Langage pour la vérification de modèles par contraintes

Pierre Talbot* **Clément Poncelet**

Institut de Recherche et Coordination Acoustique/Musique (IRCAM)
Université Pierre et Marie Curie, Paris, France
{talbot,poncelet}@ircam.fr

Résumé

La vérification de modèles consiste à établir la satisfiabilité d'une formule logique pour un système donné. Ceci implique l'exploration des états du système par des algorithmes de recherche exhaustifs pouvant être couteux. Ces algorithmes sont souvent propres aux outils de vérification de modèles et, à l'instar de la programmation par contraintes, efficaces seulement pour une classe de problèmes. Ces outils manquent donc de modularité. Afin de résoudre ce problème, nous proposons le paradigme de programmation espace-temps, basé sur la théorie des treillis et la programmation synchrone, qui considère une stratégie de recherche comme un ensemble de processus collaborant pour explorer un espace d'état. Nous utilisons ce langage pour réunir la programmation par contraintes et la vérification de modèles et explorons son utilité pour vérifier des propriétés logiques.

Abstract

Model-checking is a paradigm for verifying logic properties on a given system by exploring exhaustively the state-space of the system. However, and similarly to constraint satisfaction problems, there is not a single algorithm working well for all problems. The existing tools lack of modularity and compositionality. To solve these problems, we propose spacetime programming, a paradigm based on lattices and synchronous process calculi that views search strategies as processes working collaboratively towards the resolution of a CSP. Using this language, we investigate relations between model-checking and constraint programming for verifying properties on a model.

1 Introduction

La vérification de modèles [1]—*model-checking* en anglais—est une technique qui vise à établir la satisfiabilité de propriétés logiques pour un modèle donné. De nombreux vérificateurs de modèles ont été implémentés et utilisés en industrie pour vérifier des systèmes imposants [19, 2, 11]. Malgré cela, le problème de l'*explosion d'espace d'état* est fréquemment rencontré et la vérification d'un modèle devient trop couteuse (en temps ou en espace mémoire).

Pour palier à ce problème, plusieurs techniques et paradigmes sont utilisés dont la programmation par contraintes [6, 7, 16]. Les approches existantes proposent des algorithmes efficaces pour une classe spécifique de problème et leurs extensions restent difficiles. De plus, la combinaison du paradigme par contraintes avec la vérification de modèle est enchevêtrée et difficile à comprendre.

Nous proposons d'utiliser spacetime, un paradigme de programmation basé sur la théorie des treillis et la programmation synchrone (section 3). Un programme spacetime considère des stratégies de recherche comme des processus qui explorent simultanément un espace d'état. On compose ensuite ces stratégies à l'aide d'un temps logique, propre au paradigme synchrone, qui synchronise les processus. De cette façon, nous utilisons la programmation par contraintes pour détecter les états inatteignables d'un modèle (section 4). Dès lors, l'idée principale est de composer la stratégie de recherche permettant de montrer la satisfiabilité d'un problème de contraintes avec celle permettant de vérifier des propriétés sur les états du système.

Ces travaux sont préliminaires et le système présenté dans cet article reste en cours d'implémentation. Nous avons néanmoins un compilateur d'un fragment du langage présenté qui permet de spécifier des stratégies d'exploration [1].

*Papier doctorant : Pierre Talbot est auteur principal.

1. Disponible publiquement sur github.com/ptal/bonsai.

2 Préliminaires

2.1 Vérification de modèles

La vérification de modèles [1] consiste à établir la satisfiabilité d'une formule logique Φ pour un modèle $\mathcal{M}$ donné. Ce problème suppose deux modélisations : celle du système (sous la forme d'une machine à état fini) et de la propriété à satisfaire (sous forme d'une formule logique). Une formule Φ est composée de plusieurs *formules atomiques* définissant l'ensemble AP qui est soit une étiquette du modèle—signifiant "mon état est dans ce nœud"—ou une contrainte sur des valeurs de variables (e.g. $x > 0$). Une méthode d'étiquetage (couramment dénotée $\mathcal{L}$) retourne pour un état s de $\mathcal{M}$ l'ensemble des *formules atomiques* satisfaites par s. Le modèle $\mathcal{M}$ est ensuite parcouru pour vérifier, si pour chaque état s, l'ensemble retourné par $L(s)$ satisfait la formule Φ. Finalement, le résultat d'une vérification de modèles est soit la confirmation de la satisfaction de Φ par le modèle $\mathcal{M}$, soit son infirmation accompagnée d'un contre-exemple guidant l'utilisateur depuis un état initial de $\mathcal{M}$ jusqu'à un état atteignable s_{erreur} invalidant la propriété Φ.

Les algorithmes de parcours de $\mathcal{M}$ sont au centre de la complexité des vérificateurs de modèle qui pour les plus simples sont une extension des parcours en profondeur ou en largeur d'un graphe. En revanche, les plus compliqués (par exemple [4]) visent à combiner différents algorithmes d'exploration pour optimiser le temps et l'espace nécessaire lors du calcul de la satisfiabilité de Φ.

Formellement, on définit un modèle $\mathcal{M}$ comme un ensemble de *graphes de programme* (modélisant le graphe de flux de contrôle -CFG- et les synchronisations de chaque programme) : $PG_i = \langle Loc, \mathsf{Effect}, \hookrightarrow, Loc_0, g_0 \rangle_i$ sur un même ensemble d'actions Act_τ, d'instructions $Stmt$ et de variables Var, où :

(i) Loc est un ensemble de nœuds (en anglais *locations*) ;

(ii) $\mathsf{Effect} : Stmt \times \mathsf{Eval}(Var) \to \mathsf{Eval}(Var)$ est un ensemble de fonctions d'effets ;

(iii) $\hookrightarrow$: $Loc \times \mathsf{Cond}(Var) \times Act \times Stmt \times Loc$ est l'ensemble de transitions ;

(iv) $Loc_0 \subseteq Loc$ est un ensemble des nœuds initiaux ;

(v) $g_0 \in \mathsf{Cond}(Var)$ est un ensemble de conditions initiales sur les variables de PG_i.

Avec $\mathsf{Cond}(Var)$ l'ensemble des conditions booléennes sur l'ensemble Var et $\mathsf{Eval}(Var)$ la fonction d'évaluation de variables. On notera $\ell \xrightarrow{g:a:\alpha} \ell'$ pour $\langle \ell, g, a, \alpha, \ell' \rangle \in \hookrightarrow$ avec ℓ le nœud source et ℓ' le nœud cible d'une transition, g sa garde (la condition pour la prise de cette transition) et a (resp. α) l'action de communication (resp. l'effet) de celle-ci.

La sémantique d'un ensemble de graphes de programme est définie par un système de transition TS qui décrit le comportement de n programmes $PG_1 \| \dots \| PG_n$ s'exécutant en parallèle. TS est défini par $\langle \vec{S}, Act, \to, \vec{I}, AP, L \rangle$ où : (i) $\vec{S} = \langle s_1, \dots, s_n \rangle$ avec $s_i : Loc_i \times \mathsf{Eval}(Var)$ est une paire constituée d'un nœud du programme PG_i et des valeurs de ses variables ; (ii) $Act = \bigcup_{i \in [1 \dots n]} Act_i \cup \tau$ est l'ensemble des actions des programmes avec l'action τ spécifiant les transitions internes d'un système ; (iii) $\to$: $\vec{S} \times Act \times \vec{S}$ sont les transitions possibles définies ci-après ; (iv) $\vec{I} = \langle I_1, \dots, I_n \rangle$ le vecteur d'états initiaux, (v) $AP = \bigcup AP_i$ est l'ensemble des propositions atomiques, et (vi) $L(\langle \ell, \eta \rangle) = \{\ell\} \cup \{g \in \mathsf{Cond}(Var) \mid \eta \models g\}$ est la fonction d'étiquetage retournant les propositions atomiques satisfaites pour un état $s = \langle \ell, \eta \rangle \in \vec{S}$. Les transitions $\to$ sont définies par les règles suivantes :

$$\frac{s_i = \langle \ell_i, \eta_i \rangle \xrightarrow{g:\tau:\alpha}_i \langle \ell'_i, \eta'_i \rangle = s'_i \qquad \eta_i \models g}{\langle s_1, \dots, s_i, \dots, s_n \rangle \xrightarrow{\tau} \langle s_1, \dots, s'_i, \dots, s_n \rangle}$$

avec $\eta'_i = \mathsf{Effect}(\alpha, \eta_i)$

$$\frac{\begin{array}{c} s_i = \langle \ell_i, \eta_i \rangle \xrightarrow{g_i:a?:\alpha_i}_i \langle \ell'_i, \eta'_i \rangle = s'_i \qquad \eta_i \models g_i \\ s_j = \langle \ell_j, \eta_j \rangle \xrightarrow{g_j:a!:\alpha_j}_j \langle \ell'_j, \eta'_j \rangle = s'_j \qquad \eta_j \models g_j \end{array}}{\langle s_1, \dots, s_i, \dots, s_j, \dots, s_n \rangle \xrightarrow{a} \langle s_1, \dots, s'_i, \dots, s'_j, \dots, s_n \rangle}$$

avec $\eta'_k = \mathsf{Effect}(\alpha_k, \eta_k)$ pour $k = i, j$ et $a \in Act$. Nous définissons la fonction $\mathsf{post}(\vec{S}) = \{\ell \xrightarrow{g:a:\alpha} \ell' | s = \langle \ell, \eta \rangle \in \vec{S} \wedge \langle \ell, \eta, g, a, \alpha, \ell', \eta' \rangle \in \to\}$ qui pour un ensemble d'états $\vec{S}$ retourne toutes les transitions actives des graphes de programmes sous-jacents.

Pour représenter nos propriétés à vérifier, nous considérons dans ce papier un sous-ensemble de la logique du temps arborescent—plus connue sous le nom de *computation tree logic* (CTL) [3]. Une formule CTL se décompose en deux parties : les propriétés sur les états, notées Φ, et celles sur les chemins, notées φ. Nous utilisons la syntaxe classique pour les états et le chemins sur l'ensemble des propositions atomiques AP :

$$\Phi ::= \text{true} \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi$$

$$\varphi ::= \bigcirc\ \Phi \mid \Phi_1\ \mathsf{U}\ \Phi_2$$

avec $a \in AP$.

Pour terminer, nous appelons Sat l'ensemble des états satisfaisant une formule CTL Φ. Un modèle satisfait une propriété Φ si $\exists i \in I \mid i \models \Phi$, c'est-à-dire si au moins un de ses états initiaux satisfait la formule Φ.

2.2 Programmation par contraintes

La programmation par contraintes est un paradigme déclaratif constitué de variables, définies sur des domaines, et des relations entre ces variables—appelées contraintes. On définit un problème de satisfaction de contraintes—*Constraint Satisfaction Problem* (CSP) en anglais—comme un tuple $\langle d, C\rangle$ où d est une fonction des variables vers un ensemble de valeurs, appelé le *domaine* de la variable, et C est l'ensemble des contraintes posées sur les variables. En pratique, un CSP est résolu en entrelaçant deux étapes : la propagation qui enlève les valeurs des domaines qui ne satisfont pas au moins une contrainte, et une étape de recherche qui permet de faire un choix lorsqu'on est "bloqué" et de revenir en arrière si le choix était mauvais. Dans cette article, on considère les algorithmes de recherche exhaustifs sur des domaines finis. Nous donnons quelques définitions mathématiques telles que trouvées dans [22].

Formellement, on considère un ensemble de variables $\mathcal{X}$ et un ensemble de valeurs $\mathcal{V}$. Une affectation d'une variable à une valeur est une fonction $a : \mathcal{X} \to \mathcal{V}$. On note l'ensemble de toutes les affectations avec l'exponentiation ensembliste $Asn := \mathcal{V}^{\mathcal{X}}$. Une contrainte $c \in 2^{Asn}$ est l'ensemble de toutes les valeurs acceptées par cette contrainte. Une affectation $a \in c$ est une solution de la contrainte c. Sans perdre en généralité, on considère qu'une variable est toujours contrainte par un domaine et on définit la fonction d tel que $d : \mathcal{X} \to Asn$. Par conséquent, un domaine est un ensemble d'affectations et est défini par la contrainte $con(d) := \{a \in Asn \mid \forall x \in \mathcal{X} : a(x) \in d(x)\}$. Les solutions d'un CSP $\langle d, C\rangle$ sont définies par $sol(\langle d, \mathcal{C}\rangle) := \{a \in con(d) \mid \forall c \in \mathcal{C} : a \in c\}$.

Opérationnellement, les contraintes sont implémentées par des algorithmes de filtrage—appelés propagateurs—réduisant le domaine des variables. Soit l'ensemble des domaines $Dom := 2^{\mathcal{V}^{\mathcal{X}}}$, un propagateur p est une fonction $p : Dom \to Dom$ qui doit vérifier deux propriétés :

(P1) Contractant : $\forall d \in Dom, p(d) \subseteq d$ ce qui signifie que p peut seulement réduire les domaines.

(P2) Correct : $\forall d \in Dom$ et $\forall a \in d$, $p(\{a\}) = \{a\}$ implique $a \in p(d)$. Le propagateur p ne peut pas rejeter des affectations valides.

Le rôle d'un propagateur est donc de réduire les domaines mais aussi de vérifier si une affectation est valide. Un propagateur $p \in P$, où P est l'ensemble des propagateurs, induit une contrainte c tel que $c_p := \{a \in Asn \mid p(\{a\}) = \{a\}\}$. Un problème de propagation $\langle d, P\rangle$ est un CSP $\langle d, \{c_p \in \mathcal{C} \mid p \in P\}\rangle$.

Soit un problème de propagation $\langle d, P\rangle$ et $p_1, \ldots, p_n \in P$, l'étape de propagation consiste à obtenir un point fixe où $d' \subseteq d, p_1(\ldots p_n(d')) = d'$, c'est-à-dire que les propagateurs ne peuvent plus réduire les domaines. La théorie de la propagation étudie des algorithmes permettant d'obtenir ce point fixe le plus rapidement possible [22].

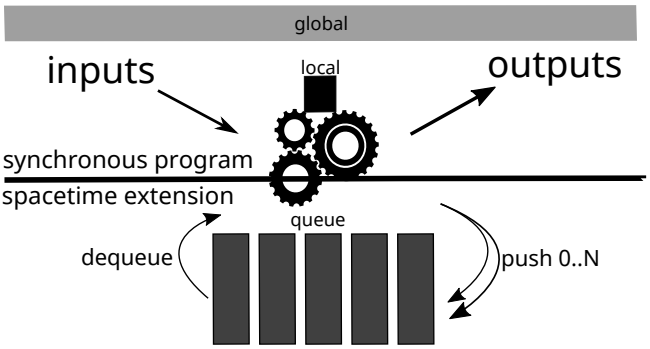

FIGURE 1 – Extension espace-temps du paradigme synchrone.

L'étape de propagation ne permet pas de résoudre un problème de contraintes. En effet considérons $x \neq y$ avec $x, y = \{1, 2\}$, le propagateur $p_{\neq}$ ne peut pas réduire davantage les domaines sans violer $P2$. Par conséquent on doit faire un choix afin d'énumérer les solutions possibles. La technique usuelle est d'utiliser une fonction $branch : Dom \to 2^{Dom}$ qui divise les domaines tel que $\forall d.sol(d) = \bigcup_{d' \in branch(d)} sol(d')$.

3 Programmation espace-temps

3.1 Définitions

Un ensemble ordonné $\langle D, \leq\rangle$ est un treillis complet si chaque sous-ensemble $S \subseteq D$ a une plus petite borne supérieure et une plus grande borne inférieure. Un treillis complet est toujours borné, c'est-à-dire qu'il possède un supremum $\top \in D$ tel que $\forall x \in D.x \leq \top$ et un infimum $\bot \in D$ tel que $\forall x \in D.\bot \leq x$.

Dans cet article, on considère la relation de déduction $x \models y \equiv y \leq x$ pour $x, y \in D$ qui signifie que y peut être déduit de x. On parle aussi de l'opération de jointure $x \vee y$ qui est égale à la plus petite borne supérieure de l'ensemble $\{x, y\}$ dans l'ensemble D. On note $x \leftarrow y$ pour $x = x \vee y$.

3.2 Paradigme de programmation

La programmation espace-temps—dont nous appellerons le langage spacetime—est un paradigme qui étend le modèle synchrone [8] avec des variables définies sur des treillis et un opérateur de non-déterminisme (dans le sens du *backtracking* ici). L'exécution d'un programme synchrone est décomposée en

une séquence d'instants logiques conceptuellement instantanés. Le programme produit donc des valeurs à l'instant t en fonction de l'instant $t-1$ et de signaux de l'environnement. Le paradigme synchrone permet de mettre en parallèle des processus de manière déterministe qui avancent simultanément d'instant en instant.

En spacetime, on voit un processus comme une stratégie d'exploration d'un espace d'état et un programme comme un ensemble de processus évoluant de manière synchrone. De plus, nous utilisons des variables avec une structure de treillis ce qui permet aux processus d'interagir dans l'instant sans que cela pose les problèmes habituels de *deadlock* ou d'indéterminisme. En effet, lorsque deux processus écrivent dans une variable en même temps, la valeur écrite est automatiquement combinée grâce à l'opérateur de jointure du treillis [21]. Ces treillis sont programmés dans un langage hôte, Java dans notre cas, sous forme de classes et nous imposons que toutes méthodes sur ces treillis soient monotones : on ne peut que rajouter de l'information. Cette idée de méthodes monotones sur des objets avec une structure de treillis nous vient du langage Bloom^L [5] utilisé pour programmer des systèmes distribués.

Dans la Figure 1, nous voyons dans la partie haute qu'un programme synchrone répond à des stimulus externes et peut stocker des variables dans une mémoire globale—préservant la valeur des variables au travers des instants—ou locale à l'instant courant. Afin de représenter le non-déterminisme, nous utilisons une troisième mémoire, sous forme de file, représentant l'espace d'état restant à explorer. À chaque instant, un nœud est enlevé de cette file et instancié. À la fin de chaque instant, l'opérateur non-déterministe `space` nous permet d'ajouter N éléments dans cette file représentant N nouveaux états successeurs à explorer dans les prochains instants. Lors de la déclaration d'une variable, nous utilisons trois attributs afin de distinguer dans quelle mémoire on stock les variables : (1) `single_time` pour les variables dans la mémoire locale qui sont ré-initialisées à leur valeur d'origine entre chaque instant ; (2) `single_space` pour les variables dans la mémoire globale et (3) `world_line` pour les variables se plaçant dans la file. Pour les variables `single_space` et `world_line`, on impose que leur évolution soit monotone pour l'entièreté ou un chemin de l'exploration de l'espace d'état.

3.3 Un solveur de contraintes minimal

Le paradigme espace-temps est utile pour isoler les différents composants d'un solveur de contraintes dans des processus distincts. On considère un solveur basique mais fonctionnel basé sur la bibliothèque Choco [17]. En particulier, nous considérons les classes `VStore` et `CStore` qui sont des abstractions (sous forme de treillis) des variables et des contraintes utilisées en Choco [2].

Nous utilisons spacetime pour programmer des stratégies d'exploration de l'arbre généré par la résolution d'un problème de contraintes. Afin de pouvoir composer des stratégies, nous utilisons l'interface suivante :

```
interface Search =
  proc search;
end
```

Elle impose aux modules qui vont l'implémenter de spécifier un processus qui s'appelle `search`. On utilise un module principal implémentant cette interface pour composer les différentes sous-stratégies :

```
module Solver implements Search =
  ref world_line VStore domains;
  ref world_line CStore constraints;
  proc search =
    trap FoundSolution in
      par
       || stop_on_solution()
       || propagation()
       || branch()
      end
    end
  [...]
```

Ce module reflète la définition mathématique d'un CSP $\langle d, C\rangle$ au travers des variables `domains` et `constraints`. Le mot-clé `ref` permet de référencer des variables qui sont des paramètres du module. En effet, le modèle du CSP est donné en entrée du module et est défini de manière usuelle à l'aide des méthodes du système de contraintes sous-jacent. Les différents composants de l'algorithme d'exploration sont codés séparément et composés à l'aide de l'instruction parallèle `par`. Cet opérateur parallèle est compilé vers du code séquentiel et n'est donc pas un parallèle au sens du *multithreading*. Tous les processus utilisent la boucle `loop` P qui exécute le programme P indéfiniment. Afin de respecter la condition qu'un instant doit être "instantané", il faut que P contienne l'instruction `pause`, retardant l'exécution du reste du processus à l'instant suivant.

Le premier processus permet d'arrêter l'exploration lorsqu'on a trouvé une solution :

```
proc stop_on_solution =
  loop
    when domains |= constraints then
      exit FoundSolution;
    end
    pause;
  end
```

2. Le code correspondant est relativement court : 200 lignes de code.

On utilise l'expression conditionnelle `when` afin de lancer l'exception `FoundSolution` lorsqu'on a trouvé une solution. Quand l'exception est lancée, le corps de l'instruction `trap` est terminé dans l'instant suivant et dans ce cas, le programme sera arrêté. L'expression `domains |= constraints` est vraie lorsqu'on peut déduire les contraintes du domaines, c'est-à-dire lorsqu'on a une solution au CSP (on décrit cette relation plus en détail en section 6).

On délègue la propagation des domaines au solveur Choco via le code suivant :

```
proc propagation =
  loop
    constraints.propagate(domains);
    pause;
  end
```

On appelle la méthode `propagate` sur les contraintes qui va se charger de réduire les domaines. Cette opération est bien monotone vu qu'on ne fait que rajouter de l'information dans les domaines.

Finalement, le processus `branch` se charge de couper le domaine des variables :

```
proc branch =
  loop
    single_time IntVar x = fail_first_var (domains);
    single_time Integer v = middle_value(x);
    space
    || constraints <- x.leq(v);
    || constraints <- x.ge(v);
    end
    pause;
  end
end
```

Ce processus décrit l'arbre de recherche avec une stratégie "fail-first". Elle choisit la variable x qui a le plus petit domaine afin de "rater au plus vite" et ne pas s'aventurer trop loin dans l'arbre—ce qui a un coût non négligeable. La fonction `middle_value` permet ensuite de récupérer la valeur v représentant le milieu du domaine. Finalement, l'arbre est construit avec l'instruction `space` qui décrit deux futurs différents : dans l'un $x \leq v$ et dans l'autre $x > v$.

4 Description de l'espace d'état

Nous reprenons les définitions des sections 2.1 et 2.2 pour définir formellement un problème de vérification de modèles par un problème de programmation par contraintes. Pour cela nous utilisons le langage spacetime présenté en section 3 et une abstraction du système de transition ts fournissant les méthodes suivantes :

post retourne l'ensemble des transitions possibles pour l'état courant du système. Le comportement de cette fonction suit la définition de la section 2.1 sans se soucier des valeurs des variables (nous projetons un état du système sur les nœuds courants des programmes uniquement, dénotés $\vec{\ell}$).

apply applique une transition à l'état courant du système. Ceci a pour effet de mettre à jour l'état du système en fonction de la transition.

4.1 Mise en commun des deux formalismes

L'espace d'état d'un problème de vérification de modèles doit être compris en terme d'une structure de treillis. De la sorte, on peut utiliser ce système comme une variable dans un programme spacetime. Les notations de ces deux formalismes ont l'équivalence suivante : la vérification de modèles utilise les ensembles Var, $\mathsf{Eval}(Var)$ dénotés η et Effect qui correspondent respectivement aux ensembles $\mathcal{X}$, à la fonction d'affectation a et à l'ensemble Asn définis en section 2.2.

Une question importante est de transformer le modèle vers un programme spacetime correspondant. On définit le système de transition abstrait ts, de sorte que chaque variable d'un graphe de programme PG soit définie sur un domaine. L'union de ces variables forme donc le premier élément d d'un CSP $\langle d, C\rangle$. Les gardes des transitions peuvent être ajoutées à l'ensemble C sans traitement particulier. En revanche les affectations, par exemple $x := x + 3$, ne peuvent pas être directement transformées vers une contrainte (car la contrainte correspondante est insatisfiable). L'idée est d'utiliser les contraintes de flux [12] et de considérer une variable x comme un flux de valeurs $x_1, \ldots, x_n$ où x_i est la $i$ème affectation de x. De cette manière l'affectation précédente est transformée vers la contrainte $x_{i+1} = x_i + 3$. Ces contraintes de flux permettent de lier les variables du problèmes de contraintes dans des états différents du modèle—nous revenons sur ce point en section 4.2.

L'espace d'état est un treillis de la forme $\langle cs, \vec{\ell}\rangle$ où : $cs : \langle d, C\rangle$ est le système de contraintes courant, et $\vec{\ell} : Loc^n$ est l'ensemble des n nœuds actifs de l'état courant (pour chaque PG_i avec $1 \leq i \leq n$).

L'ordre de ce treillis est défini tel que $\langle cs, \vec{\ell}\rangle \leq \langle cs', \vec{\ell'}\rangle$ si $cs < cs'$ ou si $cs' = cs$ alors $\vec{\ell} \leq \vec{\ell'}$. Ceci nous permet de détecter que l'on repasse sur un nœud déjà visité, c'est-à-dire quand $(\vec{\ell} = \vec{\ell'})$ sans avoir plus d'information $(cs \models cs')$.

On implémente en spacetime le module principale `ModelChecker` :

```
module ModelChecker =
  world_line VStore domains;
  world_line CStore constraints;
  world_line TS ts;
   [...]
```

À l'instar de `Solver` (section 3), ce module contient le CSP courant avec les variables `domains` et `constraints`. Le type de la variable `ts` est la classe Java `TS` fournissant les méthodes post et apply. On remarque que ces variables sont déclarées avec l'attribut `world_line` ainsi, lors d'un retour en arrière dans l'espace d'état, elles seront automatiquement restaurées.

4.2 Création de l'espace d'état

Une transition sur le treillis traduit "un pas" dans la vérification du modèle. Selon la définition de post et avec m la somme du nombre des transitions actives pour chaque graphe de programme, nous avons m transitions. Pour notre problème de contraintes, un pas implique : (i) le calcul des transitions possibles à partir de l'état courant $\vec{\ell}$, (ii) pour chacune de ces transitions $\ell_i \xrightarrow{g:a:\alpha} \ell'_i$:

1. le calcul de l'état cible ℓ', et,
2. l'ajout de la garde g et affectation α dans l'ensemble de contraintes : $C' = C \wedge g \wedge \alpha$.

(iii) le calcul du prochain "pas" pour chaque état cible, avec comme état du treillis $\langle cs', \vec{\ell'} \rangle$.

Remarque : Il est possible que deux transitions soient appliquées (si une communication est présente), dans ce cas notre système de transition retourne l'union des deux gardes ($g = g_1 \wedge g_2$) et des deux affectations ($\alpha = \alpha_1 \wedge \alpha_2$) pour traiter l'étape ii.2.

De ces définitions, on implémente le processus qui définit l'espace d'état :

```
module ModelChecker =
  [...]
  proc model_checker =
    par
    || next_transition ();
    || prune_unreachable ();
    end
```

Le processus `model_checker` lance deux processus en parallèle : `next_transition` s'occupe de passer les transitions suivantes du système ts, et `prune_unreachable` surveille l'état du système courant et coupe les états non atteignables. Le premier est implémenté comme suit :

```
proc next_transition =
  loop
    single_time Set<Transition> next_transitions =
      ts.post(domains, constraints);
    space t in next_transitions
    || ts.apply(t, domains);
       constraints <- t.guards();
       constraints <- t.effects();
    end;
    pause;
  end
```

Le calcul des prochaines transitions est donc géré par l'appel post sur la variable `ts` qui retourne un ensemble de transitions. On visite chacune de ces transitions dans des instants futurs. En programmation par contraintes, il s'agit de la fonction `branch` présentée à la section 3. On utilise la syntaxe `t in next_transitions` pour créer dynamiquement m futurs possibles (et donc passage de transition). On applique les effets de la transition en réalisant la jonction des gardes et des effets de la transition courante avec les contraintes courantes *constraints*. Cette instruction modélise le non-déterminisme présent dans une simulation de modèle lorsque plusieurs transitions sont applicables ; c'est la séparation de l'espace d'état des transitions.

Les contraintes jouent un rôle majeur pour détecter les états inatteignables et vérifier la satisfiabilité du problème de contraintes courant. On sait que résoudre un problème de contraintes nécessite sa propre exploration d'un espace d'état : dans chaque état nous calculons une solution du problème de contraintes courant. On note cependant que le problème de contraintes n'est *pas ré-initialisé* et qu'on repart en fait de la solution précédente, devenue partielle à cause des nouvelles variables de flux générées et contraintes.

```
proc unreachable_prune =
  loop
    universe
      Solver solver = new Solver(domains, constraints);
      solver.solve();
    end
    when not(domains |= constraints) then
      prune;
    end
    pause;
  end
```

On utilise l'instruction `universe` pour encapsuler la résolution du problème de contraintes. Cette instruction a pour effet de créer une nouvelle file et d'être exécutée tant que la file n'est pas vide, qu'une exception est lancée ou que le programme est terminé. Ce mécanisme est bien connu dans les langages synchrones sous le terme de raffinement temporel [15]. De plus, nous verrons en sections 5 et 6 que cette résolution est combinée avec la vérification de propriété logique. Finalement, si le problème est montré insatisfiable, alors on coupe l'espace d'état courant avec l'instruction `prune`, on ne veut pas explorer de transitions supplémentaires.

5 Formule comme stratégie d'exploration

Nous avons vu en section 2.1 que l'algorithme d'exploration d'un modèle dépend de la formule logique à

vérifier. Les deux logiques les plus répandues sont *Linear Temporal Logic* (LTL) et *Computation Tree Logic* (CTL). Nous nous restreignons à un sous-ensemble de ces logiques pour vérifier des propriétés de sécurité—*safety* en anglais—sur les états du système de transition. La logique considérée ici est la suivante :

$$\phi ::= c \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \forall\Diamond c \mid \exists\Diamond c$$

L'unique atome $c \in C$ est une contrainte définie sur les variables du modèle. On remarque que les quantificateurs ne peuvent pas apparaitre en séquence et nous expliquons cette limitation dans la section 5.1. Afin d'encoder ces formules en spacetime, nous capturons le comportement d'une formule dans l'interface `Formula` qui sera implémentée par les différents opérateurs logiques que nous présentons ensuite :

```
interface Formula =
  Consistency consistency;
  proc satisfy ;
```

On voit un processus comme la spécification de la formule et son implémentation sous forme de stratégie explorant l'espace d'états. Le processus `satisfy` établit la consistance de la formule et stock le résultat dans la variable `consistency`. Le treillis `Consistency` contient les éléments $\{true, false, bot, top\}$ où $bot \leq false \leq top$ et $bot \leq true \leq top$. La valeur `bot` indique donc que la consistance de la formule n'a pas encore été établie, `true` que la formule est satisfiable et `false` insatisfiable.

Encodage de formules logiques La conjonction est encodée par un module prenant en paramètre générique deux autres sous-formules :

```
module Conjunction<F: Formula, G: Formula>
 implements Formula =
  single_time Consistency consistency = bot;
  ref F f;
  ref G g;
  proc satisfy =
    par
    || f. satisfy () <*> g.satisfy()
    || consistency <- f.consistency.and(g.consistency);
    end
```

On exécute en parallèle les deux sous-formules en appelant respectivement leur processus `satisfy`. On remarque que ces deux sous-formules sont composées à l'aide de l'opérateur `<*>` : si une des deux veut couper l'arbre—via `prune`—on doit tout de même attendre la fin de l'autre processus. Par exemple, un processus peut établir que sa formule est vraie avant d'explorer l'entièreté du sous-arbre mais il faut tout de même continuer l'exploration pour le processus parallèle. Le troisième processus met à jour la consistance de la conjonction des formules à l'aide de la méthode `and`. Elle est programmée comme une conjonction classique de deux booléens où `true` $\wedge$ `unknown` est égal à `unknown`. La variable `consistency` est annotée avec `single_time` car la consistance d'une formule n'est valide que dans l'instant courant.

Le module `Negation` permet la négation d'une sous-formule et est implémenté comme suit :

```
module Negation<F: Formula> implements Formula =
  single_time Consistency consistency = bot;
  ref F f;
  proc satisfy =
    par
    || f. satisfy ()
    || consistency <- f.consistency.neg()
    end
```

Similairement à la conjonction on appelle une méthode sur la variable `consistency` de la sous-formule qui, dans ce cas, permet d'inverser la valeur de consistance.

L'encodage des quantificateurs universel et existentiel autorisent la vérification de propriétés dans un chemin. On se concentre ici sur un sous-ensemble où $\forall\Diamond c$ (resp. $\exists\Diamond c$) signifie que tous les (resp. un des) chemins possèdent un état dans lequel l'atome c est vérifié. On encode le quantificateur universel de la manière suivante :

```
module UniversalEventually =
  single_time Consistency consistency = bot;
  ref Atom f;
  proc satisfy =
    trap TreeInconsistent in
      loop
        par
        || f. satisfy ()
        || tree_inconsistency ()
        || path_consistency()
        end
        pause;
      end
    end
  proc tree_inconsistency =
    when f.consistency |= false /\ top then
      consistency <- false;
      exit TreeInconsistent;
     end
  proc path_consistency =
    when f.consistency |= true then
      prune;
    end
```

Ce module ne peut être initialisé qu'avec une formule atomique dû aux restrictions imposées précédemment. Le sous-processus `tree_inconsistency` permet de détecter que l'atome `f` est faux dans la branche courante et que donc, la formule du quantificateur universel est également fausse. La condition `top` est vraie lorsqu'on est dans une feuille de l'arbre (l'espace courant est en échec) et qu'on va revenir en arrière. Dans ce cas, on peut établir l'inconsistance et stopper le processus en lançant l'exception `TreeInconsistent` qui permet de sortir de la boucle. À contrario, le processus

`path_consistency` détecte que le chemin courant est consistent et qu'on peut arrêter l'exploration du sous-arbre courant, d'où l'instruction `prune`. Grâce à notre définition de la conjonction, composant les processus à l'aide de l'opérateur `<*>`, `prune` sera locale à la formule et l'exploration du sous-arbre peut continuer dans un autre processus.

De manière similaire, nous pouvons définir le quantificateur existentiel :

```
module ExistentialEventually<F: Formula> =
  single_time Consistency consistency = bot;
  ref Atom f;
  proc satisfy =
    trap PathConsistent in
      loop
        par
        || f.satisfy ()
        || path_consistency()
        end
        pause;
      end
    end
  proc path_consistency =
    when f.consistency |= true then
      consistency <- true;
      prune;
      exit PathConsistent;
    end
```

Dans ce cas, on cherche à trouver un chemin consistent et l'on s'arrête dès que l'atome sous-jacent est satisfiable. On lance également l'exception `PathConsistent` qui permet de sortir de la boucle et arrêter le processus.

On termine par le module `Atom` implémentant une propriété atomique sous forme de système de contraintes :

```
module Atom implements Formula =
  single_time Consistency consistency = bot;
  ref world_line VStore domains;
  ref world_line CStore constraints;
  ref world_line CStore property;
  proc satisfy =
    Entailment ent = new Entailment(model, property);
    par
    || ent.entail ()
    || consistency <- ent.consistency
    end
```

Ce module prends trois variables en paramètres : le CSP avec les variables `domains` et `constraints` et la propriété à vérifier `property`. La propriété atomique doit être définie sur les mêmes variables que le modèle. On cherche ensuite à établir si $\langle d, C\rangle \models \mathit{property}$, c'est-à-dire si la propriété atomique peut-être déduite de l'état courant. Ce problème de déduction—*entailment* en anglais—a une complexité potentiellement dans NP (en fonction du système de contraintes) et nécessite donc d'être résolu également avec une stratégie d'exploration. On peut réduire le coût de cette requête de déduction car nous devons de toutes façons prouver la satisfiabilité du CSP courant. Grâce à la sémantique de composition d'espace de spacetime, nous pouvons résoudre toutes les requêtes de déduction (générées par les atomes) ainsi que la satisfiabilité du problème en un unique passage dans l'arbre de recherche.

On a vu en section 3 comment implémenter un solveur de contraintes en spacetime et nous pouvons donc l'utiliser pour ce problème. L'algorithme de déduction implémenté par le module `Entailment` est présenté à la section 6.

5.1 Limitation de spacetime pour LTL et CTL

Nous avons utilisé un fragment restreint des logiques généralement utilisées en vérification de modèles telles que LTL et CTL. Prenons l'exemple des opérateurs "jusqu'à" $\Phi_1 \,\mathsf{U}\, \Phi_2$ et "suivant" $\bigcirc\Phi$ présents dans les deux logiques. Le premier est vrai si Φ_1 est vrai jusqu'à ce que Φ_2 soit vrai. Le deuxième est vrai si dans l'état suivant Φ est vrai. Ces formules agissent donc sur des chemins de l'espace d'état. Considérons la formule $(\bigcirc\Phi_1)\mathsf{U}\Phi_2$. On doit vérifier que Φ_1 est vrai dans le prochain état, le problème étant que dans ce prochain état on devra également vérifier $\Phi_1 \wedge \bigcirc\Phi_1$. Intuitivement, il y a deux manières de traiter le problème :

1. Par redémarrage : on "prends de l'avance" sur l'exploration dans le premier état pour vérifier $\bigcirc\Phi_1$. On explore une partie du sous-arbre (en concordance avec Φ_1) et une fois la satisfiabilité établie ou réfutée, on redémarre l'exploration à partir de l'état courant.
2. Par duplication : la formule est dupliquée de tel sorte que dans le second état on vérifie $\Phi_1 \wedge \bigcirc\Phi_1$.

Pour le moment, le paradigme de programmation spacetime ne permet pas d'exprimer ces deux comportements facilement. Notamment la duplication de code n'est pas permise ce qui rend la deuxième proposition difficile. Une solution serait d'introduire un opérateur de réplication $!P$ équivalent à $P \parallel \mathit{pause}; P \parallel \ldots$ tel que trouvé dans le π-calcul [13]. Ce problème intéressant est laissé à de futurs travaux.

6 Algorithme de déduction

6.1 Définitions

Lors de la résolution d'un CSP $\langle d, P\rangle$, une optimisation bien connue consiste à enlever les propagateurs qui ne peuvent plus apporter d'information. Pour cela, on regarde la relation $\langle d, \emptyset\rangle \models \langle d, p \in P\rangle$ que nous noterons plus simplement $d \models_d p$. D'un point de vue algorithmique la relation peut soit être vraie, fausse

ou ne pas exister :

$$d \preceq_{\models_d} p = \begin{cases} \models_d & \text{if } \forall a \in d, p(\{a\}) = \{a\} \\ \not\models_d & \text{if } \forall a \in d, p(\{a\}) = \{\} \\ \not\preceq_{\models_d} & \text{if } \exists a_1, a_2 \in d \text{ s.t.} \\ & p(\{a_1\}) = \{a_1\} \ \wedge p(\{a_2\}) = \{\} \end{cases}$$

On peut généraliser la relation sur un ensemble de propagateurs $d \models_d P$ ce qui est équivalent $\forall p \in P.d \models_d p$. En se basant sur ces définitions, on peut définir un problème de déduction plus général $\langle d, P\rangle \models \langle d, P'\rangle$ comme suit :

$$\langle d, P\rangle \preceq_{\models} \langle d, P'\rangle = \begin{cases} \models & \text{if } \forall d' \in sol(\langle d, P\rangle), d' \models_d P' \\ \not\models & \text{if } \forall d' \in sol(\langle d, P\rangle), d' \not\models_d P' \\ \not\preceq_{\models} & \text{if } \exists d', d'' \in sol(\langle d, P\rangle), \\ & d' \models_d P' \wedge d'' \not\models_d P' \end{cases}$$

Intuitivement, on peut déduire une information P' d'un CSP $\langle d, P\rangle$ si P' est valide pour toutes les solutions de ce CSP. La non-déduction $\langle d, P\rangle \not\models \langle d, P'\rangle$ signifie qu'on ne pourra jamais déduire P' de $\langle d, P\rangle$ même si on rajoute des contraintes dans P. Finalement, on ne sait pas si on peut ou non déduire P' dans le cas où seulement une partie du CSP permet la déduction. Il manque donc des contraintes pour pouvoir donner une réponse définitive.

6.2 Algorithme

L'algorithme spacetime suivant implémente la définition mathématique de déduction :

```
module Entailment =
  single_time Consistency consistency = bot;
  single_space Consistency entailment = bot;
  ref world_line VStore domains;
  ref world_line CStore constraints;
  ref world_line CStore property;
  proc entail =
    trap Contradiction in
      universe
        loop
          par
          || entailment_on_solution()
          || stop_on_contradiction()
          pause;
        end
      end;
      consistency <- entailment
    end
  proc entailment_on_solution =
    when domains |= constraints then
      entailment <- domains |= property;
    end
  [...]
```

De même que l'algorithme de satisfiabilité décrit en section 4, la requête de déduction est locale à un état du système de transition et il faut donc encapsuler cette résolution dans un univers sous-jacent. Dans le sous-processus `entailment_on_solution`, à chaque fois qu'on est dans un nœud solution, on met à jour le statut de la déduction avec la variable `entailment`. Si `entailment` est égale à *true* ou *false* à la fin, c'est qu'on a pu établir la (non-)validité de la propriété. Dans le cas où cette variable est égale à *top*, c'est que le résultat ne peut pas encore être établi. Dans ce cas, on fait une petite optimisation en arrêtant la vérification de la propriété courante :

```
proc stop_on_contradiction =
  when entailment |= top then
    prune;
    exit Contradiction;
  end
```

Il existe d'autres optimisations notamment en prenant en compte que la vérification d'une propriété peut se faire sur des nœuds non terminaux dans l'arbre d'exploration. Nous ne couvrons pas ces optimisations dans cette article et laissons ce travail pour des travaux futurs.

7 Travaux connexes

La vérification de modèles et la programmation par contraintes ont souvent été utilisé ensemble, notamment pour transformer la vérification de modèle dans un CSP. Principalement, de nombreux travaux utilisent la programmation logique par contraintes (CLP). On peut également encoder un problème de vérification de modèles en CLP et ainsi vérifier des propriétés de sûreté et de vivacité [7, 6, 16]. En particulier, la sémantique de la logique CTL est encodée sous forme de clauses de contraintes et le modèle est abstrait dans un tableau pour réduire le calcul [14].

Ces travaux sont généralement proposés pour répondre à un problème venant d'une application spécifique. En outre, la plupart des solutions existantes transforment un problème de vérification de modèle vers un problème de satisfiabilité de contraintes. Elles permettent difficilement d'utiliser des idées d'algorithmes d'exploration venant des deux mondes.

Dans un autre registre, considérer un programme synchrone comme une formule logique a déjà été envisagé pour tester des programmes dans le même paradigme [20]. Un des avantages de cette approche étant de rendre la formule programmée plus lisible que dans une logique comme CTL. Cependant, cette solution est spécialisée aux systèmes synchrones et ne permet pas de programmer l'exploration d'un espace d'état.

8 Conclusion

Nous avons introduit formellement le domaine de la vérification de modèle et de la programmation par contraintes. Nous proposons d'utiliser le paradigme *espace-temps* afin d'explorer des pistes communes aux algorithmes d'exploration utilisés par les deux domaines. Ceci nous permet de spécifier un algorithme de vérification de modèles utilisant la programmation par contraintes pour vérifier les états inatteignables d'un modèle. De plus, nous considérons la vérification du système à partir d'une formule logique (basée sur un sous-ensemble de CTL). L'idée est de voir cette formule logique comme une stratégie d'exploration dans l'espace d'état. Grâce à spacetime, nous pouvons composer ces formules logiques avec l'algorithme de satisfiabilité d'un problème de contrainte. Ainsi, les processus spacetime décrivent à la fois le parcours dans le système de transitions—en appliquant les gardes et affectations des transitions au CSP courant—et la stratégie de recherche pour résoudre la satisfiabilité d'un sous-ensemble de formules CTL.

Les travaux proposés dans cet article sont préliminaires. Il va sans dire que les travaux futurs sont nombreux, le premier étant de terminer l'implémentation d'un vérificateur de modèles tel que décrit dans cet article. Le compilateur de spacetime permet de programmer des stratégies d'exploration et l'implémentation du vérificateur de modèles basé sur ces stratégies est en cours de développement.

Une piste de recherche intéressante est de considérer les algorithmes d'exploration utilisés en vérification de modèles. Par exemple, la stratégie de recherche "à changement de contexte limité" [18, 10] permet d'obtenir de meilleures contre-exemples en considérant en premier lieu les chemins où les processus s'entrelacent le moins. En fait, cette stratégie est une instance de stratégie bien connue de la programmation par contraintes : la recherche en profondeur limitant le nombre d'erreur de l'heuristique—*limited discrepancy search* [9].

Références

[1] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, May 2008.

[2] T. Chen, M. Diciolla, M. Kwiatkowska, and A. Mereacre. Quantitative verification of implantable cardiac pacemakers. In *2012 IEEE 33rd Real-Time Systems Symposium*, pages 263–272, Dec 2012.

[3] Edmund M. Clarke and E. Allen Emerson. *Design and synthesis of synchronization skeletons using branching time temporal logic*, pages 196–215. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[4] Sylvain Conchon, Amit Goel, Sava Krstic, Alain Mebsout, and Fatiha Zaïdi. Invariants for finite instances and beyond. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 61–68, 2013.

[5] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 1. ACM, 2012.

[6] Giorgio Delzanno and Andreas Podelski. *Model Checking in CLP*, pages 223–239. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999.

[7] Giorgio Delzanno and Andreas Podelski. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer*, 3(3) :250–270, 2001.

[8] Nicolas Halbwachs. *Synchronous Programming of Reactive Systems*. 1993.

[9] William D. Harvey and Matthew L. Ginsberg. Limited discrepancy search. In *IJCAI (1)*, pages 607–615, 1995.

[10] Gerard J. Holzmann and Mihai Florian. Model checking with bounded context switching. *Formal Aspects of Computing*, 23(3) :365–389, 2011.

[11] Pim Kars. The application of promela and spin in the bos project (abstract). 1996.

[12] Arnaud Lallouet, Yat Chiu Law, Jimmy HM Lee, and Charles FK Siu. Constraint programming on infinite data streams. In *International Joint Conference on Artificial Intelligence*, pages 597–604, 2011.

[13] Robin Milner. *Communicating and mobile systems : the pi-calculus*. Cambridge University Press, 1999.

[14] Ulf Nilsson and Johan Lübcke. Constraint logic programming for local and symbolic model-checking. In *Proceedings of the First International Conference on Computational Logic*, CL '00, pages 384–398, London, UK, UK, 2000. Springer-Verlag.

[15] Cédric Pasteur. Raffinement temporel et exécution parallèle dans un langage synchrone fonctionnel, 2013.

[16] Andreas Podelski. Model checking as constraint solving. In Jens Palsberg, editor, *Static Analysis : 7th International Symposium, SAS 2000, Santa*

Barbara, CA, USA, June 29 - July 1, 2000. Proceedings, pages 22–37. Springer Berlin Heidelberg, 2000. DOI : 10.1007/978-3-540-45099-3_2.

[17] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2015.

[18] Shaz Qadeer and Dinghao Wu. KISS : keep it simple and sequential. *Acm sigplan notices*, 39(6) :14–24, 2004.

[19] Anders P. Ravn, Jiří Srba, and Saleem Vighio. Modelling and verification of web services business activity protocol. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems : Part of the Joint European Conferences on Theory and Practice of Software*, TACAS'11/ETAPS'11, pages 357–371, Berlin, Heidelberg, 2011. Springer-Verlag.

[20] P. Raymond, X. Nicollin, N. Halbwachs, and D. Weber. Automatic testing of reactive systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, RTSS '98, pages 200–. IEEE Computer Society, 1998.

[21] Vijay A. Saraswat and Martin Rinard. Concurrent constraint programming. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232–245. ACM, 1989.

[22] Guido Tack. *Constraint Propagation – Models, Techniques, Implementation*. PhD thesis, Saarland University, 2009.