pour lesquels l'état Goal se déplace dans le temps (voir Figure 5.1).

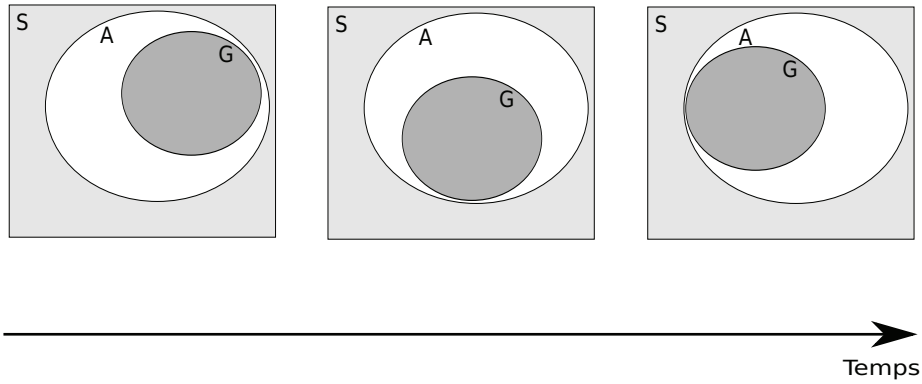


FIGURE 5.1 – Objectifs évoluant avec le temps

Pour ces objectifs, nous avons utilisé comme modèles les drones U130 de la société Novadem (voir Figure 2). Ces drones sont semi autonomes : depuis

une télécommande des *consignes* sont données et le drone essaye de les réaliser au mieux. Ces consignes sont des objectifs évoluant dans le temps. Nous avons étudié dans un premier temps des consignes de vitesse ascensionnelle : l'une progressive dite en rampe et l'autre raide sous forme d'échelon (voir Figure 5.2).

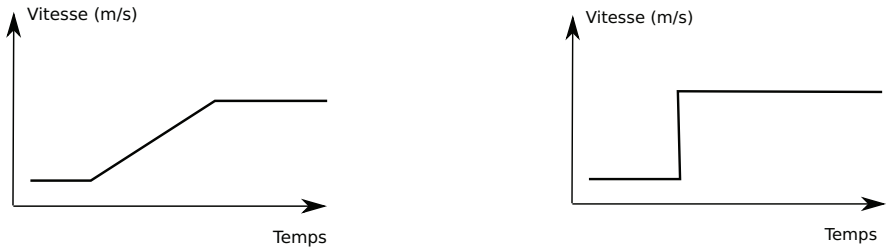


FIGURE 5.2 – Consignes en rampe et en échelon

Les algorithmes utilisés sont les mêmes que ceux décrits dans les chapitres précédents. Le système a été modélisé de la façon suivante :

- Une variable x_1 représentant la vitesse ascensionnelle
- Une entrée x_2 représentant l'environnement, incontrôlable

L'état du système (x_1, x_2) est dans l'objectif (c'est à dire $(x_1, x_2) \in \mathsf{Goal}$) si $|x_1 - x_2| < \delta$ avec δ une constante prédéfinie, représentant la « tolérance » par rapport à l'objectif à atteindre. Une particularité de notre approche est l'utlisation d'une *perturbation p*. Pour chaque trajectoire calculée aboutissant au point y, l'ensemble des états successeurs est alors $[y-p, y+p]$ (voir Figure 5.3).

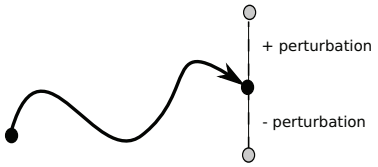


FIGURE 5.3 – Utilisation des perturbations

Le système EDO décrivant la dynamique du drone est alors :

$$\begin{cases} \dot{x_1} &= (F - m * g)/m \\ \dot{x_2} &= 0 \end{cases} \tag{5.1}$$

L'équation $\dot{x_1} = (F - m * g)/m$ est une application du principe fondamentale de la dynamique : $\Sigma F_i = m.a$, tandis que $\dot{x_2} = 0$ donnera une valeur constante pour x_2, les variations de cette variable seront dues uniquement aux perturbations.

5.1.2 Utilisation du contrôleur

Le contrôleur synthétisé lit en entrée les objectifs au fur et à mesure de l'écoulement du temps. Le cas de la consigne en rampe de la Figure 5.2 ne pose pas de problème particulier : du fait de la pente douce le contrôleur pourra respecter l'objectif. La consigne en échelon avec sa variation trop rapide ne permet pas au contrôleur d'atteindre l'objectif instantanément, c'est à dire si $o(t)$ est la valeur maximale de l'objectif au temps t, alors on aura pendant un laps de temps $|x_1 - o(t)| > \delta$.

La solution adoptée consiste à *relâcher* l'objectif afin de ne plus chercher à être à sa « proximité » mais à s'en rapprocher avec la pente *maximale* possible pour le système, voir Figure 5.4.

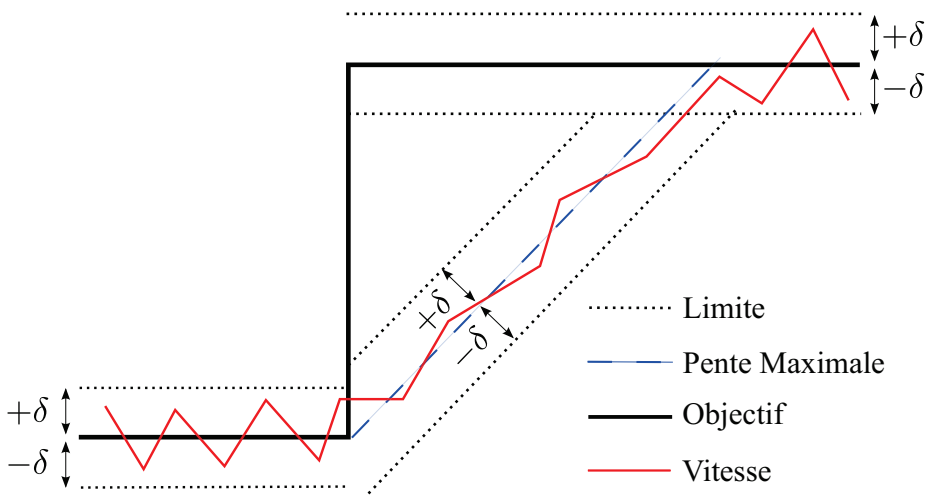


FIGURE 5.4 – Relâchement de l'objectif

A la Figure 5.5, nous avons représenté une stratégie pour un objectif de vitesse en échelon. Les paramètres sont $p_1 = 0$, $p_2 = 0.005$, $\delta = 0.05$ et $\tau = 0.01$. Le drone étant initialement hors de la zone Goal, il utilisera une stratégie Until pour y parvenir et s'y maintenir.

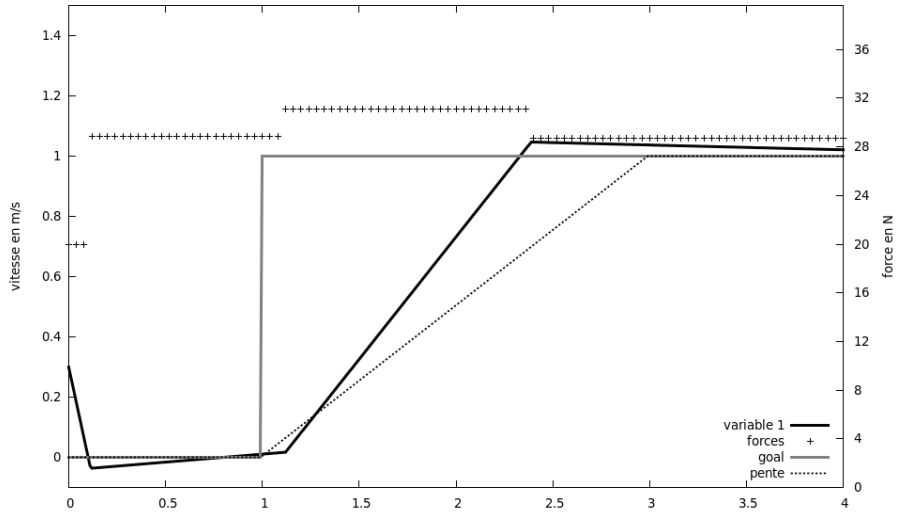


FIGURE 5.5 – Stratégie pour une consigne en échelon

5.2 Contrôleurs PID

Un contrôleur PID (aussi appelé régulateur PID) réalise les mêmes fonctions que les contrôleurs décrits dans les chapitres précédents : à partir d'un état courant et d'une consigne (un objectif) le contrôleur PID va émettre une action afin de s'approcher au mieux de cette consigne. C'est le régulateur le plus utilisé dans l'industrie.

PID signifie « Proportionnel Intégral Dérivé » et comme son nom le suggère se compose de trois éléments (voir Figure 5.6). Ces éléments sont 3 contrôleurs pouvant en théorie fonctionner indépendamment les uns des autres, mais en pratique certaines associations sont plus efficaces que d'autres.

Nous allons détailler les différents éléments avant de montrer la façon dont nous avons implémentée de tels contrôleurs. L'intérêt est de s'approcher au plus près des systèmes physiquement implémentés dans les drones de la société Novadem et d'ouvrir la voie à de nouveaux types de contrôleurs.

5.2.1 Description des différents contrôleurs

Contrôleur Proportionnel P

Le contrôleur P est le plus simple des 3 contrôleurs de cette section : il reçoit en entrée la différence entre la consigne attendue et le signal transmis (nous appelerons cette valeur $e(t)$) et transmet en sortie ce même signal pondéré par un coefficient K_p. Le signal en sortie est donc $K_p.e(t)$. Le coefficient K_p est en général déterminé de manière expérimentale : plus sa valeur est grande et plus

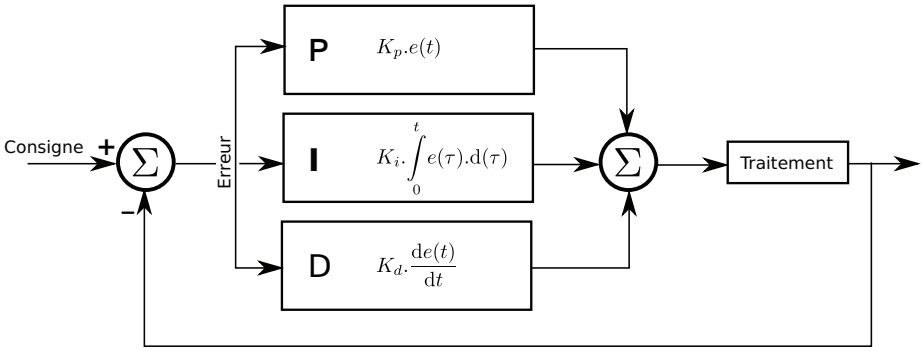


FIGURE 5.6 – Régulateur PID

vite la correction sera réalisée mais au risque d'une grande imprécision voire une impossibilité de s'approcher raisonnablement de la consigne. Un coefficient trop petit peut avoir comme conséquence un délai trop important pour approcher de la consigne et peut générer une oscillation néfaste du système qui ne parviendra pas à se stabiliser. Le contrôleur P semble suffisant à lui seul. Prenons l'exemple d'un chauffage avec thermostat, un contrôleur P avec un coefficient $\frac{1}{3}$ et un consigne fixe à 20 degrés. Si la température du local est de 17 degrés, l'erreur sera donc de 3 degrés et la sortie du contrôleur sera donc 1 signifiant que la vanne du chauffage doit être totalement ouverte. La température va donc monter et lorsqu'elle arrivera à 19 degrés, la vanne restera ouverte de $\frac{1}{3}$ et la température n'augmentera plus car si elle augmente la vanne se refermera et donc le local se refroidira, etc. Pour avoir du chauffage, la vanne doit être ouverte, un écart de température doit donc exister.

Contrôleur Proportionnel-Intégral PI

Afin de pallier aux imperfections du contrôleur P, la composante Intégrale va additioner l'erreur $e(t)$ à chaque pas de temps. La valeur résultante, pondérée par le coefficient K_i, sera ajoutée à celle du contrôleur P. Une fois que la consigne est atteinte (c'est à dire $e(t) = 0$), la composante intégrale n'est alors plus modifiée.

Contrôleur Proportionnel-Intégral-Dérivé PID

Dans certains systèmes, par exemple pour les systèmes à air conditionné, les variations de $e(t)$ risquent d'être trop rapides en fonction de la position d'ouverture de la vanne. Une troisième composante est donc utilisée venant s'ajouter aux deux précédentes et dont la valeur est d'autant plus grande que l'écart varie rapidement, c'est à dire proportionnelle à la dérivée de $e(t)$ par rapport au temps.

5.2.2 Implémentation

L'implémentation des différents contrôleurs présentés dans cette secion n'a pas nécessité de modification des algorithmes précédents, seuls les modèles décrits dans les fichiers textes ont du être modifiés.

Contrôleur P

Le contrôleur P devra aussi traiter des objectifs dynamiques et par conséquent deux variables sont nécessaires comme dans la Section 5.1. Le calcul $K_p.e(t)$ du contrôleur P fournira directement la force à appliquer dans le système EDO. Dans le cas du drone, il s'agit du même système que celui de la Section 5.1 :

$$\begin{cases} \dot{x_1} &= (F - m * g)/m \\ \dot{x_2} &= 0 \end{cases} \tag{5.2}$$

Puisque la force est directement donnée par le contrôleur PI, il n'est plus nécessaire d'utiliser une discrétisation pour cette dernière, ce qui permet de diminuer l'espace d'états à explorer. Afin de ne pas modifier les algorithmes principaux, nous avons utilisé une discrétisation pour les actions comportant un seul intervalle :

```
[action]
lower = 20.0
upper = 40.0
intervals = 1
```

Il a été alors possible d'augmenter la discrétisation des variables de vitesse et de consigne (respectivement 2000 et 200 intervalles) et d'utiliser une perturbation de 0.01 afin de parvenir à calculer un contrôleur P acceptant des consignes dynamiques :

```
[parameter]
id=1 // Speed
initial = 0.3
lower = -0.2
upper = 2.2
intervals =2000
level = 0
```

```
[input]
id = 2 // Expected speed
lower = 0.0
upper = 2.0
initial = 0.3
intervals = 200
perturbation = 0.01
level = 0
link = 1
```

La Figure 5.7 montre une stratégie pour un contrôleur P avec une consigne en échelon. Les paramètres sont $p_1 = 0$, $p_2 = 0.01$, $\delta = 0.5$ et $\tau = 0.01$. La forme de la courbe est très proche de celle obtenue à l'aide des contrôleurs électroniques, beaucoup plus rapide et plus lisse que celle de Figure 5.5. A partir du temps 2.0, les forces commencent à se stabiliser, ce qui permet une utilisation rationnelle des éléments mécaniques. On notera la valeur élevée de δ, sans quoi la synthèse n'aurait pas été possible.

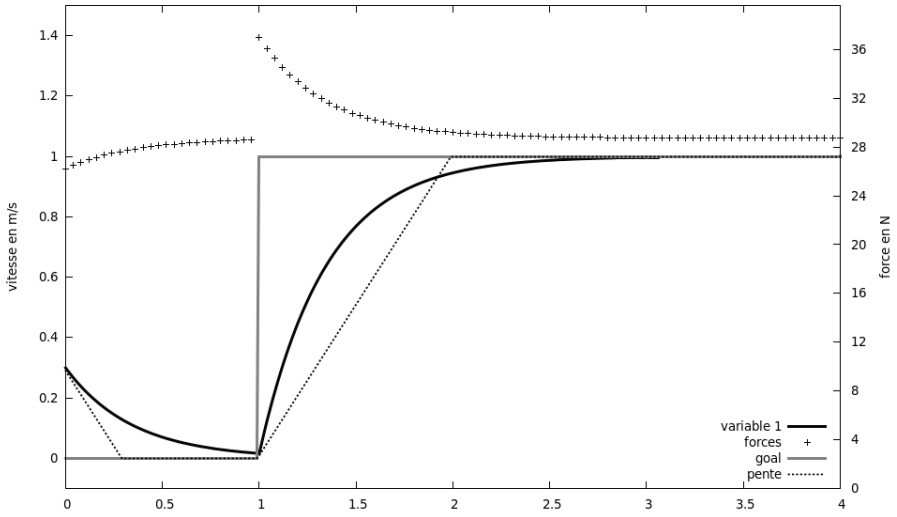


FIGURE 5.7 – Stratégie pour un Contrôleur P avec consigne en échelon

Deuxième partie

Model checking

CHAPITRE 6

Séquences temporisées

Nous allons introduire la notion de *Thread* dans le cadre d'un programme. Un thread est un fil séquentiel d'instructions, il représente l'exécution d'un ensemble d'instructions du langage machine d'un processeur. Plusieurs threads peuvent cohabiter sur le même processeur et d'un point de vue de l'utilisateur, ils semblent s'exécuter *en parallèle*.

Nous appelerons T_i avec $1 \leq i \leq N$ un thread, $trans_i$ son ensemble de transitions et $trans = \bigcup trans_i$ l'ensemble de toutes les transitions. Soit V_i l'ensemble des variables locales de T_i, V_g l'ensemble des variables globales et $V = V_g \cup \bigcup V_i$ l'ensemble de toutes les variables. Chaque variable $v \in V$ prend ses valeurs dans le domaine D_v. L'instruction en cours d'exécution d'un thread T_i est pointée par une variable locale $pc_i \in V_i$ (program counter).

Un *état* du programme est une valuation de ses variables locales et globales. Formellement $s : V \longrightarrow \bigcup D_v$ avec $s(v) \in D_v$. L'ensemble de tous les états sera noté $S = \prod D_v$. Nous admettons l'existence d'un *état initial* s_0, c'est à dire d'une valuation initiale des variables.

Les expressions sur les variables sont définies de manière usuelle, c'est à dire comme une combinaison de littéraux, de variables, d'opérateurs et de fonction (par exemple $2+3*9$). Les atomes sont des comparaisons de paires d'expressions et les *conditions* sont des combinaisons booléennes des atomes.

Syntaxiquement, une transition t de T_i est exécutable sous une *condition* (aussi appelée *garde*) sur $V_i \cup V_g$, et à comme effet de bord une *action* définie comme un ensemble d'affectations d'expressions vers des variables, c'est à dire que les valeurs des variables sont *écrites*. Pour les actions et les conditions, les variables apparaissant dans les expressions sont *lues*.

Exemple 5: *Soit la transition*

$$s \xrightarrow[x<3;x:=2+y]{t} s'$$

Dans la condition $x < 3$ la variable x est lue alors que dans l'action $x := 2 + y$ la variable x est écrite et la variable y est lue

Si t est une transition allant d'une position loc_1 du programme vers une position loc_2, alors la condition de t inclut $pc_i = loc_1$ et l'action inclut $pc_i := loc_2$.

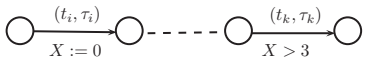
Pour deux états $s, s' \in S$, $s \xrightarrow{t} s'$ décrit une transition exécutable depuis s et transformant s en s' en appliquant l'action de t. On notera $s \xrightarrow{t_1} s_1 \ldots \xrightarrow{t_n} s_n = s'$ une séquence d'exécutions de transitions.

6.1 Notions de temps

Définition 19. *Soient des variables réelles appelées* horloges *qui diffèrent des autres variables sur les points suivants :*

- *Leurs valeurs augmentent de façon synchrone et proportionnelle au temps : si x a la valeur ρ au temps τ alors il aura la valeur $\rho + \delta$ au temps $\tau + \delta$*
- *La seule affectation possible est la remise à zéro*
- *Nous autorisons seulement les comparaisons entre une horloge est un nombre constant entier (par exemple $x \leq 3$)*
- *A l'état initial s_0, la valeur des horloges est égale à zéro*

A chaque exécution d'une transition t_k, nous allons associer un *instant d'exécution* $\tau_k \in \mathbb{R}^+$ ou *timestamp* afin de pouvoir déterminer si les conditions sur les horloges sont bien remplies ou pas. On notera $s \xrightarrow{(t_k,\tau_k)} s'$ la transition temporisée t_k exécutée au temps τ_k. De même on notera $s_0 \xrightarrow{(t_1,\tau_1)} s_1 \ldots \xrightarrow{(t_n,\tau_n)} s_n = s'$ une séquence temporisée.



$$\tau_k - \tau_i > 3$$

FIGURE 6.1 – Remise à zéro et contrainte sur une horloge X

A la figure 6.1, l'horloge X est remise à zéro à l'instant τ_i puis sa valeur est comparée avec la constante 3 à l'instant τ_k. Dit autrement, $X > 3$ compare le temps écoulé depuis la dernière remise à zéro de X, soit $\tau_k - \tau_i$. La condition $X > 3$ peut alors se réécrire $\tau_k - \tau_i > 3$.

Définition 20 (Progression normale du temps)**.** *Une séquence temporisée $s_0 \xrightarrow{(t_1,\tau_1)} s_1 \ldots \xrightarrow{(t_n,\tau_n)} s_n = s'$ satisfait une* progression normale *du temps si et seulement si pour toute paire $k < l$ on a $\tau_k \leq \tau_l$.*

Exemple 6: *La séquence* $s_0 \xrightarrow{(t_1,3.6)} s_1 \xrightarrow{(t_1,4.6)} s_2 \xrightarrow{(t_1,7.3)} s_3$ *respecte une progression normale du temps alors que la séquence* $s_0 \xrightarrow{(t_1,4.6)} s_1 \xrightarrow{(t_1,2.1)} s_2 \xrightarrow{(t_1,7.3)} s_3$ *ne la respecte pas.*

Dans notre formalisme, le problème de l'accessibilité dans les automates temporisés se traduit par l'existence d'une séquence temporisée avec une progression normale du temps menant d'un état s_0 vers un état s' satisfaisant une propriété donnée.

6.2 Concurrence

Dans cette section, nous allons revoir certaines notions sur les méthodes à ordres partiels avant d'introduire les *multisteps* c'est à dire l'exécution *en parallèle* de plusieurs transitions et nous verrons comment analyser des systèmes temporisés en utilisant les multisteps.

Une définition classique utilisée dans l'analyse de systèmes concurrents est celle de la dépendance pour les lecteurs-rédacteurs introduit dans [CHP71] :

Définition 21 (Dépendance)**.** *Deux transitions* t_1 *et* t_2 *sont* dépendantes *si une variable lue par* t_1 *(dans la garde ou dans l'action) est écrite par* t_2 *(ou vice versa), ou bien si la même variable est écrite par* t_1 *ou* t_2*. Sinon elles sont* indépendantes.

Exemple 7: *Soient les deux transitions suivantes :*

$$s_i \xrightarrow[x<3;y:=0]{(t_i,\tau_i)} s_j$$

$$s_k \xrightarrow[y\geq 4;z:=3]{(t_k,\tau_k)} s_l$$

Dans la transition t_i*, la variable* x *est lue et la variable* y *est écrite alors que dans la transition* t_k *la variable* y *est lue et* z *est écrite. Ces deux transitions sont donc dépendantes (lecture et écriture sur la variable* y*). Par contre les deux transitions suivantes sont indépendantes :*

$$s_i \xrightarrow[x<3;y:=0]{(t_i,\tau_i)} s_j$$

$$s_k \xrightarrow[x\geq 4;z:=3]{(t_k,\tau_k)} s_l$$

Nous allons à présent introduire une nouvelle notion de progrès du temps, moins intuitive que la progression normale, utilisant l'indépendance entre les transitions :

Définition 22 (Progression relâchée du temps)**.** *Une séquence temporisée* $s_0 \xrightarrow{(t_1,\tau_1)} s_1 \ldots \xrightarrow{(t_n,\tau_n)} s_n = s'$ *satisfait une* progression relâchée *du temps si pour toute paire* $k < l$ *avec* t_k, t_l *dépendantes, nous avons aussi* $\tau_k \leq \tau_l$*.*

Exemple 8: *Soit la séquence temporisée*

$$s_0 \xrightarrow{(t_1,3)} s_1 \xrightarrow{(t_2,3.5)} s_2 \xrightarrow{(t_3,9.2)} s_3 \xrightarrow{(t_4,16.4)} s_4 = s'$$

satisfaisant une progression normale du temps. Si l'on admet que les transitions t_2 *et* t_3 *sont indépendantes et que* t_3 *et* t_4 *sont dépendantes, alors la séquence*

$$s_0 \xrightarrow{(t_1,3)} s_1 \xrightarrow{(t_3,9.2)} s'_2 \xrightarrow{(t_2,3.5)} s_3 \xrightarrow{(t_4,16.4)} s_4 = s'$$

satisfait une progression relâchée du temps. On remarquera l'inversion dans l'ordre d'exécution des transitions t_2 *et* t_3 *ce qui est permit par la définition de la progression relâchée.*

Nous allons à présent montrer l'équivalence entre les deux progressions du temps vues jusqu'à présent :

Proposition 4. *Si une séquence temporisée satisfait une progression normale du temps alors elle satatisfait aussi une progression relâchée.*

Démonstration. La preuve est immédiate. □

Définition 23 (Equivalence de Mazurkiewicz)**.** *Deux séquences temporisées sont équivalentes au sens de Mazurkiewicz si l'une peut être transformée en l'autre par un nombre fini d'échanges de transitions adjacentes et indépendantes.*

$$s_0 \dots s_{k-1} \xrightarrow{(t_k,\tau_k)} s_k \xrightarrow{(t_{k+1},\tau_{k+1})} s_{k+1} \dots s'$$

$$\equiv$$

$$s_0 \dots s_{k-1} \xrightarrow{(t_{k+1},\tau_{k+1})} s_k{}' \xrightarrow{(t_k,\tau_k)} s_{k+1} \dots s'$$

$$t_k, t_{k+1} \text{ indépendantes}$$

Proposition 5. *Si une séquence temporisée satisfait une progression relâchée du temps, alors les séquences équivalentes (au sens de Mazurkiewicz) aussi. De plus, pour toute séquence temporisée satisfaisant une progression relâchée il existe une séquence temporisée équivalente satisfaisant une progression normale.*

Démonstration. La preuve a été originellement démontrée dans [LNZ05]. Dans la première partie, l'ordre d'exécutions des transitions dépendantes est préservé par les échanges et ainsi la progrès relâché du temps est toujours préservé. Dans la seconde partie, il est possible de transformer une séquence temporisée respectant un progrès relâché en appliquant un « tri à bulles » : si deux transitions adjacentes sont dans un mauvais ordre par rapport à leur timestamp, alors le progrès relâché implique que ces deux transitions sont indépendantes et il est possible de les échanger. Par induction, on obtient une séquence temporisée respectant un progrès normal du temps. □

CHAPITRE 7

Séquences Multistep

Les notions présentées dans cette section sont apparues en premier dans le contexte des réseaux de Petri (non temporisés) sous le nom de *step semantics* [GLT80], généralisées ici aux automates temporisés.

Définition 24. *Soit* $MT = \{(t_1, \tau_1), ..., (t_n, \tau_n)\}$ *un ensemble de transitions deux à deux indépendantes et dotées de leur timestamp : cet ensemble est* exécutable en concurrence *à partir de l'état global* s *si et seulement si chacune des transitions* t_i *est exécutable à partir de l'état* s *et à l'instant* τ_i*. Un tel ensemble sera appelé* multistep*.*

On remarquera que la définition précédente implique que si $(t, \tau_a), (t, \tau_b) \in MT$ alors $\tau_a = \tau_b$ parce qu'une transition est toujours dépendante d'elle-même. Par la définition de l'indépendance, toutes les exécutions possibles utilisant ces transitions et démarrant à l'état s sont équivalentes, et aboutissent toutes au même état s'. Nous dirons alors qu'elles peuvent être exécutées en parallèle et nous écrirons $s \xrightarrow{MT} s'$.

Une *séquence temporisée multistep* est une séquence $s_0 \xrightarrow{MT_1} s_1 ... \xrightarrow{MT_n} s_n = s'$. L'intérêt des multisteps est immédiat : puisque plusieurs transitions sont exécutées à chaque multistep (en parallèle), les exécutions peuvent être plus courtes (c'est à dire qu'un plus petit nombre de multisteps peut être exécuté) pour atteindre un état donné qu'avec des exécutions par *entrelacement* c'est à dire des exécutions ne comportant qu'une seule transition à chaque étape, comme celles décrites dans les sections précédentes sous la forme $s_0 \xrightarrow{(t_1,\tau_1)} s_1 ... \xrightarrow{(t_n,\tau_n)} s_n = s'$.

Exemple 9: *Nous allons illustrer les multisteps à l'aide d'un exemple non temporisé inspiré du problèmes des 8 reines : le problème des cavaliers (voir Fi-*

gure 7.1). Dans ce problème, des cavaliers sont placés sur un échiquier de taille quelconque.

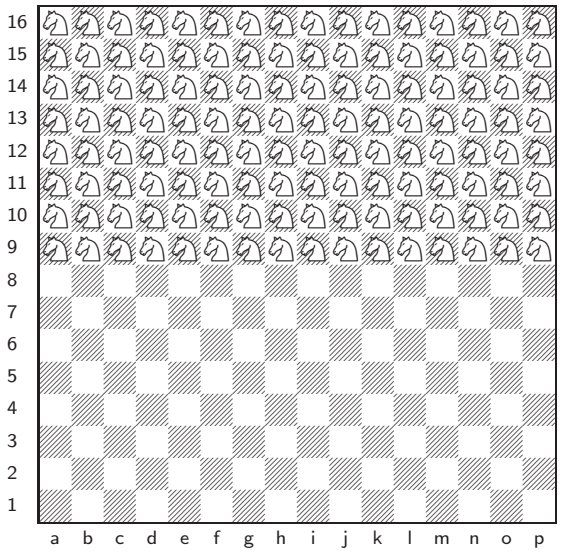


FIGURE 7.1 – Le problème des cavaliers

Les cavaliers peuvent manger et se déplacer en respectant les régles en vigueur aux échecs : de deux cases en suivant la verticale puis une case en suivant l'horizontale (et vice versa, c'est à dire deux cases en suivant l'horizontale puis une case sur la verticale). Un cavalier peut passer par dessus d'autres pièces, voir la Figure 7.2.

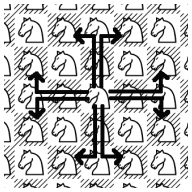


FIGURE 7.2 – Déplacements possibles d'un cavalier

Le problème consiste à déplacer les cavaliers de façon à ce qu'aucun d'entre eux ne puisse manger un de ses congénères, en respectant toujours les règles des échecs. Une approche intuitive serait de déplacer les cavaliers un par un jusqu'à trouver une solution (méthode par entrelacement). Bien que simple à mettre en œuvre cette solution demande beaucoup d'étapes avant d'atteindre une solution. Une autre possibilité serait de déplacer plusieurs cavaliers simultanément sur

des cases libres, chaque cavalier devant respecter les règles des échecs et dans ce cas précis le fait qu'il ne peut y avoir qu'un seul cavalier par case à un moment donné. En examinant l'échiquier, on constate qu'il est possible de déplacer en une seule étape tous les cavaliers des lignes 9 et 10 (Figure 7.3). A la seconde étape, les lignes 7 et 8 peuvent être déplacées, de même que poursuivre le déplacement des deux lignes précédentes (Figure 7.4). On poursuit ainsi jusqu'à trouver une solution (Figures 7.5 et 7.6).

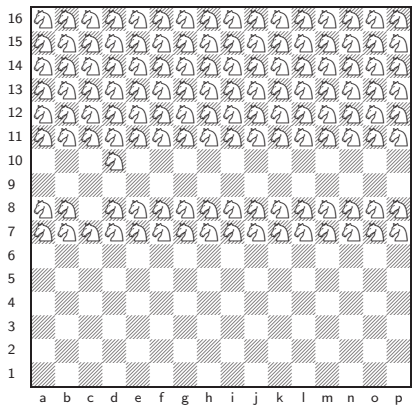


FIGURE 7.3 – Etape 1

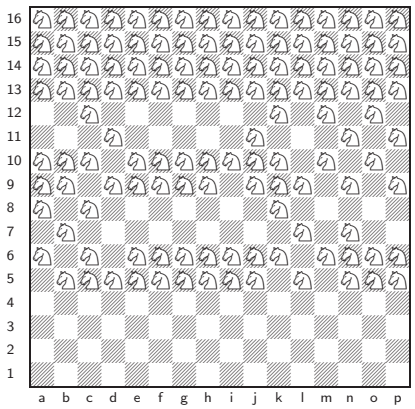


FIGURE 7.4 – Etape 2

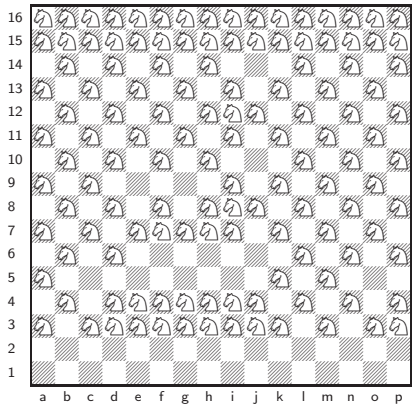


FIGURE 7.5 – Etape 3

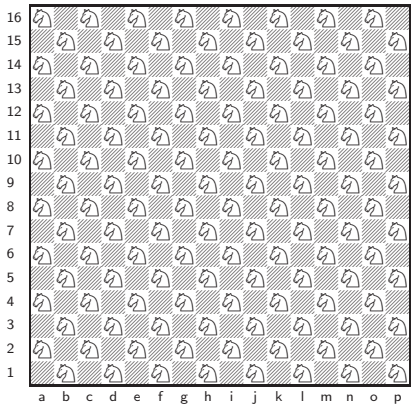


FIGURE 7.6 – Etape 4

7.1 Multisteps et progrès du temps

Tout comme pour les progrès normal et relâché pour les séquences temporisées avec entrelacement de la section 6.1, nous allons présenter plusieurs progrès possibles du temps : synchrone, semi-synchrone et relâché pour répondre à la question : « Que signifie d'exécuter plusieurs transitions temporisées en parallèle ? ».

7.1.1 Progrès synchrone

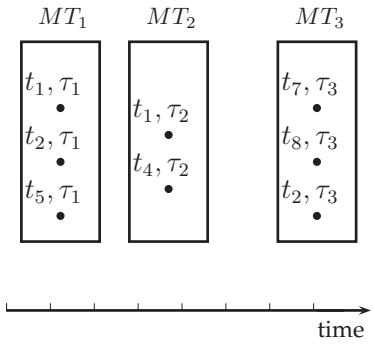


FIGURE 7.7 – Progrès du temps synchrone

Soit des multisteps $MT_1, MT_2, ...$ chacun composé de transitions indépendantes entre elles. Une première approche intuitive consiste à considérer que dans un multistep toutes les transitions ont lieu au même instant τ et que le temps progresse *entre* les multisteps (Figure 7.7). Ce progrès ressemble au progrès normal pour les séquences entrelacées de la Section 6.2. Plus formellement : pour tout multistep MT_k et pour toute paire de transitions $(t_1, \tau_1), (t_2, \tau_2) \in MT_K$ nous avons $\tau_1 = \tau_2$. De plus pour tout $k < l$ et pour tous $(t_a, \tau_a) \in MT_k, (t_b, \tau_b) \in MT_l$ nous avons $\tau_a \leq \tau_b$.

7.1.2 Progrès semi-synchrone

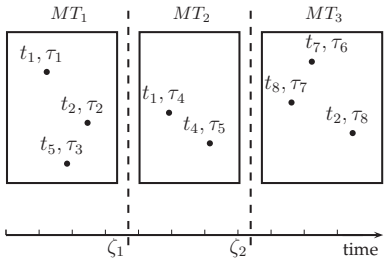


FIGURE 7.8 – Progrès du temps semi-synchrone

Le progrès semi-synchrone assouplit la règle du progrès synchrone en permettant aux transitions d'un multistep de ne pas s'exécuter toutes en même temps, mais avant celles des multisteps suivants (voir Figure 7.8). Formellement : pour tout $k < l$ et pour tous $(t_a, \tau_a) \in MT_k, (t_b, \tau_b) \in MT_l$, nous avons $\tau_a \leq \tau_b$.

7.1.3 Progrès relâché

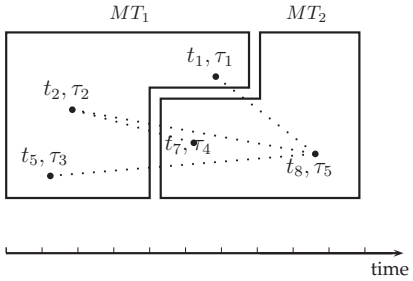


FIGURE 7.9 – Progrès du temps relâché

La notion de progrès relâché pour les multisteps reprend celle pour les séquences avec entrelacement : deux transitions dépendantes doivent respecter l'ordre de leur timestamps. Avec cette définition, rien n'empêche une transition d'un multistep d'être exécutée *après* une transition d'un multistep suivant. Formellement : pour tout $k < l$ et pour toute $(t_a, \tau_a) \in MT_k, (t_b, \tau_b) \in MT_l$ telle que t_a et t_b sont dépendants, nous avons $\tau_a \leq \tau_b$.

7.2 Comparaison des différents progrès

Nous allons dans cette section montrer qu'il est possible de transformer chacun des cinq progrès présentés en un autre à l'aide d'un nombre fini de transformations.

Lemme 3. *Soit* $s_0 \xrightarrow{MT_1} s_1 ... \xrightarrow{MT_n} s_n = s'$ *une séquence de multisteps satisfaisant un progrès relâché du temps, alors il existe une séquence temporisée entrelacée* $s_0 \xrightarrow{(t_1,\tau_1)} s_1 ... \xrightarrow{(t_m,\tau_m)} s_m = s'$ *avec* $m = \sum_i |MT_i|$ *(le nombre total de transitions simples) satisfaisant un progrès relâché du temps.*

Démonstration. On construit une séquence $s_0 \xrightarrow{(t_1,\tau_1)} s_1 ... \xrightarrow{(t_m,\tau_m)} s_m = s'$ en extrayant chaque couple (t_i, τ_i) de chaque MT_k en respectant l'ordre des MT_k (l'ordre des (t_i, τ_i) issues d'un même MT_k n'est pas important car toutes les transitions de ce MT_k sont par définition indépendantes). □

Théorème 1. *Soit* $s, s' \in S$*, alors chacune des séquences suivantes peut être transformée en toutes les autres :*

1. *Une séquence temporisée* $s \xrightarrow{(t_1,\tau_1)} ... \xrightarrow{(t_n,\tau_n)} = s'$ *avec un progrès normal du temps*
2. *Une séquence temporisée* $s \xrightarrow{(t_1,\tau_1)} ... \xrightarrow{(t_n,\tau_n)} = s'$ *avec un progrès relâché du temps*

3. *Une séquence multistep* $s \xrightarrow{MT_1} ... \xrightarrow{MT_k} s'$ *avec un progrès du temps synchrone*
4. *Une séquence multistep* $s \xrightarrow{MT_1} ... \xrightarrow{MT_l} s'$ *avec un progrès du temps semi-synchrone*
5. *Une séquence multistep* $s \xrightarrow{MT_1} ... \xrightarrow{MT_m} s'$ *avec un progrès du temps relâché*

Démonstration.
- $1 \Rightarrow 2$: par définition
- $2 \Rightarrow 1$: voir Proposition 5
- $2 \Rightarrow 3$: on construit une séquence multistep $s \xrightarrow{MT_1} ... \xrightarrow{MT_n} s_n = s'$ dans laquelle chaque MT_k est composé d'une seule paire $\{(t_k, \tau_k)\}$. On obtient trivialement une séquence multistep avec un progrès du temps synchrone
- $3 \Rightarrow 4$: par définition
- $4 \Rightarrow 5$: par définition
- $4 \Rightarrow 2$: voir Lemme 3

□

CHAPITRE 8

Codage SAT

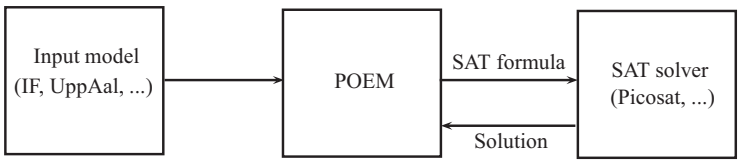


FIGURE 8.1 – Interactions autour du plugin SAT

L'idée d'utiliser la résolution SAT comme un outil pour le model checking apparaît au début des années 90 [SS90] et a été popularisé sous le nom « Bounded Model Checking » depuis la publication de [BCC99] dix ans plus tard. Ce développement va de paire avec les progrès dans la conception des solveurs SAT, certains d'entre eux ont été adaptés pour les besoins particulier du model checking [Bie08].

L'idée générale est de réduire un problème de recherche de chemin d'une taille bornée pour un objectif donné (reachability, safety, LTL, ...) en une formule booléenne qui possède une solution si et seulement si un chemin précédemment défini existe. De plus, ce chemin peut être reconstruit à partir de la solution. Les réductions SAT peuvent dans certains cas éviter l'explosion combinatoire se produisant dans l'exploration d'un graphe, car dans ce dernier cas une partie significative du graphe des états est explicitement construite. Contrairement aux méthodes de model checking basées sur l'exploration d'états telles qu'utilisées dans le model checker Spin [Hol03], les plus rapides solveurs SAT disponibles utilisent une petite quantité de mémoire. De plus, la propagation de contraintes utilisée dans la résolution SAT ne connait pas de direction particulière et peut

potentiellement combiner le meilleur de la recherche en avant et en arrière.

D'un autre côté, certaines réductions traitées dans un premier temps par des méthodes d'exploration d'états ont été transférées avec succés aux méthodes symboliques. C'est le cas en particulier pour les techniques d'ordre partiel, intégrées avec des méthodes symboliques de model checking [BCM+92, ABH+97]. C'est sans surprise que cela a été aussi étudié dans le contexte du Bounded Model Checking [Hel01].

Ce qui est plus surprenant est le peu d'attention que les travaux ultérieurs ont porté sur le model checking avec réductions SAT. Une des raisons est peut être que la réduction explorée dans [Hel01] est assez éloignée des « réductions d'ordre partiel » telles que nous allons les décrire. Le seul travail à notre connaissance allant dans la même direction que nous est [JN02].

Avec pour objectif l'exploration d'algorithmes distribués à l'aide du model checking, nous allons revisiter et expérimenter l'idée de [Hel01] dans une configuration plus proche des languages de modélisation basés sur des variables partagées et en utilisant une réduction en clauses plus standard (au format Dimacs). Cette partie est une explication technique de notre réduction utilisée conjointement avec les algorithmes des chapitres précédents.

Nos analyses utilisent le paradigme des *systèmes à réécriture de variables*. Dans ces systèmes, les transitions dépendent entièrement des valeurs des variables et les effets des transitions sont les affectations. Il s'agit du paradigme implémenté dans la plate-forme POEM que nous avons utilisée afin de tester différents algorithmes autour du thème des sémantiques à ordre partiel. Dans un sens plus large, les systèmes à réécriture de variables sont utilisés comme code intermédiaire dans POEM [NQ06b, KNQV06] et comme frontends pour des langages de modélisation comme IF2 de VERIMAG [BGM02], potentiellement de gros fragments de Promela (excluant certains aspects trop liés à l'analyseur de Spin). Dans l'exploration d'états utilisé par les backends de POEM, ce code est transformé en C de la même façon que Spin.

Dans les systèmes à réécriture de variables, *l'indépendance lecture-écriture* est utilisée pour identifier les transitions indépendantes, c'est à dire les paires de transitions opérant sur des ensembles disjoints de variables. Dans les réductions d'ordre partiel, cette forme d'indépendance est utilisée pour éliminer l'exploration de « chemins équivalents » : dans les systèmes de transitions déterministes, deux transitions indépendantes a et b exécutables à partir d'un état s peuvent être exécutées dans n'importe quel ordre ab ou ba et leur exécution aboutit au même état s'. Les réductions d'ordre partiel ne transforment pas le graphe d'états mais coupent des branches dans celui ci.

A contrario, dans [Hel01, JN02] tout comme dans notre approche SAT, il est permis à plusieurs transitions de s'exécuter « simultanément » c'est à dire qu'elles peuvent se combiner en un multistep commun. Cette possibilité augmente le degré de liberté du modèle et n'aurait aucun effet dans le cadre d'une exploration d'états. Avec les réductions SAT, les chemins peuvent être « compressés » en ce sens, donnant des instances SAT plus petites pour le même niveau de problème. Mais cette compression n'est pas seule responsable des améliorations apportées par les réductions SAT : en fait le solveur SAT n'a pas besoin

de choisir entre des entrelacements équivalents, alors qu'il existe un nombre exponentiel de possibilités. D'un point de vue des sémantiques d'ordre partiel, ces choix sont artificiels et peuvent être éliminés avec un effet sensible sur la résolution SAT. D'un autre côté [Hel01] s'est aussi intéressé à ce degré de liberté additionnel et suggère de les réduire en ajoutant des contraintes qu'il appelle *process semantics*. Alors que dans le travail cité, ces process semantics apportent de substantielles améliorations, nos expériences n'ont pas permis de confirmer ce fait : en réalité les formules augmentées par ces contraintes sont évaluées plus lentement par PicoSAT.

A la lueur de nos résultats, nous pensons que le potentiel des sémantiques à ordre partiel est plus intéressant pour le model checking basé sur SAT que les réductions d'ordre partiel pour l'exploration d'états.

Notre réduction SAT a été construite en partant de zéro. Le backend SAT est un « plugin » de POEM, réutilisant le frontend et l'analyse statique des transitions ainsi que des transformations du modèle. Dans la version courante, seuls « l'accessibilité bornée » et la vérification d'assertions sont implémentées. En dépit de certaines limitations connues de cette implémentation, le chapitre consacré aux expérimentations inclut des exemples dont l'espace d'état est de 10^{300} états et la découverte de chemins comportant plus de 2000 transitions, sans que l'on aperçoive de limites.

8.1 Communication avec le solveur SAT

Comme indiqué précédemment, le modèle en entrée est lu par la plate-forme POEM, analysé statiquement puis transformé en clauses CNF écrites dans un fichier texte au format DIMACS. Le fichier est ensuite transmis à un SAT solveur qui essaye de trouver une solution puis la transmet à nouveau sous forme d'un fichier texte indiquant si la formule est satisfaisable et dans ce cas les variables booléenne à 0 et celles à 1. Voici un exemple de fichier DIMACS :

```
c Ceci est un commentaire
p cnf 3 2
1 -3 0
2 3 -1 0
```

FIGURE 8.2 – Exemple de dichier dimacs

Dans ce format de fichier, les lignes commençant par la lettre c sont des commentaires, celles commençant par `p cnf` représentent l'entête du fichier est les deux nombres qui suivent sont resppectivement le nombre de variables booléennes et le nombre de clauses contenues dans la formule. Chaque ligne suivante spécifie une clause composée de littéraux : un littéral positif est indiqué par le nombre correspondant et un littéral négatif est noté par le nombre négatif correspondant. Le dernier nombre de la ligne doit être un 0 pour indiquer la fin de la clause. Dans l'exemple de la Figure 8.2, si l'on nomme a, b et c les trois

variables booléennes, la formule correspondante est $(a \vee \neg c) \wedge (b \vee c \vee \neg a)$. Le résultat du fichier SAT solveur est soit

```
s UNSATISFIABLE
```

pour indiquer que la formule n'a pas de solution et donc notre problème non plus, ou alors

```
s SATISFIABLE
v 1 2 3 -4 -5 6 7 -8 -9 10 11 -12
v -13 14 15 -16 -17 -18 -19 -20
v -21 22 -23 24 25 -26 27 -28 -29
```

Dans ce dernier cas, la première ligne indique qu'il existe une solution à notre formule et les lignes suivantes commençant par la lettre v comportent une liste de littéraux : si le littéral est positif cela signifie que la solution trouvée affecte la valeur 1 à la variable correspondante, si le littéral est négatif, la valeur 0 est affectée à la variable. A partir de ces résultats, la solution est reconstruite par POEM avant d'être affichée de diverses façons. Dans la suite nous utiliserons indifféremment le terme *litéral* ou le terme variable booléenne afin de désigner une variable pouvant prendre 2 valeurs.

8.2 Variables et expressions

Chaque *variable* $v \in V$ du modèle d'entrée est transformée en un vecteur de variables booléennes de taille $\log_2 |D_v|$. A titre d'exemple, examinons la déclaration suivant dans un modèle écrit en IF2 :

```
var x range 0..3
```

Cette commande déclare une variable x prenant ses valeurs dans le domaine [0..3] soit un total de 4 valeurs possibles. De fait, nous avons besoins d'un vecteur de variables booléennes de taille 2 : $\{x_1, x_2\}$.

Le *program counter* pc_i pour chaque thread T_i est codé comme une variable ordinaire, c'est à dire à l'aide d'un vecteur de variables booléennes dont la taille dépend du nombre d'états dans le thread.

Les expressions sont codées comme des circuits digitaux (voir par exemple l'additionneur de la Figure 8.3) pour lesquels chaque port est codé à l'aide d'un petit ensemble de clauses utilisant les variables d'entrée et de sortie, tandis que les *câbles* auxiliaires du circuit (qui ne sont ni des entrées, ni des sorties, il y en a par exemple 3 dans l'additionneur de la Figure 8.3) sont codés en utilisant des variables booléennes additionnelles.
Examinons quelques exemples :

- L'expression booléenne $a \wedge b$ sera transformée en $v := a \wedge b$ où v est la variable booléenne codant l'expression. Il en résulte les clauses

$$(v \vee \bar{a} \vee \bar{b}) \wedge (\bar{v} \vee a) \wedge (\bar{v} \vee b)$$

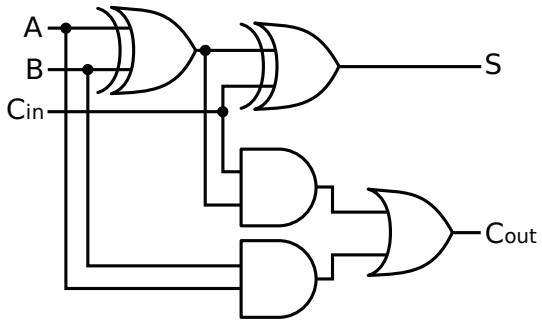


FIGURE 8.3 – Additionneur 1 bit complet

– Soit deux variables entières x et y sur des domaines finis, elles seront représentées par des vecteurs de bits (de variables booléennes) $(x_1, .., x_n)$ et $(y_1, ..., y_n)$. Alors, l'expression $x+y$ sera codée $a = x+y$ où a est un vecteur de bits, ainsi qu'en utilisant un circuit logique composé d'additionneurs complets et de variables de retenue(voir figure 8.3).

L'introduction de variables « de câblage » amène automatiquement la notion de sous expression. Ainsi l'expression $x = y + (z - u) \vee x = 2$ sera subdivisée en $v_1 = z - u$, $v_2 = y + v_1$ et les variables booléennes $B_2 := x = 2$, $B_1 := x = v_1$, $B_0 := B_1 \vee B_2$.

A la différence des circuits électroniques avec lesquels on peut tenter de minimiser l'énergie ou les temps de transmission, ici nous nous intéressons à faciliter au maximum la tâche des solveurs SAT, car la satisfiabilité d'une formule SAT est en théorie un problème NP-Complet.

L'utilisation de sous formules, sous expressions et de variables auxiliaires a permis d'optimiser la taille de la formule finale en effectuant un partage des sous expressions (à l'aide d'une table de hachage) : par exemple chaque occurrence de la formule précédente sera remplacée par B_0.

De plus, ces variables « câbles », peuvent représenter des circuits très complexes, permettant ainsi d'augmenter les possibilités de propagation de contraintes : la présence de ces variables dans différentes clauses permet une propagation immédiate avant même l'instantiation des variables internes aux circuits.

8.3 Le temps

Les valeurs des horloges augmentent avec le temps, ce qui n'est pas aisé à implémenter directement. A la place, nous introduisons des nouvelles variables réelles $last_x$ pour enregistrer le moment de la dernière remise à zéro de l'horloge x, c'est à dire que si nous avons la transition $s \xrightarrow[x:=0]{t,\tau} s'$, alors l'action $x := 0$ sera

codé comme $last_x := \tau$. Une comparaison d'horloge dans la transition $s \xrightarrow[x \bowtie c]{t,\tau} s'$ avec $\bowtie \in \{\leq, <, =, >, \geq\}$ et c une constante entière sera codée $\tau - last_x \bowtie c$.

Ainsi, il est possible de remplacer une horloge x par la variable correspondante $last_x$ avec les affectations et les conditions décrites ci dessus : dans ce codage pratique, les variables ne sont pas modifiées entre les occurences de transitions. Les variables $last_x$ ont le même pouvoir d'expression que les horloges.

Comme indiqué précédemment, des variables réelles sont utilisées pour manipuler le temps telles que $last_x$ et les timestamps τ. Toutefois, comme analysé dans [PWZ02, Zbr05], il est possible de restreindre les timestamps dans un intervalle borné et une partie décimale fixe (un certain nombre de variables booléennes est affecté pour les bits de la partie entière et pour les bits de la partie fractionnaire), où la taille de la partie entière et la précision de la partie décimale dépendent de la taille de la séquence cherchée (plus précisément, le nombre d'exécutions de transitions).

Alternativement, le codage peut être appliqué à un solveur SMT comme dans [SBM06, ACKS02], où toutes les variables sont codées à l'aide de vecteurs booléens sauf les timestamps codés en utilisant des variables réelles.

8.4 Duplication des variables

Etant donné que la formule finale Φ doit représenter une exécution de longueur K, nous devons ajouter une copie de toutes les variables à chaque étape.

On notera $v^i \in V^i$ avec $1 \leq i \leq K$ la copie de $v \in V$ à l'étape i. Par exemple, si $K = 5$ et $v \in [0..3]$, il faut créer les vecteurs booléens suivants $\{v_1^1, v_2^1\}$, $\{v_1^2, v_2^2\}$, $\{v_1^3, v_2^3\}$, $\{v_1^4, v_2^4\}$, $\{v_1^5, v_2^5\}$.

Le résultat est que l'affectation $x := y+3$ à l'étape k sera codée $x^{k+1} = y^k+3$, c'est à dire qu'une affectation devient une relation entre la valeur x^{k+1} de x après l'étape courante et la valeur y^k de y avant l'étape courante. On notera que cette transition lit y et écrit x.

8.5 Transitions et Multisteps

Chaque exécution d'une transition t d'un multistep MT_k est codée avec une variable booléenne t^k indiquant si la transition est exécutée ou pas. Si la transition $s_k \xrightarrow[cond,action]{(t,\tau)} s_{k+1}$ est exécutée, alors sa condition *cond* est vraie à l'état s_k et au temps τ et l'affectation de l'action est réalisée (*action* est codée sous la forme de contraintes entre les variables en s_k et celles en s_{k+1} comme mentionné précédemment). Formellement la transition est codée sous la forme

$$t^{k+1} \rightarrow cond_t(s_k, \tau) \wedge action_t(s_k, s_{k+1})$$

A ce point, si s_k est déterminé et si l'ensemble des transitions exécutées inclut une transition t qui écrit v, alors $action_t(s_k, s_{k+1})$ détermine la valeur de

v en s_{k+1}. Par contre, si v n'est écrit par aucune transition exécutée, alors sa valeur doit être maintenue. Supposons que l'ensemble des transitions écrivant v est $\{t_a, t_b, t_c\}$, alors ce maintien est codé par la clause

$$t_a^{k+1} \vee t_b^{k+1} \vee t_c^{k+1} \vee v_k = v_{k+1}$$

Comme pour la dépendance, les conditions d'indépendance pour les transitions d'un même multistep peuvent être codées comme une conjonction de contraintes $(\neg t_a^k \vee \neg t_b^k)$ pour les paires de transitions dépendantes (t_a, t_b).

La combinaison des clauses pour les actions, pour les relations de dépendances et celles pour le maintien des valeurs garantit la consistance des états successeurs. En pratique, les contraintes concernant l'écriture et la lecture des variables, ainsi que celles pour les dépendances sont codées ensemble, autorisant un codage plus compact grâce à un partage des sous formules. Par contre, ces clauses de conflits constituent une partie non négligeable de la formule générale.

8.6 Progrès du temps

– Synchrone : toutes les transitions d'un multistep MT_k sont exécutées au même instant τ_i, c'est à dire qu'un seul timestamp est nécessaire pour à chaque étape i. Nous obtenons les contraintes suivantes :

$$\bigwedge_{i=1..K-1} \tau_i \leq \tau_{i+1}$$

– Semi-synchrone : toutes les transitions du multistep MT_k sont exécutées avant un instant ζ_k (une variable additionnelle) et toutes les transitions du multistep MT_{k+1} sont exécutées après ζ_k (voir Figure 7.8). Chaque transition t a son propre timestamp τ_t, ce qui donne les contraintes suivantes :

$$\bigwedge_{\substack{i=1..K-1 \\ t,t' \in \text{trans}}} (\tau_{t^i} \leq \zeta^i) \wedge (\zeta^i \leq \tau_{t'^{i+1}})$$

– Relâché : il n'est pas trivial de coder le progrès relâché du temps car les conditions décrites dans la Section 7.1 ne sont pas locales à deux multisteps adjacents. Une astuce a été utilisée pour rendre les instants d'occurence des multisteps précédents localement accessibles : pour les transitions non exécutées, le timestamp τ_t^k n'a pas de signification. Nous l'utilisons alors pour représenter *l'instant de la dernière exécution* de t avant ou en incluant le multistep courant. Cela amène deux cas : si la transition (t, τ_t) n'est pas exécutée à l'étape i alors la valeur de τ_t doit être maintenue à l'étape $i+1$ et si elle est exécutée nous ajoutons les contraintes $\leq$ avec les timestamps des dernières exécutions des transitions dépendantes :

$$\bigwedge_{k=1..K-1} \bigwedge_{t\in\text{trans}} \neg t^k \rightarrow (\tau_t^k = \tau_t^{k-1}) \wedge$$

$$\bigwedge_{k=1..K-1} \bigwedge_{t_a\in\text{trans}} \bigwedge_{\substack{t_b\in\text{trans} \\ t_a D t_b}} t_a^k \rightarrow (\tau_{t_b}^{k-1} \leq \tau_{t_a}^k)$$

Dans cette formule $t_a \; D \; t_b$ indique que t_a et t_b sont des transitions dépendantes.

8.7 Formule globale

Nous résumons à présent toutes les étapes de la construction de la formule globale Φ qui atteste de l'existence d'une séquence de multistep :

- Allouer des vecteurs booléens $v_1^k, \ldots, v_n^k$ pour tout $v \in V$ et pour tout $1 \leq k \leq K$
- Initialiser Φ avec les affectations initiales (les contraintes) pour chaque variable v^0
- Pour chaque étape $1 \leq k \leq K-1$
  - $\Phi := \Phi \wedge$ codage des transitions
  - $\Phi := \Phi \wedge$ codage des dépendances
  - $\Phi := \Phi \wedge$ maintien des valeurs
  - $\Phi := \Phi \wedge$ codage du progrès du temps
- Ajouter à Φ les contraintes concernant la propriété à vérifier. Pour l'accessibilité, ceci s'obtient en vérifiant si le dernier état satisfait la propriété désirée.

CHAPITRE 9

Les invariants

Un invariant d'état est une condition sur les horloges associée à l'état d'un thread (une valeur du program counter local si l'on se réfère à la terminologie du Chapitre 6) et est intuitivement un « permis de résidence » : la condition $pc_3 = loc_1 \Rightarrow x < 5$ décrit que l'état 1 du thread 3 doit être quitté par une transition avant que l'horloge x n'atteigne 5. Pour éviter cette violation, soit une transition de ce thread quittant cet état doit être exécutée soit une transition d'un autre thread peut remettre à zéro x ce qui effectivement étend le permis de résidence de cet état.

Plus généralement, les invariants d'état sont de la forme $pc_i = loc_k \Rightarrow \bigwedge x_j \leq c_j$. Pour les systèmes avec un seul thread, les invariants d'état ont le même effet qu'ajouter les contraintes sur chaque transition sortante de l'état ; ils n'ajoutent rien à l'expressivité du formalisme. Pour les systèmes parallèles, les invariants impliquent aussi d'ajouter des contraintes sur les transitions sortantes, toutefois ils ont un effet plus global : le système dans son intégralité est forcé d'exécuter des transitions avant l'expiration des invariants de chaque thread. Cette particularité est très utile pour modéliser le couplage de sous-systèmes par le temps, par exemple pour modéliser des timeouts.

Il est possible d'étendre le prototype développé afin d'inclure les invariants, mais techniquement cette intégration dépend de la notion de progrès du temps et peut s'avérer relativement complexe pour le progrès relâché du temps.

9.1 Sémantiques entrelacées et progression du temps synchrone

Soit la séquence temporisée $s_0 \xrightarrow{(t_1,\tau_1)} s_1 \ldots \xrightarrow{(t_n,\tau_n)} s_n = s'$. Pour les sémantiques entrelacées standards, c'est à dire celle pour lesquelles une seule transition peut être exécutée à la fois, la conjonction de tous les invariants d'états (locaux à un thread) à l'état global s_i doit être satisfaite au temps τ_{i+1} (le temps d'exécution de la prochaine transition après s_i). Puisque $\tau_{i+1} \leq \tau_{i+2}$, un invariant valide *après* l'exécution de la transition est encore valide au temps τ_{i+1}.

Pour le progrès synchrone du temps, le même codage est utilisé que pour la sémantique avec entrelacement. Bien que l'on exécute plusieurs transitions au temps τ_{i+1}, on n'exécute aucune transition avant ce moment et les invariants d'état doivent être satisfaits à τ_{i+1}. Puisque des séquences multisteps avec un progrès du temps synchrone ne laissent pas le temps s'écouler entre deux multisteps, il est facile de constater que cette condition est nécessaire et suffisante.

9.2 Progrès du temps semi synchrone

Avec le progrès du temps semi synchrone, un raisonnement similaire à la version synchrone peut aider à comprendre pourquoi est-il suffisant qu'à chaque moment d'exécution τ_{i+1}^k d'une transition dans un multistep, l'invariant soit satisfait.

Ceci peut être efficacement codé en exigeant que l'invariant soit satisfait au temps ζ_{i+1} (la variable de séparation introduite dans la Section 8.6 pour le codage du progrès semi synchrone) : si tous les τ_{i+1}^k du multistep k satisfont l'invariant alors le maximum d'entre eux aussi. Puisque ζ_{i+1} est situé entre les timestamps du multistep MT_{i+1} et ceux de MT_{i+2}, on peut lui attribuer la valeur minimale qui est donc celle du plus grand timestamp de MT_{i+1}. Imposer que ζ_{i+1} respecte l'invariant d'état de s_i est équivalent à l'imposer pour chaque timestamp de MT_{i+1}.

9.3 Progrès du temps relâché

Pour le progrès du temps relâché, il est possible d'utiliser une technique développée pour manipuler les invariants dans le contexte de l'exploration d'états avec zones et avec des sémantiques d'ordre partiel (voir [NQ06a]). Dans cette approche, il faut distinguer la vue locale et globale des invariants : localement, les transitions sortantes d'un état doivent satisfaire l'invariant. Globalement, une transition mettant à zéro une horloge doit satisfaire tous les invariants des états courants des autres threads manipulant cette horloge. Finalement, l'état final doit satisfaire l'invariant global. Ces trois types de contraintes ne sont pas complexes à coder, la condition pour le progrès du temps relâché est-elle même plus compliquée.

CHAPITRE 10

Implémentation et résultats

10.1 Implémentation de POEM

POEM (Partial Order Environment of Marseille) est un outil de model checking modulaire construit pour supporter plusieurs langages d'entrée, plusieurs algorithmes d'analyse et de résolution. Le code de POEM est réutilisable pour implémenter de puissants algorithmes de model checking basés sur les ordres partiels [KNQV06, LNZ05, MN10].

POEM se compose d'un ensemble de modules interconnectés comme dans la Figure 10.1. Dans un premier temps un modèle et une propriété sont lues par un *frontend* et représentés par la *global data structure* GDS jouant un rôle d'interface entre les trois sous-systèmes.

Le *core* analyse et transforme la structure GDS en réalisation des vérifications de types, des propagations de constantes, une transformation du modèle, la génération des transitions incluant l'analyse des dépendances pour les algo-

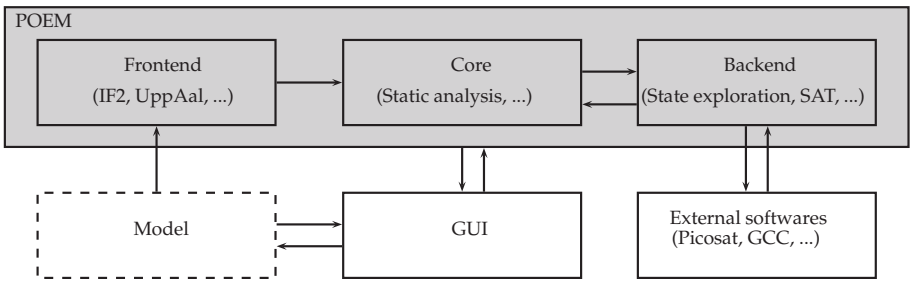


FIGURE 10.1 – Structure de POEM

rithmes d'ordre partiel, etc.

Le résultat de cette analyse est passée au *backend* qui implémente les algorithmes de résolutions permettant de vérifier la propriété donnée en entrée. Le core transforme alors la preuve (la trace) et l'écrit dans un fichier XML lisible par une interface graphique (GUI) permettant de visualiser l'exécution trouvée avec un lien vers le code source correspondant.

Le GUI (écrit en Java) comporte les fonctionnalités suivantes :

- Edition des fichiers de modèles avec une coloration syntaxique,
- Configuration de POEM
- Affichage convivial des résultats et des messages d'erreur
- Plusieurs représentations des exécutions temporisées : une ligne en temps réel, une représentation par ordres partiels des évènements.

L'exécutable de POEM est écrit en OCAML, un langage à la fois fonctionnel et impératif. Les algorithmes nécessitant des performances critiques sont déléguées à un programme externe écrit en C.

Les trois parties de POEM (le frontend, le core et le backend) sont indépendantes et liées par GDS (aisi que par quelques tables additionnelles), ce qui signifie que l'ajout d'un nouveau frontend, d'un nouveau backend ou d'un nouvel algorithme dans le core a un impact limité sur le reste du programme. En réalité, les frontends et les backends sont des plugins pour le core et peuvent être ajoutés ou enlevés sans toucher une seule ligne de code partagé. Les plugins peuvent déclarer leurs propres options de configuration (pour des variantes d'algorithmes par exemple), ce qui pour l'exécutable se matérialise par des options au niveau de la ligne de commande et dans le GUI sous forme de boites de dialogues présentant les options.

Actuellement, nous avons implémenté des frontends pour lire des modèles écrits en IF2 [BGM02] et en XTA, le langage de Uppaal [BDL+06]. Une grande partie des spécifications des deux langages est couverte, seuls certains concepts spécialisés ne sont pas présents et peuvent être ajoutés si nécessaire. Deux backends ont été implémentés :

- Un backend d'exploration d'états générant du code C pour l'analyse d'une manière similaire à Spin [Hol03] avec des réductions d'ordre partiel pour les systèmes non temporisés [KNQV06] et des sémantiques d'ordre partiel pour les automates temporisés [LNZ05]
- Un backend SAT générant des clauses et délégant leur solution à un solveur SAT pour systèmes temporisés et non temporisés avec une optimisation multistep [MN10] permettant de réaliser plusieurs transitions en parallèle abouttissante à des gains de performance substantiels

Une attention toute particulière a été apportée à la représentation des transitions basée sur le modèle lecture/écriture. La représentation est optimisée pour permettre une utilisation efficace pour les deux backends actuellement implémentés.

10.1.1 Sémantiques d'ordre partiel dans POEM

La représentation intermédiaire GDS met l'accent sur la lecture et l'écriture des variables ce qui est le cœur de l'analyse de dépendance basée sur la théorie des traces de Mazurkiewicz [DR95] : une transition qui lit une certaine variable x est dépdendante d'une autre qui écrit x. L'idée des optimisations sur les ordres partiels est basée sur le fait que des transitions indépendantes peuvent être exécutée dans n'importe quel ordre (ou bien en parallèle), aboutissant au même état.

Pour chaque transition, l'ensemble des variables lues ou écrites est accessible soit simplement de manière statique, soit dynamiquement offrant une meilleur réduction, en fonction du backend. Les redondances dans la représentation des dépendances lecture/écriture (groupes de variables lues et écrites par la même transition) peuvent aussi être eliminées.

GDS permettant la représentation de transitions complexes (ayant des effets de bord décrits par des petits programmes impératifs) ne pouvant pas être facilement analysées par touts les backends, un moteur paramétré de réécriture permet de réaliser les simplifications nécessaires pour le backend tout en préservant la séparation des modules autant que possible.

Le backend Explore inspiré par la génération de code C de Spin, accède aux dépendances à l'aide de tables et et de fonctions générées par l'analyse du core. Une structure de données pour la manipulation des traces de Mazurkiewicz [NQ06b] est disponible pour les différents algorithmes. Dans le passé, deux algorithmes différents basés sur les ordres partiels ont été ajoutés à Explore, un pour les systèmes non temporisés [KNQV06] et l'autre pour les systèmes temporisés [LNZ05].

La backend POEM [MN10] exploite l'indépendance permettant une compression de plusieurs transitions indépendantes en un seul multistep, réduisant la longueur des exécutions et augmentant la performance de l'approche SAT pour les systèmes asynchrones.

10.1.2 Presentation du GUI

Le GUI a été développé en Java avec les contraintes suivantes :

- La communication avec POEM est réalisée à l'aide de fichiers XML
- Les options du GUI sont lues et écrites depuis et vers des fichiers XML, permettant d'ajouter ou d'enlever certaines d'entres elles sans avoir besoin de modifier le code JAVA
- Chaque frontend est décrit avec son ensemble d'options
- Ajouter un nouveau backend est facile : il suffit juste d'ajouter quelques lignes de code et la liste des mots clés pour la coloration syntaxique

Dans les figures suivantes nous allons voir quelques usages typiques de POEM :

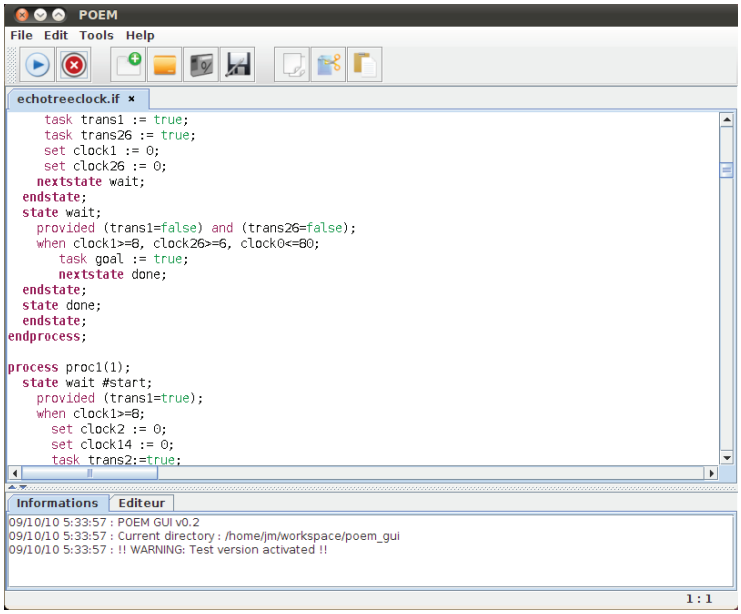


FIGURE 10.2 – LE GUI avec un modèle prêt à être édité

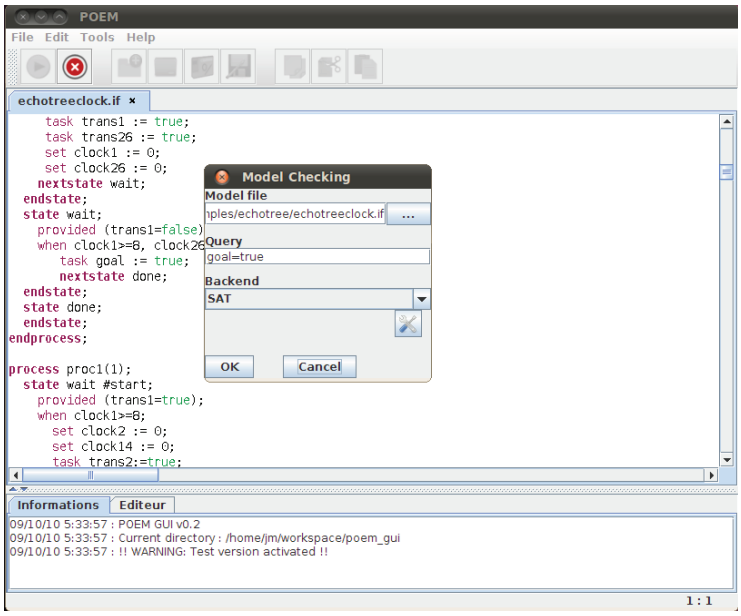


FIGURE 10.3 – Début du model checking pour le modèle du protocole

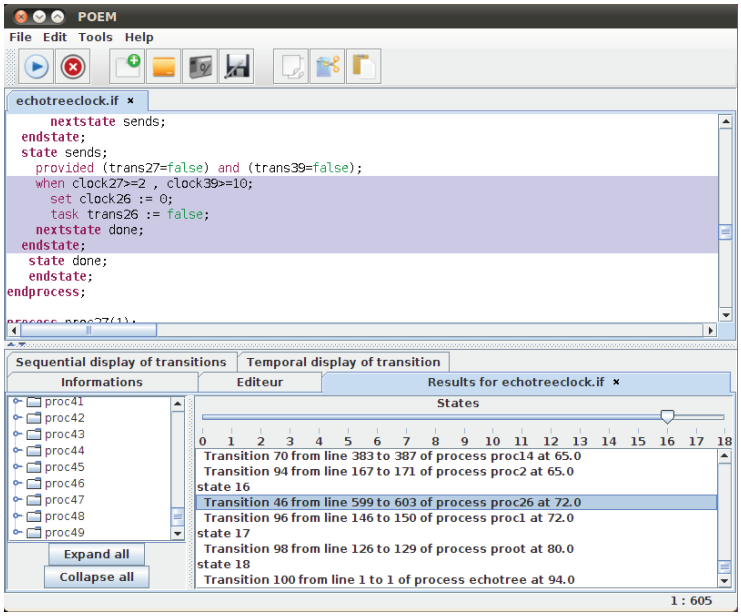


FIGURE 10.4 – Le résultat du model checking

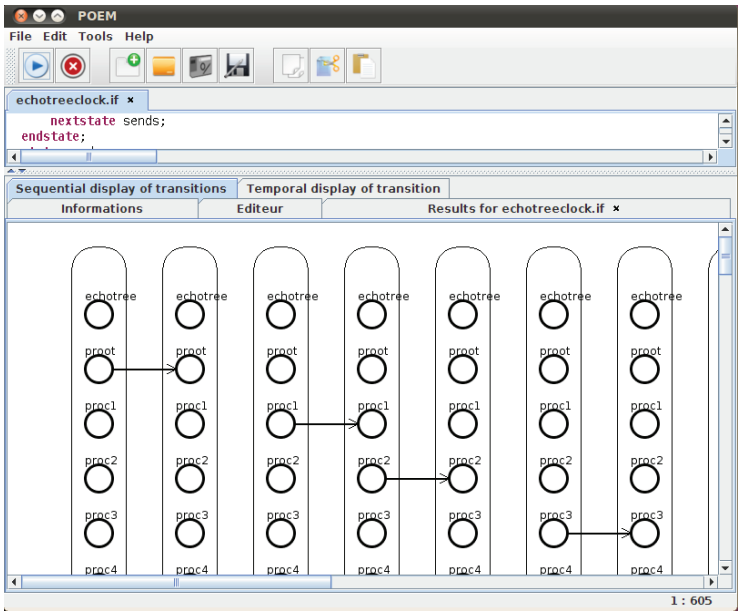


FIGURE 10.5 – Une représentation des transitions temporisées exécutées

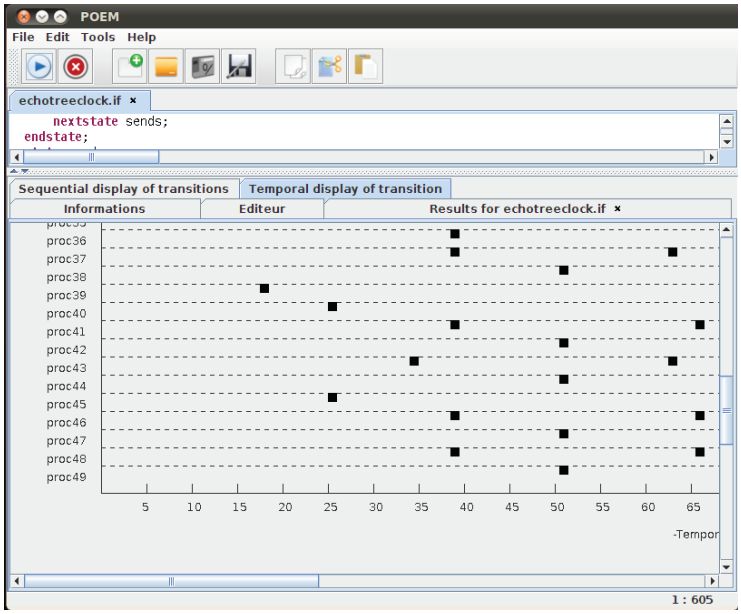


FIGURE 10.6 – Une autre représentation temporisée de l'exécution

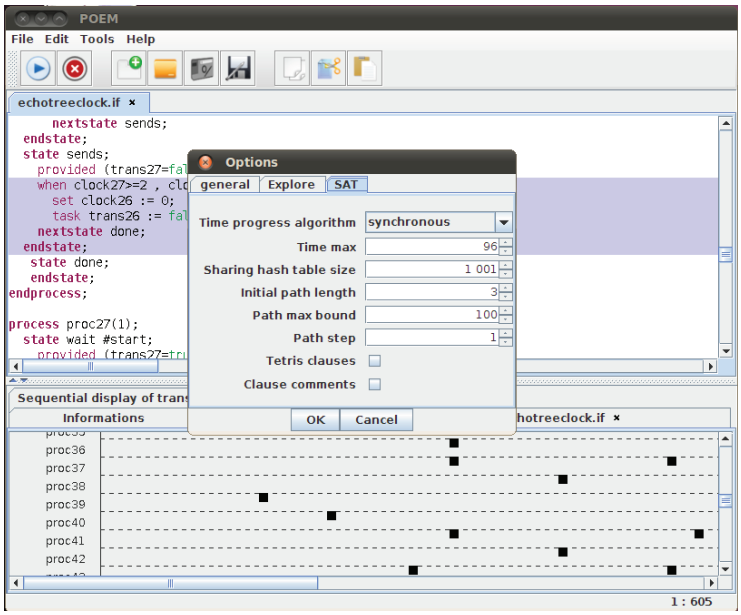


FIGURE 10.7 – Quelques options

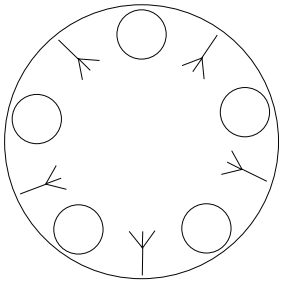


FIGURE 10.8 – Le dîner des philosophes

10.2 Expérimentations

Dans cette section nous allons présenter des résultats expérimentaux : dans une première partie nous allons comparer nos différentes approches sur des modèles non temporisés en utilisant Spin [Hol03] comme référence et dans une seconde partie nous utiliserons des modèles temporisés avec UppAal [BDL+01] comme référence. Les tests ont été effectués sur un Mac Pro quad-core 2.66 Ghz, avec 16Go de mémoire. Toutefois nous n'avons pas exploité les possibilités multi thread de cette plate-forme. Tous les résultats sont exprimés en secondes, sauf lorsque m apparait pour minutes. *Err* signifie qu'une erreur a été rencontrée avec Spin (généralement lorsque plus de 256 processus sont présents dans le modèle) et un tiret _ signifie qu'aucune solution n'a été trouvée en *20 minutes*. Enfin les résultats pour les différents algorithmes de POEM sont de la forme `T1/T2/NB(STEP)` où T1 est le temps total pour trouver une solution, T2 est le temps uniquement pris par le SAT solver, NB est le nombre de clauses générées et STEP est le nombre de pas (la borne) pour aboutir à la solution.

10.2.1 Le dîner des philosophes

Dans ce problème introduit par Edsger Dijkstra en 1971, des philosophes sont assis autour d'une table. Entre chaque philosophe est disposée une fourchette. Un philosophe peut être dans plusieurs états : soit il pense, soit il a faim, soit il mange. Pour manger, chaque philosophe a besoin de deux fourchettes : celle située à sa droite et l'autre à sa gauche, empêchant ainsi les philosophes à ses côtés de manger. Dés lors, plusieurs problèmes peuvent se poser comme par exemple :

- Comment faire en sorte que chaque philosophe finisse par manger ?
- Existe-t-il une situation où plus aucun philosophe ne peut manger ?

Nous allons nous intéresser à cette seconde question qui est relativement triviale puisqu'il suffit que tous les philosophes prennent en même temps la fourchette de droite pour obtenir un interblocage. La table 10.1 présente les résultats pour une plage de 5 à 1000 philosophes.

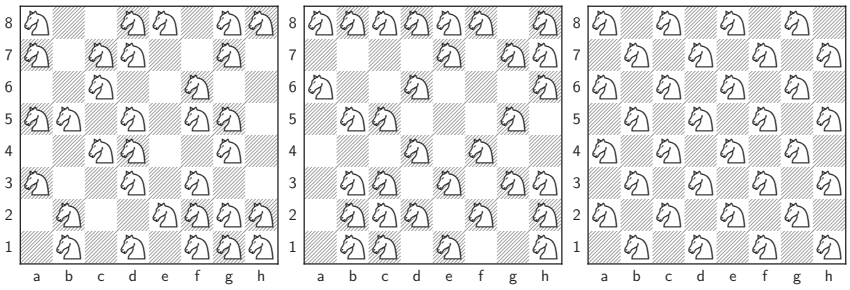


FIGURE 10.9 – Solution depuis une position aléatoire en 2 étapes multi-step

Nb	Spin	Interleaved	Multi step
5	2.8	0.2/0.1/4K(5)	0.1/0/2K(1)
10	13.1	0.4/0.2/14K(10)	0.1/0.1/5K(1)
14	2.9	2.4/2.1/28K(14)	0.2/0.1/6K(1)
15	-	1.5/1.2/32K(15)	0.2/0.1/7K(1)
18	-	6.5/6.0/45K(18)	0.2/0.1/8K(1)
19	3.1	8.6/8.1/51K(19)	0.2/0.1/9K(1)
20	-	22.4/21.9/56K(20)	0.2/0.1/9K(1)
29	3.1	14 :35/14 :35/116K(29)	0.4/0.1/14K(1)
99	3.2	-	2.3/0.6/55K(1)
100	-	-	2.7/0.6/55K(1)
1000	Err	-	3 :56/18.0/1.4M(1)

TABLE 10.1 – Les résultats pour le dîner des philosophes

Spin trouve des solutions à l'aide d'une recherche en profondeur pour de grandes instances uniquement par chance : en générant des modèles dans lesquels l'ordre des transitions est aléatoire, on constate que Spin ne parvient pas à résoudre des instances de plus de 9 philosophes.

10.2.2 Le problème des cavaliers

Ce problème est une variante du célèbre problème des 8 dames : un certain nombre de cavaliers doivent être placés sur un échiquier de telle sorte qu'ils ne puissent pas se menacer conformément aux règles des échecs.

Le problème a été transformé en un problème d'accessibilité en disposant aléatoirement (ou bien dans un coin) un nombre n de cavaliers sur l'échiquier, puis de demander aux algorithmes de trouver une succession de déplacements, conformes aux règles des échecs, aboutissant à un état dans lequel plus aucun cavalier ne menace l'autre. Dans la figure 10.9 on peut voir de la gauche vers la droite un échiquier dans sa position initiale, l'échiquier après une étape multi-step et enfin l'état final. Nous avons construit un simple modèle dans lequel chaque case de l'échiquier est gérée par un processus qui peut, si elle contient

un cavalier, décider de le déplacer sur une autre case libre et autorisée. Une transition de contrôle est utilisée pour vérifier si tous les cavaliers sont dans une situation de "sécurité".

Nous avons effectué des tests pour différentes tailles d'échiquiers, différentes quantités de cavaliers et pour différentes positions initiales aléatoires ou dans un coin de l'échiquier.

En fonction de la taille du problème initial, plusieurs pas multistep sont nécessaires pour atteindre un état sûr. Un haut degré de parallélisme est utilisé pour parvenir à une solution : Spin et Interleave sont seulement capables de résoudre de petites instances alors que multistep peut aller beaucoup plus loin. A titre indicatif, le plus grand exemple a un espace d'états de l'ordre de 10^{75} et l'obtention d'une solution nécessite 257 déplacements individuels de cavaliers.

Board	Knights	Spin	Interleave	Multi step
4x4	8	0.3	0.2/0.1/6K(5)	0.1/0.1/5K (2)
8x8	16	-	5.6/4.3/64K(9)	2.3/0.6/65K(2)
8x8	24	-	-	2.8/0.9/86K(3)
8x8	32	-	-	2.9/0.8/86K(3)
16x16	128	Err	-	1 :04/7.9/749K(5)

TABLE 10.2 – Résultats du problème des cavaliers positionnés sur la partie haute de l'échiquier

Board	Knights	Spin	Interleave	Multi step
4x4	8	3.2	18.0/3.8/28K(17)	0.2/0.1/7K(3)
8x8	8	3.2	5.9/2.0/80K(7)	2.2/0.6/65K(2)
8x8	16	-	-	2.3/0.6/65K(2)
8x8	24	-	-	2.9/0.9/86K(3)
8x8	32	-	-	2.9/0.9/86K(3)
16x16	128	Err	-	47.8/5.2/500K(3)

TABLE 10.3 – Résultats du problème des cavaliers positionnés aléatoirement sur l'échiquier

10.2.3 Protocoles réseau

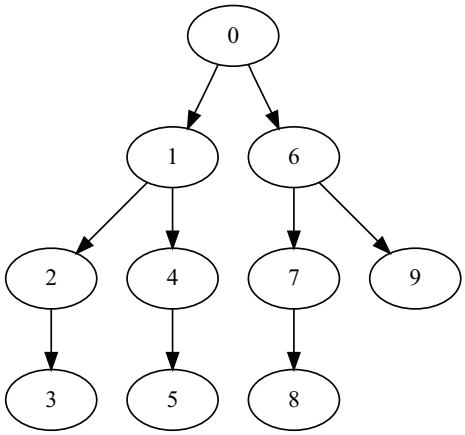


FIGURE 10.10 – Réseau comportant 10 noeuds

Nous allons à présent considérer un protocole réseau simpliste : les éléments du réseau sont connectés sous la forme d'un arbre binaire *aussi équilibré que possible*.

Les transmissions s'effectuent selon les règles suivantes :

- La racine commence à transmettre à ses fils
- Un noeud transmet à ses descendants ssi il a reçu un signal de son parent
- Une feuille recevant un signal, transmet un acquittement à son parent
- Un noeud recevant l'acquittement de ses enfants en envoie un à son parent
- Le protocole s'arrête ssi la racine reçoit les acquittements de ses enfants

En résumé, les transmissions s'effectuent de la racine vers les feuilles, puis des feuilles vers la racine.

Version non temporisée

Nous allons tout d'abord examiner une version non temporisée du réseau dans laquelle chaque transmission s'effectue sans délai. Nous demandons alors au model checker de chercher une terminaison correcte du protocole. On constatera que ce protocole est confluent et Spin avec une recherche en profondeur (DFS) trouve un chemin en temps constant pour toutes les instances inférieures à 256 noeuds (il est en erreur au delà). Toutefois, nous avons pu effectuer une comparaison entre nos différentes approches Interleave et Multistep, et aussi résoudre avec ces deux derniers des réseaux comportant 1000 noeuds.

Nodes	Spin	Interleave	Multistep
5	3.2	0.1/0/2K(9)	0.1/0/1K(6)
15	3.3	0.6/0.3/25K(29)	0.2/0.1/6K(8)
30	3.3	2.8/1.5/100K(59)	0.4/0.2/16K(10)
50	3.3	12.4/7.2/280K(99)	1.0/0.4/33K(12)
100	3.3	1 :29/43.1/1.1M(199)	3.4/1.0/83K(14)
200	3.3	-	13.3/2.8/215K16)
1000	Err	-	12 :26/32.3/2.4M(20)

TABLE 10.4 – Résultats du protocole réseau non temporisé

Version temporisée

Nous avons voulu étendre le protocole précédent en ajoutant un délai aléatoire de transmission entre un noeud et ses enfants, puis vérifier s'il existe une terminaison correcte pour un délai optimal. Dans la table 10.5 on constate que le codage *semi synchrone* permet d'obtenir des solutions significativement plus courtes que les autres algorithmes. Parfois le progrès du temps relâché parvient à trouver un chemin plus court ou identique mais au prix d'un temps de calcul plus long dû à l'accroissement du nombre de clauses.

nodes	UppAal	entrelacement	Multi step		
			synchrone	semi synchrone	relâché
5	0.0	0/0/11K(9)	0.1/0.1/7K(6)	0.2/0.2/19K(6)	0.3/0.2/29K(6)
10	0.1	2.9/2.9/49K(19)	0.9/0.8/31K (12)	0.8/0.3/59K(8)	1.1/0.8/95K(8)
15	20.0	19.0/19.0/110K (29)	0.8/0.5/40K (10)	1.3/0.5/90K(10)	1.7/1.1/144K(8)
20	-	4 :48/4 :46/196K(39)	4.6/3.5/93K (18)	4.5/3.5/175K (12)	3.2/2.3/240K(10)
50	-	-	21.6/18.2/292K (20)	13.3/5.6/508K(12)	18.5/8.2/846K(12)
100	-	-	11 :21/11 :06/812K (28)	1 :21/19.2/1.2M(14)	1 :36/34.1/2M(14)
200	-	-	-	4 :33/3 :09/2.7M(18)	14 :21/2 :15/4.6M(16)

TABLE 10.5 – Résultats du protocole réseau temporisé

10.3 Analyse de circuits temporisés

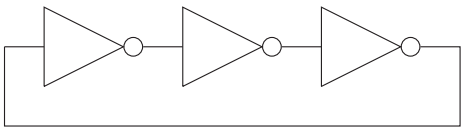


FIGURE 10.11 – Un circuit avec des portes NOT

L'analyse du comportement de circuits numériques est un domaine très important de la vérification. Nous introduisons ici un problème d'analyse d'un circuit composé de portes logiques NOT : plusieurs portes NOT sont connectées

comme dans la figure 10.11. Chaque porte dispose d'un délai entre la réception et l'émission d'un signal. Initialement toutes les valeurs sont égales à zéro et l'on souhaite savoir si le circuit peut se stabiliser, c'est-à-dire s'il existe un moment t à partir duquel les valeurs cessent définitivement de changer. Bien entendu, une stabilisation n'est possible qu'avec un nombre pair de portes et il est possible de réaliser cette stabilisation avec un seul pas multistep. On constate dans la table 10.6 que les algorithmes multistep les plus simples permettent de trouver un résultat plus rapidement, les plus complexes n'apportant aucun avantage ici.

nodes	UppAal	entrelacement	Multistep		
			synchrone	semi synchrone	relâché
4	0	0.2/0/12K(4)	0.2/0/8K(3)	0.3/0/23K(3)	0.6/0/53K(3)
10	0.1	0.8/0.3/48K(7)	0.4/0/22K(3)	0.9/0.1/58K(3)	2.1/0.9/139K(3)
16	-	2.5/1.7/112K(10)	0.9/0.5/38K(3)	1.9/1.6/99K(3)	3.4/1.2/242K(3)
20	-	3.6/2.4/164K(12)	1.3/0.8/48K(3)	2.3/1.8/123K(3)	4.0/1.3/301K(3)
50	-	-	4.3/1.2/129K(3)	6.9/2.3/327K(3)	12.9/2.8/811K(3)
100	-	-	27.4/3.1/279K(3)	37.2/3.9/704K(3)	57.3/5.6/1.7M(3)
200	-	-	5 :54/4.3/608K(3)	7 :11/6.1/1.5M(3)	8 :55/11.1/3.8M(3)

TABLE 10.6 – Résultats pour l'analyse de circuits temporisés

Conclusions et perspectives

Synthèse de contrôleur pour systèmes hybrides Nous avons introduit une nouvelle méthode pour la synthèse de contrôleurs, l'approche *hiérarchique* permettant de diminuer drastiquement l'espace d'états à explorer pour des systèmes dynamiques comportant plusieurs variables. Il a été ainsi possible d'explorer des systèmes impossible à calculer directement. Une autre particularité de notre approche a été de s'intéresser à des objectifs Stabilize : trouver un chemin vers une zone cible, sans jamais emprunter d'états non autorisés, et y rester. Nous avons par la suite étendu cette notion afin de permettre à la zone cible de ne plus être constante dans le temps, synthétisant ainsi un contrôleur pour des systèmes réels comme un drone. Toutefois, pour chaque système il a été nécessaire de déterminer manuellement les paramètres de chaque variable : les bornes (même si le système les impose plus ou moins) ainsi que la discrétisation. Ce dernier point est critique car le comportement d'un système change selon que la discrétisation est fine (beaucoup d'états discrets) ou grossière (peu d'états discrets). Avec une discrétisation grossière, chaque état discret englobe un nombre important de points réels, par conséquent le système peut rester « bloqué » dans un même état sans pouvoir en sortir. Augmenter la discrétisation signifie augmenter le nombre d'états à explorer et donc les ressources nécessaires. Il nous a fallu de nombreux essais avant de trouver des paramètres optimaux c'est à dire les moins élevés possibles permettant d'obtenir un contrôleur. Une voie à explorer serait d'automatiser cette paramétrisation en utilisant par exemple une approche itérative : commencer avec une discrétisation grossière et raffiner à chaque essai infructueux.

Nous aurions souhaité aussi approfondir nos recherches dans la voie de la compositionnalité : pouvoir « connecter » entre eux différents contrôleurs, la sorie de l'un devenant l'entrée de l'autre. Il serait ainsi possible de synthétiser des systèmes comportant de nombreuses variables tout en gardant une complexité réduite pour chaque sous système.

Une autre approche non explorée est la synthèse à l'aide d'un « oracle » qui nous donnerait les actions à réaliser à partir d'un état donné. Cet oracle aura au préalable calculé un chemin lui permettant d'atteindre l'objectif et d'y rester, mais il n'existe aucune garantie sur la qualité de sa « stratégie » car il peut

exister des cas où il est possible d'aboutir dans des états interdits. Le rôle du système serait alors de vérifier et de corriger les erreurs de cet oracle. L'intérêt serait de profiter de l'existence de ce dernier pour accélérer les calculs. Il existe plusieurs méthodes pour obtenir un oracle comme par exemple l'apprentissage par renforcement.

Model checking Nous avons introduit trois méthodes basées sur les ordres partiels avec une réduction SAT permettant la vérification de systèmes temporisés multithreadés. Ces trois méthodes : synchrone, semi-synchrone et relâchée utilisent les propriétés de dépendances existantes entre les variables lues et celles écrites. Ainsi si une même variable est seulement lue par deux processus distincts, alors il est possible d'exécuter les transitions associées *en parallèle* réduisant ainsi le nombre de pas à réaliser pour aboutir à une solution. Ces transitions exécutées en parallèle ont été appelées *multisteps* et les méthodes évoquées précédemment permettent d'apporter plus ou moins de souplesse dans la répartition des moments d'exécutions des transitions d'un même multistep voire entre deux multisteps. Ces techniques sont utilisées dans le cadre du *Bounded Model Checking* introduit par [Hel01], c'est à dire que les chemins des solutions sont de taille bornée. Un des problèmes rencontré a été la longueur n que la solution doit avoir : il faut l'anticiper avant le lancement des calculs. Deux cas peuvent se produire : une solution est trouvée et le chemin peut avoir une taille égale ou inférieure à n. Dans le cas contraire, il n'est pas possible de savoir si la longueur n est trop petite pour trouver une solution ou bien si effectivement il n'existe pas de solution à ce problème. Or plus la longueur n est importante et plus la taille de la formule générée pour le solveur SAT est importante, augmentant ainsi les temps de calcul. Une solution serait de procéder itérativement c'est à dire d'augmenter la taille du chemin recherché à chaque réponse négative. Il existerait aussi un autre intérêt pour une telle approche : la possibilité de réutiliser les résultats obtenus pour la recherche de taille $n-1$ lors du calcul d'un chemin de taille n. Ce codage utilise en grande partie les progrès réalisés dans le domaine des SAT solveurs mais il serait aussi envisageable d'utiliser un codeur SMT pour Satisfiability Modulo Theories comme dans [SBM06, ACKS02]. Les formules à générer seraient alors des formules de la logique du premier ordre. Une partie non négligeable du temps de recherche a consisté à implémenter les méthodes théoriques décrites précédemment. Bien que cela puisse paraître ingrat et considéré comme une perte de temps, nous avons pu mettre en lumière les forces et faiblesses de notre approche. Nous avons utilisé la plate-forme d'expérimentation POEM ([ZYN03]) successeur de la plate-forme ELSE utilisée dans [Zen04]. Le code étant modulable, nous avons pu ajouter les éléments nécessaires en évitant de toucher au maximum au code existant. En entrée, un sous ensemble du langage IF2 ([BGM02]) a été implémenté ainsi que des parties d'autres langages, en particulier le langage implémenté dans Uppal [BDL$^+$06]. En raison de la forte renommée de ce logiciel, il serait intéressant de finir l'implémentation de ce langage. L'une des problématiques étant la présence de parties impératives (style code C) qui n'est pas pris en charge par POEM. L'autre langage à implémenter

serait Promela utilisé en particulier dans Spin [Hol03]. La difficulté de ce langage est qu'il est formé de nombreux blocs comme en C, bien que cela ne soit pas insurmontable. Une autre partie à améliorer est l'interface graphique : rajouter plus de fonctionnalités, rendre plus souple la saisie des modèles, améliorer leur affichage, etc.

Bibliographie

[ABH+97] R Alur, R Brayton, T Henzinger, S Qadeer, and S Rajamani. Partial order reduction in symbolic state space exploration. In *In Proc. 9th Int. Conf. on Computer Aided Verification*, 1997.

[ACKS02] G Audemard, A Cimatti, A Kornilowicz, and R Sebastiani. Sat-based bounded model checking for timed systems. In *In Proc. FORTE, volume 2529 of LNCS*. Springer, 2002.

[ADG07] Eugene Asarin, Thao Dang, and Antoine Girard. Hybridization methods for the analysis of nonlinear systems. *Acta Inf.*, 43(7) :451–476, 2007.

[Alu92] Rajeev Alur. *Techniques for automatic verification of real-time systems*. PhD thesis, Stanford, CA, USA, 1992.

[Alu98] Rajeev Alur. Timed automata. In *NATO-ASI 1998 Summer School on Verification of Digital and Hybrid Systems*, 1998.

[AT02] Karine Altisen and Stavros Tripakis. Tools for controller synthesis of timed systems. In Paul Pettersson and Wang Yi, editors, *Proceedings of the 2nd Workshop on Real-Time Tools (RT-TOOLS'02)*, August 2002.

[BCC99] Armin Biere, Alessandro Cimatti, and Edmund Clarke. Symbolic model checking without bdds. pages 193–207. Springer-Verlag, 1999.

[BCM+92] J R Burch, E M Clarke, K L McMillan, D L Dill, and L J Hwang. Symbolic model checking : 10 20 states and beyond. *Information and Computation*, (98), 1992.

[BDL+01] G. Behrmann, A. David, K. G. Larsen, O. Moeller, P. Pettersson, and W. Yi. UPPAAL - present and future. In *Proc. of 40*th *IEEE Conference on Decision and Control*. IEEE Computer Society Press, 2001.

[BDL+06] G. Behrmann, A. David, K.G. Larsen, J. Hakansson, P. Petterson, W. Yi, and M. Hendriks. Uppaal 4.0. In *3rd international conference on the Quantitative Evaluation of Systems QEST*, pages 125–126, Washington, DC, USA, 2006.

[BGM02] M Bozga, S Graf, and L Mounier. If-2.0 : A validation environment for component-based real-time systems. In *Computer Aided Verification, LNCS 2404*. Springer, 2002.

[Bie08] A. Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4 :75–97, 2008.

[BS06] J. Brandfield and C. Stirling. *The Handbook of Modal Logic*, chapter Modal mu-calculi, pages 721–756. Elsevier, 2006.

[CDF+05] 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. CONCUR'2005*, volume 3653 of *LNCS*, pages 66–80. Springer, 2005.

[CHP71] P. J. Courtois, F. Heymans, and D. L. Parnas. Concurrent control with "readers" and "writers". *Commun. ACM*, 14(10) :667–668, 1971.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71 : Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.

[CS93] R. Cleaveland and B. Steffen. A linear-time model-checking algorithm for the alternation-free modal mu-calculus. In *FMSD*, pages 48–58. Springer-Verlag, 1993.

[DM07] Alexandre Donzé and Oded Maler. Systematic simulation using sensitivity analysis. In *Proc. HSCC'07*, volume 4416 of *LNCS*, pages 174–189. Springer, 2007.

[DR95] V. Diekert and G. Rozenberg, editors. *The Book of Traces*. World Scientific, 1995.

[Gim06] Hugo Gimbert. *Jeux positionnels*. Thèse de doctorat, Lab. Informatique Algorithmique : Fondements et Applications, Université Paris 7, France, December 2006.

[GLT80] H.J. Genrich, K. Lautenbach, and P. S. Thiagarajan. Elements of general net theory. In *Proceedings of the Advanced Course on General Net Theory of Processes and Systems*, pages 21–163, London, UK, 1980.

[GTW02] E. Grädel, W. Thomas, and Th. Wilke, editors. *Automata, Logics, and Infinite Games : A Guide to Current Research*, volume 2500 of *LNCS*. Springer, 2002.

[Hel01] K Heljanko. Bounded reachability checking with process semantics. In *In Proceedings of the 12th International Conference on*

Concurrency Theory (Concur 2001), pages 218–232. Springer-Verlag, 2001.

[HHMWT00] Th. A. Henzinger, B. Horowitz, R. Majumdar, and H. Wong-Toi. Beyond hytech : Hybrid systems analysis using interval numerical methods. In *Proc. HSCC'00*, volume 1790 of *LNCS*, pages 130–144. Springer, 2000.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12 :576–580, 1969.

[Hol03] Gerard J. Holzmann. *The SPIN Model Checker : Primer and Reference Manual*. Addison-Wesley Professional, 2003.

[JN02] T Jussila and Niemela. I. : Parallel program verification using bmc. In *In : ECAI 2002 Workshop on Model Checking and Artificial Intelligence*, pages 59–66, 2002.

[KNQV06] Marcos E. Kurbán, Peter Niebert, Hongyang Qu, and Walter Vogler. Stronger reduction criteria for local first search. In *Theoretical Aspects of Computing - ICTAC*, volume 4281 of *LNCS*, pages 108–122, 2006.

[LNZ04] D. Lugiez, P. Niebert, and S. Zennou. A partial order semantics approach to the clock explosion problem of timed automata. In *Tools and Algorithms for the Construction and Analysis of Systems : 10th International Conference, TACAS 2004*, volume 2988 of *Lecture Notes in Computer Science*, pages 296–311. Springer Verlag, 2004.

[LNZ05] D. Lugiez, P. Niebert, and S. Zennou. A partial order semantics approach to the clock explosion problem of timed automata. *Theoretical Computer Science*, 345(1) :27–59, 2005.

[LS98] X. Liu and S. A. Smolka. Simple linear-time algorithms for minimal fixed points. In *Proc. ICALP'98*, volume 1443 of *LNCS*, pages 53–66. Springer, 1998.

[Lyg04] John Lygeros. Lecture notes on hybrid systems. Technical report, 2004.

[MN10] Janusz Malinowski and Peter Niebert. SAT based bounded model checking with partial order semantics for timed automata. In *TACAS 2010 – 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 6015 of *Lecture Notes in Computer Science*, pages 405–419. Springer, 2010.

[MNR11] Janusz Malinowski, Peter Niebert, and Pierre-Alain Reynier. A hierarchical approach for the synthesis of stabilizing controllers for hybrid systems. In *Proceedings of the 9th international conference on Automated technology for verification and analysis*, ATVA'11, pages 198–212, Berlin, Heidelberg, 2011. Springer-Verlag.

[NMA+02] P. Niebert, M. Mahfoudh, E. Asarin, M. Bozga, N. Jain, and O. Maler. Verification of timed automata via satisfiability checking. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 2469 of *Lecture Notes in Computer Science*, pages 225–244, 2002.

[NQ06a] P. Niebert and H. Qu. Adding invariants to event zone automata. In *Formal Modelling and Analysis of Timed Systems*, pages 290–305. LNCS 4202, 2006.

[NQ06b] P. Niebert and H. Qu. The implementation of mazurkiewicz traces in poem. In *Automated Technology for Verification and Analysis, 4th International Symposium, ATVA*, pages 508–522, 2006.

[Pel93] D. Peled. All from one, one for all : On model checking using representatives. In *International Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423, 1993.

[PWZ02] W Penczek, B Wozna, and Zbrzezny. A. : Towards bounded model checking for the universal fragment of tctl. In *Proc. of the 7th Int. Symp. on Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT'02), LNCS 2469*, pages 265–288. Springer-Verlag, 2002.

[SBM06] Ramzi Ben Salah, Marius Bozga, and Oded Maler. On interleaving in timed automata. In *CONCUR 2006 - Concurrency Theory, 17th International Conference, CONCUR 2006, Bonn, Germany, August 27-30, 2006, Proceedings*, volume 4137 of *Lecture Notes in Computer Science*, pages 465–476. Springer, 2006.

[SS90] G. Stalmarck and M. Saflund. Modelling and verifying systems and software in propositional logic. In BK Daniels, editor, *Safety of Computer Control Systems 1990 (SAFECOMP'90)*. Pergamon Press, Oxford, 1990.

[Val90] A. Valmari. A stubborn attack on state explosion. In *Proceedings of the 2nd International Workshop on Computer Aided Verification*, volume 531 of *Lecture Notes in Computer Science*, pages 156–165. Springer, 1990.

[Zbr05] A. Zbrzezny. Sat-based reachability checking for timed automata with diagonal constraints. *Fundam. Inf.*, 67(1-3) :303–322, 2005.

[Zen04] Sarah Zennou. *Méthodes d'ordre partiel pour la vérification de systèmes concurrents et temps réel*. PhD thesis, Université Aix-Marseille I, 2004.

[ZYN03] S. Zennou, M. Yguel, and P. Niebert. Else : A new symbolic state generator for timed automata. In *Proceedings of the 1st International Conference FORMATS*, volume 2791 of *Lecture Notes in Computer Science*, pages 273–280. Springer, 2003.