Étude de l'analyse statique de programmes synchrones par interprétation abstraite

Rapport de stage

Alex Auvolat, juin-juillet 2014, sous l'encadrement de Jean-Louis Colaço

Dans le cadre de ma L3 d'informatique à l'ENS, j'ai été amené à faire un stage de deux mois dans une entreprise, ANSYS–Esterel Technologies, dont le coeur de métier est la conception d'outils certifiés permettant la programmation de systèmes embarqués critiques. Le langage SCADE développé par ANSYS–Esterel Technologies appartient à la famille des langages flot de données synchrones, particulièrement adaptés à cette tâche.

Étant donné la nécessité de prouver un certain nombre de propriétés critiques à la sûreté de tels systèmes, de nombreuses techniques ont été mises en oeuvre. Une de ces techniques est l'interprétation abstraite, une forme d'analyse statique basée sur un cadre théorique puissant [4] et dont la mise en oeuvre a déjà permis de grandes réalisations [2].

Ce stage se place dans la continuité des travaux déjà effectués sur l'interprétation abstraite des langages synchrones [5], [6], [7]. Il s'agit ici de faire de l'interprétation abstraite sur des programmes SCADE 6, un langage qui bénéficie de nouvelles structures : les blocs de contrôle [3], utilisés notamment pour écrire explicitement des machines à états (ou automates), structures que l'on ne trouve pas dans les versions précédentes, ni dans le langage historique Lustre.

Les outils actuellement disponibles pour faire de l'interprétation abstraite travaillent en général sur du C ; on peut citer par exemple Astrée ou PolySpace. Ainsi, les techniques d'interprétation abstraite les plus couramment utilisées pour vérifier des programmes SCADE dans l'industrie sont basées sur l'analyse de code C généré par compilation de SCADE. En nous intéressant directement au code SCADE d'origine, le but recherché ici est de pouvoir profiter des propriétés implicites et explicites du programme SCADE dans notre analyse (structures de contrôle, garanties d'initialisation, modèle mémoire simple), propriétés qui sont essentiellement perdues lors de la compilation vers C.

Pendant mon stage, j'ai été amené à développer un nouveau domaine abstrait, basé sur les domaines numériques classiques de l'interprétation abstraite (polyèdres, octogones), et permettant de prendre en compte de manière plus fine des disjonctions de cas, disjonctions qui s'avèrent cruciales pour permettre à l'analyse de conclure. J'ai réalisé un prototype d'implémentation de ce domaine abstrait en OCaml, en me basant sur la bibliothèque Apron pour la partie numérique. J'ai testé avec succès ce prototype sur plusieurs exemples (de petite taille en comparaison à des programmes industriels), ce qui a confirmé le potentiel de mon approche.

1 Contexte et outils

Dans cette section, je rappelle quelques informations sur la théorie et les outils qui m'ont été utiles lors de mon stage.

1.1 Langages flot de données synchrones

Les langages flot de données synchrones sont une famille de langage permettant d'exprimer un programme qui s'exécute sur un temps discret, composé d'un nombre fixe de variables définies par des équations mutuellement récursives, variables qui prennent à chaque cycle d'horloge une nouvelle valeur (toutes ensemble), en se basant sur les valeurs d'entrées à ce cycle et sur des mémoires propagées du cycle précédent.

Les variables considérées sont de types élémentaires : leurs valeurs à chaque cycle sont des entiers, des réels ou des booléens.

1.1.1 Définition sous forme de suites de valeurs

Une variable x définie dans un programme flot de données synchrone peut être vue comme la suite des valeurs qu'elle prend aux cycles successifs d'exécution du programme :

$$x \ = \ (x_0, x_1, ..., x_n, ...)$$

Un certain nombre de ces variables sont des variables d'entrée du système ; d'autres sont définies par combinaison d'opérateurs sur d'autres variables. Il existe deux types d'opérateurs :

- opérateurs combinatoires : ceux-ci étendent point par point une opération sur des valeurs numériques. Par exemple, l'addition de deux variables x et y est définie par :

$$x + y \;=\; (x_0 + y_0, x_1 + y_1, ..., x_n + y_n, ...)$$

- opérateurs séquentiels : ces opérateurs permettent de manipuler explicitement le temps dans les équations. Les deux opérateurs de base du langage Lustre sont pre et $\rightarrow$, qui sont définis comme suit :

$$\begin{aligned} \text{pre}\, x &= (\text{nil}, x_0, x_1, ..., x_{n-1}, ...) \\ x \rightarrow y &= (x_0, y_1, y_2, ..., y_n, ...) \end{aligned}$$

Une constante numérique ou booléenne représente la suite constante égale à cette valeur.

Les équations qui définissent nos différentes variables sont mutuellement récursives, mais tout cycle doit passer par au moins un opérateur pre, afin d'assurer la causalité du système, c'est-à-dire qu'aucune variable ne dépend instantanément d'elle-même.

1.1.2 Exemple de programme

Durant tout mon stage, l'exemple suivant a été un exemple canonique :

```
node updown() returns(x: int)
var up: bool;
let
  up = true -> (if pre up then pre x < 10 else pre x <= 0);
  x = 0 -> (if up then pre x + 1 else pre x - 1);
tel
```

Ce programme est constitué d'un noeud qui ne prend aucune entrée et renvoie en sortie un flot d'entiers `x`. Le fonctionnement de ce programme correspond à une machine à deux états, qui sont encodés dans la variable `up`. Lorsque `up = true`, la variable `x` est incrémentée à chaque cycle, et elle est décrémentée quand `up = false`. L'automate change d'état lorsque la variable `x` atteint une de ses bornes (10 dans l'état croissant et 0 dans l'état décroissant).

La suite x renvoyée par ce programme est donc la suivante :

$$x \;=\; (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 1, 2, 3, ...)$$

On souhaiterait réaliser une analyse par interprétation abstraite qui soit capable de déterminer que $x \in [0, 10]$.

1.1.3 Structures de contrôle étendues : les automates

Avec les structures de contrôle nouvelles introduites dans SCADE 6, on peut réécrire le programme précédent d'une façon beaucoup plus naturelle, en faisant apparaître la structure d'automate :

```
node updown() returns(x: int)
var nx: int;
let
  x = 0 -> pre nx;
  automaton
    state UP
      nx = x + 1;
    until if nx = 10 resume DOWN;
    state DOWN
      nx = x - 1;
    until if nx = 0 resume UP;
  returns nx;
tel
```

Avec la notion d'automate vient la notion de *scope*, c'est-à-dire un bloc d'équations qui a son horloge propre : les équations de ce bloc ne sont actives que lors de certains cycles. Les équations écrites dans le scope d'un état d'automate ne sont actives que lors des cycles où l'automate est dans cet état.

1.1.4 Définition sous forme de système de transition

On note $\mathbb{X}$ l'ensemble des variables de notre programme et $\mathbb{V}$ l'ensemble des valeurs prises (entiers, booléens, réels). On note $\mathbb{M} = \mathbb{X} \to \mathbb{V}$ l'ensemble des environnements mémoire, c'est-à-dire des valuations des variables de $\mathbb{X}$.

Un programme SCADE peut maintenant être vu comme un système de transition sur des environnements mémoire, partant d'un environnement $s_0 \in \mathbb{M}$ et recevant à chaque cycle des entrées i_n, lui permettant de passer de s_{n-1} à s_n. Les variables de sortie du système sont une partie des variables apparaissant dans s_n.

$$s_0 \xrightarrow{i_1} s_1 \xrightarrow{i_2} s_2 \to \cdots$$

Un formalisme utile consiste à découper le cycle en deux fonctions. La première fonction, que l'on note c, copie un certain nombre de variables de l'environnement s_{n-1} sous un nouveau nom. Ces variables servent de mémoire pour l'autre système. À ces définitions de variables on rajoute les définitions des variables d'entrée données par i_n. La seconde fonction, notée f, s'occupe alors de calculer toutes les variables de s_n à partir des informations qu'on lui donne.

$$s_n \;=\; f(c(s_{n-1}) + i_n)$$

La fonction f correspond habituellement à l'exécution d'un code impératif issu de la compilation du programme synchrone. Dans notre étude, nous voyons plutôt f comme l'application d'une formule logique représentant les contraintes entre les variables du système. Cette vision, très adaptée à l'interprétation abstraite, permet de mettre de côté toute problématique d'ordonnancement des équations.

1.2 Interprétation abstraite

1.2.1 Environnements mémoire et abstraction

Lorsque l'on travaille sur de l'analyse statique, on est souvent amenés à étudier des ensembles d'environnements mémoire, c'est-à-dire des éléments de $\mathcal{P}(\mathbb{M})$. Lorsqu'on étudie du code C séquentiel, ces valeurs sont définies à chaque point de contrôle du programme (à chaque lieu entre deux instructions successives) comme l'ensemble des combinaisons de valeurs qui peuvent être prises par les variables du programmes à cet endroit lors d'une exécution.

$\mathcal{P}(\mathbb{M})$ ne peut pas être manipulé par un programme informatique, car celui-ci ne peut travailler que sur des représentations finies d'objets. On construit donc des abstractions qui approximent des éléments de $\mathcal{P}(\mathbb{M})$. Une abstraction est définie par un domaine de représentation $\mathcal{D}^\#$, dont les éléments représentent une partie de $\mathcal{P}(\mathbb{M})$. Le sens d'une représentation est donné par la fonction de concrétisation, qui à une représentation abstraite associe l'ensemble des environnements représentés :

$$\gamma \;:\; \mathcal{D}^\# \to \mathcal{P}(\mathbb{M})$$

$\mathcal{D}^\#$ doit être munit d'une structure de treillis, basée sur l'existence de plusieurs opérateurs : une relation d'ordre partiel, notée $\sqsubseteq$, deux éléments spéciaux, $\bot$ et $\top$ (élément minimal et élément maximal), ainsi que deux opérateurs $\sqcup$ et $\sqcap$ d'arité deux (borne supérieure et borne inférieure).

On travaille toujours par sur-approximation, c'est-à-dire que la valeur abstraite $s^\#$ utilisée pour représenter les exécutions d'un programme à un certain point contient toujours au moins les exécutions réelles :

$$s \;\subseteq\; \gamma(s^\#)$$

Si on considère qu'une valeur abstraite $s^\#$ représente un ensemble de contraintes sur les variables des environnements mémoire, alors la relation $s \subseteq \gamma(s^\#)$ signifie que les contraintes exprimées par $s^\#$ sont toutes vérifiées dans les environnements mémoire de s.

On peut également définir une fonction de meilleure approximation qui, étant donné un élément s de $\mathcal{P}(\mathbb{M})$ donne les contraintes exprimables dans $\mathcal{D}^\#$ qui approximent « le mieux possible » l'ensemble d'environnements s :

$$\alpha \ : \ \mathcal{P}(\mathbb{M}) \to \mathcal{D}^\#$$

Dans le cadre de l'interprétation abstraite [4], pour garantir la sûreté de l'analyse on demande à ce que les fonctions α et γ réalisent une correspondance de Galois entre $\mathcal{D}^\#$ et $\mathcal{P}(\mathbb{M})$, c'est-à-dire que :

$$\forall s \in \mathcal{P}(\mathbb{M}), \forall t \in \mathcal{D}^\#, \qquad s \subseteq \gamma(t) \ \Leftrightarrow \ \alpha(s) \sqsubseteq t$$

Pour permettre à l'analyse de terminer dans le cas de l'analyse de boucles, où l'on est amenés à chercher un post-point fixe d'une certaine fonction, on fait également appel à un opérateur spécial : l'opérateur de widening ∇. Celui-ci est défini tel que pour toute suite $(s_n) \in (\mathcal{D}^\#)^{\mathbb{N}}$, la suite t_n définie par $t_0 = s_0$ et $t_{n+1} = t_n \nabla s_{n+1}$ est constante à partir d'un certain rang. Cela permet ainsi de garantir la convergence d'une suite de valeurs abstraites vers le post-point fixe recherché en un nombre fini d'itérations. En contrepartie, les approximations effectuées par l'opérateur de widening sont parfois problématiques, car elles induisent nécessairement une perte de précision dans la représentation.

1.2.2 Domaines numériques

Si $\mathbb{V} = \mathbb{N}$ ou $\mathbb{V} = \mathbb{Q}$, on dispose de bonnes approximations pour les valeurs de $\mathcal{P}(\mathbb{X} \to \mathbb{V})$.

Les plus simples sont les abstractions non-relationnelles, qui ne représentent pas les relations entre les variables mais uniquement les classes de valeurs des variables indépendamment les unes des autres. Un domaine abstrait non-relationnel particulièrement utilisé est basé sur des intervalles : $\mathcal{D}^\# = \mathbb{X} \to \text{itv}(\mathbb{V})$, où $\text{itv}(\mathbb{V})$ est le treillis des intervalles de $\mathbb{V}$. Un autre treillis non relationnel particulièrement utile est le treillis des signe, qui se contente de représenter le signe que peut prendre la valeur ($\geqslant 0, \leqslant 0, = 0, < 0, > 0, \neq 0$). Celui-ci est particulièrement utile, lorsqu'on le compose avec d'autres treillis, pour prouver l'absence de division par zéro dans un programme.

Il existe ensuite des domaines abstraits dits relationnels, qui sont capables de représentations plus fines, et en particulier de prendre en compte une certaine classe de relations entre les variables. Le plus ancien domaine relationnel est basé sur les polyèdres convexes : un nuage fini de points dans un espace à $|\mathbb{V}|$ dimensions est abstrait par son enveloppe convexe, enveloppe qui peut être décrite par un ensemble de contraintes linéaires (égalités ou inégalités). Cette abstraction permet en particulier de représenter de façon exacte des objets tels que les droites, segments, polygones, plans, demi-plans, etc. Un autre domaine relationnel à base d'octogones a été développé [8] ; celui-ci permet un compromis entre une précision moindre et une rapidité de calcul améliorée sur de grands nombres de variables.

1.2.3 La bibliothèque Apron

La bibliothèque Apron[1] propose une variété de domaines abstraits à disposition pour l'analyse de propriétés sur des variables numériques. En particulier, il y a une implémentation d'un domaine non-relationnel basé sur les intervalles (Box), d'un domaine basé sur les polyèdres (Polka) et d'un domaine basé sur les octogones (Oct). Ces trois domaines sont disponibles derrière une interface unifiée qui rend possible leur utilisation d'une manière simple et homogène, les rendant très facilement interchangeables.

1. `http://apron.cri.ensmp.fr/`

2 Élaboration d'un cadre théorique

Dans cette section, je décris le cadre théorique que j'ai élaboré lors de mon stage pour répondre à la problématique posée.

2.1 Sémantique collectrice sur un programme SCADE

Reprenons nos deux fonctions f et c qui définissent la fonction de transition de notre programme.

On fait une première abstraction, qui consiste à considérer toutes les entrées possibles pour le système lors d'un cycle : cela nous permet de déterminer, pour un état mémoire donné, l'ensemble de ses successeurs possibles. Pour cela, on définit sur $\mathbb{M}$ une propriété q qui est vraie pour un environnement mémoire si et seulement si il contient, pour les variables d'entrée du système, des valeurs conformes à une certaine spécification (cette spécification, donnée dans le texte du programme, permet de faire des hypothèses sur les entrées qui sont des conditions pour prouver une propriété sur les sorties). Pour $A \in \mathcal{P}(\mathbb{M})$, on pose l'ensemble de ses successeurs :

$$g(A) \;=\; \bigcup_{e \in A} \{f(a), a \in c(s), q(f(a))\}$$

On définit S_0 comme l'ensemble des environnements mémoire possibles avant le premier cycle : ceux-ci sont des environnements mémoire qui contiennent, en particulier, une variable *init* qui est vraie seulement à ce moment-là et qui indique que l'on est au temps zéro. Cela est particulièrement utile pour l'opérateur $\rightarrow$ de Lustre : la variable *init* indique que l'on est encore à gauche de la flèche.

Enfin, on définit :

$$S_{n+1} \;=\; g(S_n)$$

$$S \;=\; \bigcup_{n \in \mathbb{N}} S_n \;=\; \mathrm{lfp}_{S_0}(\lambda s.S_0 \cup g(s))$$

S est appelé *sémantique collectrice* du programme, et contient tous les environnements mémoire possibles pour le programme, quelles qu'aient été les entrées, et après un nombre arbitraire de cycles.

La sémantique collectrice offre une représentation générale du programme : si on est en mesure de prouver une propriété dans S (typiquement, on veut montrer qu'une variable booléenne du programme qui représente un invariant est toujours vraie) alors la propriété est garantie pour toutes les exécutions du programme.

2.2 Abstraction, itération et point fixe

Afin de prouver des propriétés sur S, on cherche à en construire une représentation abstraite $S^\#$. Si $S^\#$ contient comme contrainte une propriété que l'on souhaite prouver, alors on sait que cette propriété est effectivement vérifiée sur toutes les exécutions du programme.

2.2.1 Notations et outils

Les variables de notre programme sont de deux types : variables numériques (entiers, réels) et variables énumérées (booléens, états d'automates). On note $\mathbb{X}_n$ l'ensemble des variables numériques et $\mathbb{X}_e$ l'ensemble des variables énumérées. On note $\mathbb{V}_e$ l'ensemble fini des valeurs prises par les variables énumérées.

Notre programme SCADE est traduit sous la forme d'une formule logique qui représente les contraintes imposées sur les variables par la fonction f. Cette formule s'appuie sur les opérateurs $\wedge, \vee, \neg$ de la logique propositionnelle, qui permettent de combiner des contraintes sur les variables énumérées ou les variables numériques.

Les contraintes sur les variables numériques sont des égalités ou des inégalités sur les variables numériques. On note $\mathbb{E}_n$ l'ensemble de ces contraintes.

Les contraintes sur les variables énumérées sont du type $x \equiv v$ ou $x \not\equiv v$, où $x \in \mathbb{X}_e$ et $v \in \mathbb{V}_e$, ou bien du type $x \equiv y$ ou $x \not\equiv y$, avec $x, y \in \mathbb{X}_e$. On note $\mathbb{E}_e$ l'ensemble de ces contraintes.

On note $\mathbb{E} = \mathbb{E}_n \cup \mathbb{E}_e$.

La fonction de cycle c est représentée par un ensemble de variables $C \subset \mathbb{X}$ dont on a besoin de conserver les valeurs pour les calculs du cycle suivant. Cette fonction c crée un nouvel environnement mémoire où les valeurs des variables de C sont copiées dans de nouvelles variables.

2.2.2 Passage dans l'abstrait

f peut s'exprimer dans l'abstrait, puisque tous les opérateurs de la formule logique qui la définissent ont un équivalent dans l'abstrait (il faut supprimer les négations) : $\wedge$ et $\vee$ sont calculés avec $\sqcap$ et $\sqcup$, et les contraintes sur les variables sont traitées par le domaine abstrait. On note $f^\#$ la fonction abstraite correspondante.

De même c peut s'exprimer dans l'abstrait comme l'oubli des contraintes sur certaines variables et la copie de certaines valeurs. On note $c^\#$ la fonction abstraite correspondante.

On note $g^\# = f^\# \circ c^\#$. En considérant que c'est la formule qui définit $f^\#$ qui encode les contraintes sur les entrées, on a bien une abstraction de notre fonction g.

La représentation abstraite $S^\#$ de S est également définie comme un point fixe, c'est en fait la même définition que celle de S mais avec des opérations sur $\mathcal{D}^\#$:

$$S^\# \;=\; \mathrm{lfp}_{S_0^\#}\big(\lambda s.S_0^\# \sqcup g^\#(s)\big)$$

En faisant un appel judicieux à l'opérateur de widening, un post-point fixe de la fonction $\lambda s.S_0^\# \sqcup g^\#(s)$ peut être calculé en un nombre fini d'itérations, ce qui donne une sur-approximation de $S^\#$. Dans la suite, nous développons une autre méthode d'approximation, mieux adaptée au problème.

2.3 Disjonctions et itérations chaotiques

2.3.1 Motivations pour une nouvelle technique d'analyse

L'analyse d'un programme SCADE comme un seul point fixe calculé sur l'ensemble de la structure de contrôle s'avère insuffisante dans la majorité des cas, pour plusieurs raisons :

- Lors de l'analyse d'une boucle *while* dans un langage impératif, on peut s'appuyer sur la condition de sortie de la boucle pour inférer un invariant précis sur l'ensemble d'environnements accessibles dans le corps de la boucle. Dans notre cas, on s'intéresse à une boucle infinie du type *while(true)...* qui englobe l'ensemble du programme. Ainsi on ne peut pas profiter des conditions qui apparaissent à différents endroits du programme, en particulier les condition de sortie sur les transitions des automates.
- De manière générale, l'analyse gagne en précision si plutôt que de considérer une valeur abstraite dans un domaine numérique qui représente l'ensemble des exécutions, on partitionne cette représentation selon des valeurs de variables de type énuméré. Avec une seule valeur abstraite numérique, on n'est pas en mesure de représenter des invariants qui ne sont vrais, par exemple, que pour un état d'automate : on fait l'union abstraite de cet invariant avec un autre invariant vrai dans un autre état, ce qui peut mener à une perte de précision considérable. L'encodage des valeurs booléennes par les entiers 0 et 1 confiés au domaine numérique s'avère insuffisant pour exprimer ce genre de relations.

C'est pour résoudre ces problèmes qu'il m'est apparu nécessaire de développer un nouveau domaine abstrait capable de traiter des disjonctions de cas, ainsi qu'un processus d'itération basé sur le principe des *itérations chaotiques*.

2.3.2 Définition du domaine abstrait

On considère dans cette section que l'on a un domaine abstrait $\mathcal{D}_0^\#$ capable de gérer les contraintes sur les variables numériques, et éventuellement d'enregistrer des informations sur les variables énumérées, mais sans relation entre les deux (typiquement on se contente d'enregistrer pour chaque variable énumérée l'ensemble des valeurs qu'elle peut prendre). Le domaine $\mathcal{D}_0^\#$ représente une abstraction de $\mathbb{M} = \mathbb{X} \to \mathbb{V}$. On note $\bot_0$ et $\top_0$ les éléments minimaux et maximaux (*bottom* et *top*) de ce treillis, $\sqsubseteq_0$ sa relation d'inclusion, $\sqcup_0$ et $\sqcap_0$ ses opérateurs borne supérieure et borne inférieure (*lub* et *glb*), γ_0 la fonction de concrétisation associée, ainsi que ∇_0 son opérateur de widening. On note $[\![\cdot]\!]_0$ la fonction de restriction par une contrainte : si $c \in \mathbb{E}$ et $s \in \mathcal{D}_0^\#$, alors $[\![c]\!]_0(s)$ est une approximation (la meilleure possible) de $\{e \in \gamma_0(s) \mid e \vdash c\}$.

On choisit $\mathbb{X}_d \subset \mathbb{X}_e$ un sous-ensemble des variables énumérées qui vont nous servir à faire des disjonctions de cas. On note $\mathbb{V}_d$ l'ensemble (fini) des valeurs prises par ces variables.

La particularité des variables de disjonction est que l'on ne réalise pas d'approximation sur celles-ci : on représente directement un cas de la disjonction par une valuation de ces variables, dans $\mathbb{X}_d \to \mathbb{V}_d = \mathbb{M}_d$. On note σ l'injection canonique de $\mathbb{M}_d$ dans $\mathcal{P}(\mathbb{M})$: $\sigma(d) = \{e \in \mathbb{M} \mid \forall x \in \mathbb{X}_d, e(x) = d(x)\}$.

Le domaine abstrait que l'on construit est défini ainsi :

$$\mathcal{D}^\# \;=\; \mathbb{M}_d \to \mathcal{D}_0^\#$$

Autrement dit, pour chaque combinaison possible des variables énumérées que l'on a choisies, on enregistre des informations différentes concernant les autres variables dans le domaine de base.

Pour $s \in \mathcal{D}^\#$, la concrétisation $\gamma(s)$ est donnée par :

$$\gamma(s) \;=\; \bigcup_{d \in \mathbb{M}_d} \gamma_0(s(d)) \cap \sigma(d)$$

En supposant l'existence d'une fonction de meilleure abstraction α_0 sur $\mathcal{D}_0^\#$, on peut définir une meilleure abstraction sur $\mathcal{D}^\#$ en utilisant α_0 sur la restriction de notre ensemble d'environnements en entrée à chaque combinaison pour les variables de disjonction :

$$\alpha(s) \;=\; \lambda d.\alpha_0(s \cap \sigma(d))$$

L'inclusion entre deux valeurs abstraites correspond à l'inclusion des valeurs abstraites dans le domaine de base pour chaque combinaison des variables de disjonction.

$$s \sqsubseteq t \;\Leftrightarrow\; \forall d \in \mathbb{M}_d, s(d) \sqsubseteq_0 t(d)$$

Les éléments $\top$ et $\bot$ sont trivialement définis comme suit :

$$\bot = \lambda d.\bot_0 \qquad \top = \lambda d.\top_0$$

On vérifie bien que $\gamma(\bot) = \varnothing$ et $\gamma(\top) = \mathbb{M}$.

On définit les opérations $\sqcup$ et $\sqcap$ par une simple application point-par-point des opérateurs correspondants sur $\mathcal{D}_0^\#$:

$$\begin{aligned} s \sqcup t &= \lambda d.(s(d) \sqcup_0 t(d)) \\ s \sqcap t &= \lambda d.(s(d) \sqcap_0 t(d)) \end{aligned}$$

Les propriétés de treillis de $(\mathcal{D}^\#, \sqsubseteq, \bot, \top, \sqcup, \sqcap)$ découlent directement des propriétés de treillis de $\left(\mathcal{D}_0^\#, \sqsubseteq_0, \bot_0, \top_0, \sqcup_0, \sqcap_0\right)$ appliquées pour chaque $d \in \mathbb{M}_d$.

2.3.3 Correspondance de Galois

On a bien une correspondance de Galois entre $\mathcal{P}(\mathbb{M})$ et $\mathcal{D}^\#$.

Démonstration. Soit $c \in \mathcal{P}(\mathbb{M})$ et $a \in \mathcal{D}^\#$. Alors :

$$\begin{aligned} c \subseteq \gamma(a) &\Leftrightarrow \forall d \in \mathbb{M}_d, c \cap \sigma(d) \subseteq \gamma(a) \\ &\Leftrightarrow \forall d \in \mathbb{M}_d, c \cap \sigma(d) \subseteq \gamma_0(a(d)) \\ &\Leftrightarrow \forall d \in \mathbb{M}_d, \alpha_0(c \cap \sigma(d)) \sqsubseteq_0 a(d) \\ &\Leftrightarrow \forall d \in \mathbb{M}_d, \alpha(c)(d) \sqsubseteq_0 a(d) \\ &\Leftrightarrow \alpha(c) \sqsubseteq a \end{aligned}$$

□

2.3.4 Application de contraintes

Enfin, on peut définir un certain nombres de nouveaux opérateurs de restriction permettant de prendre en compte des contraintes sur les variables énumérées. Pour les contraintes ne faisant appel qu'à des variables de $\mathbb{X}_d$, c'est-à-dire qui sont pour chaque $d \in \mathbb{M}_d$ soit vraies soit fausses, on peut définir ces opérateurs de restriction comme éliminent de notre représentation les cas qui ne satisfont pas notre contrainte :

$$[\![c]\!](s) \;=\; \lambda d. \begin{cases} [\![c]\!]_0(s(d)) & \text{si } d \vdash c \\ \bot_0 & \text{sinon} \end{cases}$$

La condition de restriction c est également passée au domaine $\mathcal{D}_0$ sous-jacent car celui-ci peut également représenter des contraintes sur ces variables.

Pour toutes les expressions $c \in \mathbb{E}_e$ qui font appel à des variables hors $\mathbb{X}_d$, ainsi que pour les contraintes numériques, on ne peut s'appuyer que sur le domaine $\mathcal{D}_0^\#$ pour les prendre en compte. On définit donc :

$$[\![c]\!](s) \;=\; \lambda d. [\![c]\!]_0(s(d))$$

2.3.5 Widening et itérations chaotiques

L'opérateur de widening reste problématique. On pourrait définir un opérateur de widening point par point, comme on l'a fait pour les opérateurs $\sqcap$ et $\sqcup$. Mais on peut faire mieux : chaque cas de disjonction $d \in \mathbb{M}_d$ peut être vu un état d'un système de transitions, pouvant avoir des transitions vers lui-même ou vers un autre état ; les transitions étant gardées par des conditions. Il est donc intéressant d'étudier le fait de rester sur un état de ce système de transitions comme on ferait sur du code séquentiel l'analyse d'une boucle *while*, où la condition de sortie de boucle correspond à la disjonction des gardes des transitions sortantes : ces gardes nous permettent de calculer des invariants plus fins sur nos états.

Pour $d_0 \in \mathbb{M}_d$, notons $r_{d_0} : \mathcal{D}^\# \to \mathcal{D}^\#$ tel que $r_{d_0}(s) = \lambda d. \begin{cases} \bot_0 & \text{si } d \neq d_0 \\ s(d) & \text{si } d = d_0 \end{cases}$. On se donne $\tau \in \mathbb{N}$ un paramètre de l'analyse qui servira de délai de widening.

On propose de faire des itérations chaotiques comme suit, pour faire grossir à chaque étape une représentation s_n des états accessibles pour notre programme, jusqu'à ce qu'elle soit stable par $g^\#$, ce qui impliquera alors que l'on a une sur-approximation de S. Pour cela, on s'appuie sur deux valeurs auxiliaires :

- $\delta_n \subseteq \mathbb{M}_d$ est l'ensemble des cas de disjonction qui ont « grossi » à la dernière itération, et sur lesquels il faut donc refaire une passe afin d'être sûr d'explorer tous les successeurs
- $K_{\nabla,n} : \mathbb{M}_d \to \mathbb{N}$ est une fonction qui nous permet de mémoriser le nombres d'itérations depuis lequel un certain cas $d \in \mathbb{M}_d$ est considéré, afin qu'on puisse appliquer un widening sur ce cas-là lorsqu'un certain délai est dépassé.

Pour l'initialisation de notre procédé d'itérations chaotiques, on pose :

$$\begin{aligned} s_0 &= S_0^\# \\ \delta_0 &= \{d \in \mathbb{M}_d \mid s_0(d) \neq \bot_0\} \\ K_{\nabla,0} &= \lambda d.0 \end{aligned}$$

Puis, tant que $\delta_n \neq \varnothing$, on itère le calcul suivant :

- Choisir (arbitrairement) $a \in \delta_n$ un cas à étudier.
- Calculer D^0 un post-point fixe de la fonction suivante :

$$F(i) \;=\; r_a(s_n \sqcup g^\#(i))$$

Cela correspond à faire l'analyse du cas a sélectionné comme s'il s'agissait d'une boucle *while* : on profite, avec l'application de r_a, des conditions nécessaires à ce que l'on reste dans ce même cas d'un cycle au suivant. Pour ce calcul de post-point fixe, on peut avoir besoin de faire appel à un opérateur de widening, mais puisqu'on ne s'intéresse qu'à un seul cas de notre disjonction, l'opérateur de widening sur $\mathcal{D}_0^\#$ est le seul dont nous ayons besoin : c'est pourquoi on ne définit pas de widening sur $\mathcal{D}^\#$ (en fait dans cette étape on peut se contenter de travailler avec $\mathcal{D}_0^\#$).

- Poser $D = D^0 \sqcup g^\#(D^0)$: on s'intéresse maintenant à tous les successeurs (immédiats) de notre cas a, pour les valeurs D^0 que l'on a trouvé.
- Faire grandir notre représentation avec les nouvelles possibilités que l'on a trouvé dans D :

$$\begin{aligned} s_{n+1} &= \lambda d. \begin{cases} s_n(d) \nabla_0 D(d) & \text{si } K_{\nabla,n}(d) \geqslant \tau \text{ et } d \neq a \\ s_n(d) \sqcup_0 D(d) & \text{sinon} \end{cases} \\ K_{\nabla,n+1} &= \lambda d. \begin{cases} K_{\nabla,n}(d) + 1 & \text{si } s_{n+1}(d) \not\sqsubseteq_0 s_n(d) \\ K_{\nabla,n}(d) & \text{sinon} \end{cases} \\ \delta_{n+1} &= (\delta_n \backslash \{a_{n+1}\}) \cup \{d \in \mathbb{M}_d | s_{n+1}(d) \not\sqsubseteq_0 s_n(d)\} \end{aligned}$$

Lorsque $\delta_n = \varnothing$, on a la garantie que s_n est bien stable par $g^\#$. De plus comme $S_0^\# \sqsubseteq s_n$, on a bien une sur-approximation de la sémantique collectrice S.

2.4 Une représentation plus évolué, à base de graphes de décision

Plusieurs problèmes se posent avec le domaine abstrait défini à la section précédente. En particulier le fait que ce domaine impose de considérer séparément tous les cas pour toutes les variables choisies pour faire des disjonctions, alors qu'il y a de nombreux cas où plusieurs combinaisons de variables énumérées sont associées à la même valeur dans le domaine abstrait de base : il faudrait pouvoir réduire la représentation et traiter ça comme un seul cas de disjonction. La représentation précédente mène très vite à des explosions inutiles de la taille de la représentation, qui engendrent à leur tour des temps de calcul démesurément longs.

Nous proposons dans ce paragraphe une seconde représentation de ces valeurs abstraites, permettant de mieux traiter des problèmes ayant un nombre important de variables de type énuméré reliées entre elles par des relations complexes. Cette nouvelle représentation est basée sur des arbres de décision, sur lesquels on fait du partage de sous-arbre, ce qui les transforme en DAG (c'est la technique employée dans les graphes de décision de type BDD). On fait abstraction des problématiques de mémoisation et de partage des sous-graphes, qui font tout l'intérêt de la technique d'un point de vue pratique mais qui peuvent être considérés comme une optimisation à part, en dehors du formalisme mathématique qui donne la définition de cette représentation et des opérateurs qui travaillent dessus.

Pour le choix d'un même ensemble de variables de disjonctions $\mathbb{X}_d$, les deux représentations ont exactement la même capacité expressive. Ceci dit, j'ai été obligé d'adapter un peu le processus d'itérations chaotiques pour l'adapter à cette nouvelle représentation : en effet, la définition de ce qu'est un « cas » dans l'analyse est un peu différente.

Il semblerait qu'un tel domaine abstrait ait déjà été mis en pratique dans le projet BDDApron de B.Jeannet[2], mais je n'ai pas pu trouver de bibliographie satisfaisante sur le sujet.

Par la suite, on parlera indifféremment d'arbre de décision ou de graphe de décision.

2.4.1 Les graphes de décision

Un arbre, ou graphe de décision, correspond à une structure ayant un noeud racine, et où les noeuds internes correspondent à faire un choix de valeur pour une variable de décision. Les feuilles de cet arbre correspondent à des valeurs abstraites dans le domaine de base $\mathcal{D}_0^\#$. La valeur représentée par un tel graphe de décision correspond aux concrétisations de toutes les feuilles, chacune étant restreinte aux bons cas pour les variables énumérées qui ont été parcourues pour arriver jusqu'à cette feuille. Pour l'unicité de la représentation, on impose que tout chemin de la racine à une feuille parcoure les variables de $\mathbb{X}_d$ dans un certain ordre. Pour cela, on les numérote : $\mathbb{X}_d = \{x_1, x_2, ..., x_n\}$.

2. `http://www.inrialpes.fr/pop-art/people/bjeannet/bjeannet-forge/bddapron/`

Le type somme représentant un graphe de décision peut être défini comme suit :

$$\begin{array}{rcl} s & := & V\big(t \in \mathcal{D}_0^{\#}\big) \\ & | & C(x_i, v_1 \to s_1, v_2 \to s_2, ..., v_p \to s_p) \end{array}$$

On note $\mathcal{D}^{\#}$ l'ensemble des arbres ainsi formés.

Si on voit ça comme un arbre, la contrainte sur l'ordre des variables peut être exprimée par : si un noeud $C(x_i, ...)$ est ancêtre d'un noeud $C(x_j, ...)$, alors $i < j$.

Pour faciliter les notations, on introduit le rang d'un noeud, qui correspond à l'indice de la variable utilisée pour le choix fait dans ce noeud, ou à l'infini pour les feuilles de l'arbre (les feuilles arrivent après tous les noeuds de choix) :

$$\begin{array}{rcl} \delta(C(x_i, v_1 \to s_1, ..., v_p \to s_p)) & = & i \\ \delta(V(t)) & = & \infty \end{array}$$

La contrainte d'ordre se traduit par : pour tout noeud $C(x_i, v_1 \to s_1, ..., v_p \to s_p)$, on a $\forall j, i < \delta(s_j)$.

On se pose aussi la contrainte suivante : on n'a pas le droit d'avoir de noeud $C(x_i, v_1 \to s_1, v_2 \to s_2, ..., v_p \to s_p)$ si $s_1 = s_2 = \cdots = s_p$. La conjonction de ces deux contraintes implique l'unicité de l'arbre qui représente une valeur abstraite donnée.

La formule suivante donne la définition mathématique de la concrétisation :

$$\begin{array}{rcl} \gamma(V(t)) & = & \gamma_0(t) \\ \gamma(C(x_i, v_1 \to s_1, ..., v_p \to s_p)) & = & \bigcup_{j=1}^{p} \{s \in \gamma(s_j) \,|\, s(x_i) = v_j\} \end{array}$$

Les éléments $\top$ et $\bot$ sont trivialement définis comme suit :

$$\top = V(\top_0) \qquad \bot = V(\bot_0)$$

Pour assurer l'unicité lors des transformations, on définit la fonction de réduction r, qui ne garde un noeud de choix que si les différentes branches ne sont pas toutes égales :

$$r(x_i, v_1 \to s_1, ..., v_p \to s_p) \;=\; \begin{cases} s_1 & \text{si } \forall i, s_i = s_1 \\ C(x_i, v_1 \to s_1, ..., v_p \to s_p) & \text{sinon} \end{cases}$$

L'opération $\sqcap$ entre deux graphes de décision t et t' est définie par une double récurrence sur les deux graphes, où l'on descend à chaque fois d'un noeud dans le graphe dont le noeud racine a le degré le plus faible, pour conserver l'ordre des noeuds de choix dans l'arbre final. L'opération $\sqcap_0$ n'est utilisée que pour les feuilles ; des simplifications sont effectuées en appliquant la fonction r lorsque l'on remonte la structure construite. Ce qui donne :

$$\begin{array}{rcl} V(t) \sqcap V(t') & = & V(t \sqcap_0 t') \\ C\begin{pmatrix} x_i, & v_1 \to s_1 \\ & \vdots \\ & v_p \to s_p \end{pmatrix} \sqcap C\begin{pmatrix} x_i, & v_1 \to s'_1 \\ & \vdots \\ & v_p \to s'_p \end{pmatrix} & = & r(x_i, v_1 \to s_1 \sqcap s'_1, ..., v_p \to s_p \sqcap s'_p) \\ C(x_i, v_1 \to s_1, ..., v_p \to s_p) \sqcap t & \overset{\text{si } i \leq \delta(t)}{=} & r\begin{pmatrix} x_i, & v_1 \to s_1 \sqcap t \\ & \vdots \\ & v_p \to s_p \sqcap t \end{pmatrix} \end{array}$$

La définition se fait symétriquement pour le cas où le noeud de rang le plus faible se trouve à droite. L'opération $\sqcup$ est définie de même.

L'inclusion est également définie par induction structurelle : il faut que, quels que soient les choix que l'on fait dans s et s', si ces choix sont cohérents alors le sous-graphe obtenu de s est inclus dans le sous-graphe obtenu de s'.

L'égalité entre les valeurs représentées par deux graphes de décision est équivalente à l'égalité structurelle de ces deux graphes : c'est une propriété qui découle de l'unicité de la représentation.

2.4.2 Application de contraintes

Pour n'importe quelle contrainte $c \in \mathbb{E}$, on définit d'abord $[\![c]\!]'$ la fonction qui applique cette contrainte sur toutes les feuilles au niveau de $\mathcal{D}_0^\#$, et effectue ensuite les simplifications qui s'imposent :

$$\begin{array}{rcl} [\![c]\!]'(V(s)) & = & V([\![c]\!]_0(s)) \\ [\![c]\!]'(C(x_i,\ v_1 \to s_1, ..., v_p \to s_p\ )) & = & r(x_i, v_1 \to [\![c]\!]'(s_1), ..., v_p \to [\![c]\!]'(s_p)) \end{array}$$

Pour une contrainte c sur les énumérés de $\mathbb{X}_d$, on définit d'abord un arbre de décision χ qui représente la fonction booléenne associée à cette contrainte. Par exemple :

$$\begin{array}{rcl} \chi(x \equiv v) & = & C(x, v \to \top, v' \to \bot, v' \in \mathbb{V}_d \backslash \{v\}) \\ \chi(x_i \equiv x_j) & \overset{\text{si } i<j}{=} & C\left(\begin{array}{c} x_i, \\ \end{array} \begin{array}{c} v_1 \to \chi(x_j \equiv v_1) \\ \vdots \\ v_p \to \chi(x_j \equiv v_p) \end{array}\right) \end{array}$$

Il suffit ensuite de faire l'intersection de cette description de la contrainte avec le graphe obtenu par application de cette contrainte dans $\mathcal{D}_0^\#$:

$$[\![c]\!](s) \ = \ \chi(c) \sqcap [\![c]\!]'(s)$$

Pour les contraintes numériques, ou pour les contraintes faisant appel à des variables de $\mathbb{X}_e \setminus \mathbb{X}_d$, on n'a pas d'autre choix que de compter sur $\mathcal{D}_0^\#$; on pose donc :

$$[\![c]\!] \ = \ [\![c]\!]'$$

2.4.3 Équivalence des deux représentations

Il y a bien une bijection entre $\mathcal{D}^\#$ et la première représentation donnée en 2.3, que l'on notera dans ce paragraphe $\mathcal{D}_s^\#$; c'est-à-dire que pour le même domaine $\mathcal{D}_0^\#$ sous-jacent et pour le même choix de variables de disjonction $\mathbb{X}_d$, ces deux ensembles ont exactement la même capacité expressive. On construit cette bijection comme suit :

$$\Phi \ : \ \left\{ \begin{array}{lcl} \mathcal{D}^\# & \to & \mathcal{D}_s^\# \\ s & \mapsto & \lambda d.\mathrm{leaf}_d(s) \end{array} \right.$$

Où $\mathrm{leaf}_d(s)$ est la valeur de la feuille sur laquelle on arrive en suivant dans le graphe s les choix indiqués par la valuation $d \in \mathbb{M}_d$.

La bijection réciproque peut être exprimée par la construction d'un arbre explorant toutes les valuations de $\mathbb{M}_d$, auquel on applique notre opération de réduction pour produire la représentation canonique. On ne montre pas que c'est bien une bijection.

La relation d'inclusion $\sqsubseteq$ sur $\mathcal{D}^\#$ est bien mise en bijection avec la relation d'inclusion $\sqsubseteq_s$ sur $\mathcal{D}_s^\#$, en effet $\sqsubseteq$ est construite de telle sorte que $s \sqsubseteq t$ si et seulement si pour tout $d \in \mathbb{M}_d$, alors $\mathrm{leaf}_d(s) \sqsubseteq_0 \mathrm{leaf}_d(t)$.

2.4.4 Opérateur de widening

Les deux représentations ont beau avoir la même puissance expressive, la délimitation des différents cas de l'analyse est plus compliquée sur les graphes de décision que dans la première représentation. En effet, avant un cas correspondait à une et une seule valuation des variables de $\mathbb{X}_d$. Par ailleurs, cette valuation pouvait être représentée dans $\mathcal{D}_0^\#$ à partir du moment où celui-ci représente, pour chaque variable énumérée, la valeur qu'elle prend (ou l'ensemble des valeurs qu'elle peut prendre). Ainsi, on n'avait pas besoin d'utiliser un opérateur de widening sur le domaine représentant plusieurs cas : ∇_0 suffisait.

Dans cette deuxième formulation des disjonctions de cas, la définition d'un cas correspond maintenant à une formule booléenne sur les variables énumérées, et non à une seule valuation. On se retrouve donc à utiliser $\mathcal{D}^\#$ comme support aussi au moment du calcul du point fixe local lors d'une itération chaotique. C'est pourquoi on a besoin de définir un opérateur de widening sur celui-ci.

Une formule booléenne sur les variables de disjonction peut être représentée par un graphe de décision de $\mathcal{D}^\#$ dont les seules valeurs sur les feuilles sont $\bot_0$ et $\top_0$.

Pour identifier un cas dans notre graphe de disjonction, on utilise une fonction qui, étant donné une valeur $t_0 \in \mathcal{D}_0^\#$, extrait d'un graphe de disjonction t la formule booléenne de tous les chemins dans ce graphe qui mènent vers une feuille $V(s)$ avec $s = t_0$ exactement. Un cas est donc défini par une valeur de $\mathcal{D}_0^\#$. Formellement, l'extraction de cette formule logique est effectuée par un opérateur $\rho : \mathcal{D}_0^\# \times \mathcal{D}^\# \to \mathcal{D}^\#$ défini comme suit :

$$\begin{aligned} \rho(t_0, V(t)) &= \begin{cases} \top & \text{si } t = t_0 \\ \bot & \text{sinon} \end{cases} \\ \rho(t_0, C(x_i, v_1 \to s_1, ..., v_p \to s_p)) &= r(x_i, v_1 \to \rho(t_0, s_1), ..., v_p \to \rho(t_0, s_p)) \end{aligned}$$

En ce qui concerne l'opérateur de widening sur $\mathcal{D}^\#$, on pourrait se contenter de reprendre la définition de $\sqcup$ en remplaçant $\sqcup_0$ par ∇_0 lorsqu'on arrive sur les feuilles. Pour gagner en précision, on utilise ici une approche un peu plus subtile : lorsque l'on calcule $a \nabla b$ et que l'on arrive sur deux feuilles, on applique ∇_0 seulement si on est en train de faire l'union de deux feuilles apparaissant pour le même cas dans les deux graphes, c'est-à-dire si et seulement si les deux feuilles sont accessibles selon exactement la même formule booléenne sur les énumérés de $\mathbb{X}_d$ dans a et b.

Une définition formelle de l'opérateur de widening ∇ serait donc :

$$\begin{aligned} a \nabla b &= f^\nabla_{a,b}(a, b) \\ f^\nabla_{a,b}(V(t), V(t')) &= \begin{cases} V(t \nabla_0 t') & \text{si } \rho(t, a) = \rho(t', b) \\ V(t \sqcup_0 t') & \text{sinon} \end{cases} \end{aligned}$$

Les cas manquants sont définis par double récurrence sur les deux graphes de décision, comme pour les opérateurs $\sqcap$ et $\sqcup$, sauf qu'il faut garder une trace des deux graphes de décision originaux pour pouvoir appliquer ρ : c'est ce que permettent les indices a et b de $f^\nabla_{a,b}$.

L'intérêt de cette construction plutôt que d'appliquer ∇_0 systématiquement sur les feuilles est que l'on gagne en précision au moment où des cas nouveaux apparaissent dans notre point fixe : on ne se permet d'appliquer un widening que si le cas sur lequel on veut l'appliquer est déjà stabilisé.

2.4.5 Itérations chaotiques

Pour effectuer nos itérations chaotiques, on a besoin d'enrichir un peu notre arbre au niveau des feuilles, avec deux structures permettant d'enregistrer quelques informations supplémentaires : une étoile, indiquant « cette feuille est nouvelle, il faut l'analyser comme nouveau cas », et un indice $i \in \mathbb{N}$ correspondant à « cette feuille est là depuis i itérations », où le i permet d'implémenter un délai de widening.

$$\begin{aligned} s &:= V(t)_i \\ &\mid V(t)_i^* \\ &\mid C(x_i, v_1 \to s_1, v_2 \to s_2, ..., v_p \to s_p) \end{aligned}$$

On définit ensuite l'*opérateur d'accumulation*, dont le but est de faire grandir à chaque itération une valeur s_n qui représente les valeurs possibles pour notre programme que nous avons découvert jusqu'à maintenant. Cet opérateur est essentiellement le même que l'opérateur de widening ∇ que nous avons défini ci-dessus, mais seulement il s'appuie sur l'indice i de nos feuilles pour implémenter un délai de widening. Il s'occupe également de faire croître l'indice i lorsque l'on rajoute des nouveaux cas à la valeur initiale.

On définit finalement une fonction de détection des nouveaux cas par rapport à une valeur précédente : cette fonction rajoute des étoiles sur les feuilles de notre graphe de décision lorsque le cas représenté par cette feuille a grandi depuis la dernière itération. Cette opération correspond à la mise à jour de δ_n précédemment.

Le processus d'itérations chaotiques est ensuite essentiellement le même que celui que nous avons donné pour la première version. Pour choisir un cas à étudier lors d'une itération, on cherche une feuille $V(t_0)_i^*$ étoilée ; la définition du cas est ensuite donnée par $c = \rho(t_0, s_n)$. On utilise ensuite les données présentes dans l'arbre (étoiles, indices i) pour jouer le rôle de nos ensembles δ_n et $K_{\nabla,n}$; données exploitées grâce aux deux opérateurs que l'on vient de définir.

3 Mise en pratique et résultats

3.1 Le prototype

Dans le cadre de mon stage, j'ai été amené à mettre en pratique la théorie décrite ci-dessus. En premier lieu, j'ai implémenté un interprète pour la sémantique concrète du sous-ensemble de SCADE 6 que j'ai choisi, ce qui a été pour moi une façon de se familiariser avec le langage. J'ai ensuite implémenté un prototype d'analyseur statique, avec les deux représentations du domaine à disjonction définies ci-dessus, ainsi que les processus d'itérations chaotiques associés. Le tout s'appuie sur la bibliothèque Apron en ce qui concerne les domaines numériques (intervalles, polyèdres, octogones).

3.1.1 Détails sur les graphes de décision

L'implémentation des graphes de décision utilise extensivement du partage et de la mémoisation, ce qui permet de profiter au maximum de la structure de données. J'ai pu vérifier expérimentalement que cette deuxième structure est bien plus efficace que la première.

L'implémentation des graphes de décision que j'ai réalisée utilise actuellement uniquement des valeurs dans un domaine abstrait numérique pour les feuilles. On pourrait très simplement, étant donné la structure de mon code, utiliser une conjonction d'un domaine numérique et d'un domaine non-relationnel sur les variables énumérées, mais l'avantage à faire cela ne semble pas évident puisque la structure de graphe de décision représente déjà les informations sur les valeurs énumérées de façon satisfaisante.

Par ailleurs, les valeurs $\top$ et $\bot$ pour les feuilles du graphe sont représentées directement sans faire appel à la bibliothèque numérique sous-jacente, ce qui permet d'optimiser les temps de calcul.

3.1.2 Heuristique pour déterminer l'ordre des variables

Pour trouver un bon ordre sur les variables, c'est-à-dire un ordre permettant d'optimiser la taille du graphe de décision, j'ai implémenté une heuristique basé sur l'algorithme Force décrit dans [1]. Les groupes de variables énumérées utilisés pour l'application de l'algorithme sont créés en exploitant la structure de scopes imbriqués du programme SCADE. Cette heuristique a considérablement amélioré les performances de la méthode et m'a permis de résoudre un problème complexe comportant plus de 160 variables énumérées.

3.2 Résultats

3.2.1 Quelques preuves remarquables

Voici une liste (non exhaustive) de petits programmes qui m'ont servi d'exemples et sur lesquels j'ai pu prouver des propriétés intéressantes :

- Un compteur qui incrémente une variable x jusqu'à une certaine borne, et la remet à zéro lorsque la borne est atteinte. Pour une borne arbitrairement grande, on est capable de prouver avec un faible nombre d'itérations que la variable x est positive et inférieure à cette borne.
- Le compteur `updown` qui incrémente une variable jusqu'à une borne, puis la décrémente jusqu'à une autre borne. De même, pour des bornes arbitrairement grandes, on est capable de prouver que x est compris entre les deux bornes avec un faible nombre d'itérations.
- Deux compteurs `updown` en parallèle, tels que l'un monte toujours lorsque l'autre descend, et réciproquement. On est capable de montrer que la somme des deux compteurs est constante.
- Une variable x incrémentée lors des cycles pairs, une variable y incrémentée lors des cycles impairs. On montre que $x - y$ est borné et vaut 1 lors des cycles pairs et 0 lors des cycles impairs.
- Des exemples qui se rapprochent plus du *model-checking*, en particulier un programme permettant de contrôler les feux d'un système de demi-tour pour des rames de métro (exemple issu de [6]). Cet exemple non-trivial a permis de tester et d'améliorer les performances de l'implémentation des graphes de décision. En fait pour un programme qui n'utilise que des variables énumérées, il suffit de choisir $\mathbb{X}_d = \mathbb{X}_e$ pour que l'analyse ne fasse aucune approximation : on peut ainsi prouver n'importe quelle propriété booléenne sur ce genre de programmes.

3.2.2 Partitionnement dynamique

J'ai passé un peu de temps à expérimenter avec une analyse basée sur un partitionnement dynamique de S comme décrit dans [7]. L'idée d'une telle analyse consiste à ne pas déplier toute la structure de contrôle dès le début comme on le fait lorsque l'on se fixe un ensemble de variables de disjonctions, mais plutôt de définir explicitement des états pour un système de transition qui représente la sémantique de notre programme, définitions qui sont raffinées de façon itérée afin d'avoir une analyse de plus en plus précise. Pour raffiner ces définitions, on peut se baser sur des conditions qui apparaissent dans le texte du programme (conditions de `if` par exemple), ce qui permet de diviser un état en deux. Entre les raffinements, on calcule un point fixe sur les états que l'on définit par des itérations chaotiques, en traitant chaque état avec un domaine abstrait simple. À l'issue de ces itérations on élimine les états dont on a prouvé qu'ils étaient inaccessibles. Je n'ai pas tenté de faire d'analyse en arrière comme proposé dans [7] : l'idée était que l'analyse en arrière est guidée par une propriété, alors que pour ma part j'aimerais aussi pouvoir tout simplement faire tourner une analyse et que celle-ci soit capable, au bout d'un certain temps, de trouver des invariants toute seule. Toute la difficulté se trouve alors dans le choix du raffinement qu'il faut faire : en pratique, mes expérimentations n'ont pas été très concluantes car les raffinements ne se faisaient pas aux bons endroits, et l'ensemble d'états grandissait inutilement sans parvenir à prouver quoi que ce soit d'intéressant.

3.2.3 Expérimentations avec Astrée ; comparaison

Pendant mon stage, j'ai eu l'occasion d'utiliser le logiciel Astrée, développé à l'ENS et à l'INRIA par l'équipe de P.COUSOT[3], et spécialisé dans l'interprétation abstraite de programmes C. J'ai pu travailler sur les preuves de quelques programmes simples, issus de la compilation de programmes SCADE, et suis en mesure de fournir quelques points de comparaison.

Au niveau conceptuel, il y a de grandes différences entre les analyses que l'on peut faire avec Astrée et la technique que j'ai développée. Astrée travaille sur du code C, qui est très différent du code synchrone SCADE, ce qui a plusieurs conséquences. Premièrement, Astrée ne sait pas forcément profiter des propriétés qui sont garanties sur le code SCADE et qui sont dégradées lors de la compilation. Typiquement, la bonne initialisation des variables est difficile à prouver par Astrée, alors qu'elle est garantie par le typage de SCADE, ce qui peut provoquer de nombreuses fausses alarmes. Deuxièmement, la structure du code C ne correspond pas du tout à la structure d'un programme synchrone : le C est séquentiel, alors que le synchrone correspond typiquement à un système de transitions où toutes les variables évoluent en même temps. Ainsi il n'est pas naturel de faire une étude par itérations chaotiques sur du code C, alors que cela découle logiquement de la structure d'un programme synchrone. Précisons par ailleurs que pour analyser du code SCADE, on peut se contenter de le réécrire comme un système d'équations qui doit être vérifié lors du cycle, indépendamment de l'ordre des dépendances des variables. Ce n'est évidemment pas le cas en C, puisque la compilation effectue un tri par ordre de dépendance pour que les calculs puissent être réellement effectués. Il y a donc plusieurs avantages à se placer directement au niveau du code SCADE, plutôt que de travailler sur le code C généré.

Au niveau de l'implémentation, il y a aussi quelques différences :

- Astrée implémente un domaine abstrait capable de traiter des graphes de décision du style de ceux que je décris ci-dessus, mais ceux-ci sont limités dans la mesure où les feuilles de ce graphe ne peuvent pas être des valeurs abstraites relationnelles, mais simplement des intervalles de valeurs pour des variables.

- Astrée étant conçu pour fonctionner sur des programmes ayant un très grand nombre de variables, l'analyse de base n'est effectuée qu'avec des intervalles. Par dessus on peut rajouter un certain nombre de domaines relationnels (octogones, ellipsoïdes, graphes de décision), qui travaillent sur un petit nombre de variables à la fois, ce qui produit des raffinements locaux de l'analyse.

3. `http://www.astree.ens.fr/`

3.3 Prolongements envisageables

Le programme que j'ai développé lors de mon stage n'est qu'un prototype, et souffre de nombreuses limitations. Plusieurs pistes mériteraient d'être explorées pour développer un outil puissant d'analyse statique capable de traiter des programmes SCADE plus complexes. En particulier :

- On pourrait reprendre le fonctionnement d'Astrée : une analyse globale simple avec des intervalles de valeurs, raffinée par des analyses relationnelles sur des petits groupes de variables, avec des domaines abstraits différents selon les applications. On pourrait tenter de développer des heuristiques pour générer automatiquement des groupes de variables susceptibles d'améliorer la précision de l'analyse, en fonction de propriétés à prouver et en s'aidant de la structure du programme.
- Il faudrait implémenter des outils spécifiquement adaptés à l'analyse de programmes de contrôle pour des systèmes continus, par exemple un domaine abstrait à base d'ellipsoïdes (utile pour l'étude de filtres du second ordre).
- Mon prototype ne prend pas du tout en compte la sémantique des entiers machine (overflow), ni la sémantique très complexe des flottants machine. Cela est indispensable pour que les résultats de l'analyse puissent avoir de la valeur quant au comportement effectif du programme en situation réelle.

3.4 Mise en perspective et conclusion

Pendant mon stage j'ai pu travailler sur des techniques peu pratiquées (la combinaison entre domaines abstraits classiques et graphes de décision ; les itérations chaotiques), et documenter partiellement ces techniques. En travaillant sur SCADE 6, j'ai pu mettre en évidence les avantages spécifiques à ce langage, qui n'avaient pas pu être exploités dans d'autres études de l'analyse statique de programmes synchrones telles que [5] ou [7]. De nombreuses autres techniques, décrites dans [2], pourraient maintenant être utilisées en complément de mon approche, afin de la rendre utilisable dans des cas réels.

Bibliographie

[1] Fadi A. Aloul, Igor L. Markov et Karem A. Sakallah. Force: a fast and easy-to-implement variable-ordering heuristic. Dans *ACM Great Lakes Symposium on VLSI*, pages 116–119. ACM, 2003.

[2] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux et X. Rival. A static analyzer for large safety-critical software. Dans *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207. San Diego, California, USA, June 7–14 2003. ACM Press.

[3] Jean-Louis Colaço, Bruno Pagano et Marc Pouzet. A Conservative Extension of Synchronous Data-flow with State Machines. Dans *ACM International Conference on Embedded Software (EMSOFT'05)*. Jersey city, New Jersey, USA, September 2005.

[4] P. Cousot et R. Cousot. Systematic design of program analysis frameworks. Dans *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282. San Antonio, Texas, 1979. ACM Press, New York, NY.

[5] N. Halbwachs. About synchronous programming and abstract interpretation. Dans B. LeCharlier, éditeur, *International Symposium on Static Analysis, SAS'94*. Namur (belgium), September 1994. LNCS 864, Springer Verlag.

[6] N. Halbwachs, F. Lagnier et C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language lustre. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, , September 1992.

[7] B. Jeannet, N. Halbwachs et P. Raymond. Dynamic partitioning in analyses of numerical properties. Dans *Static Analysis Symposium, SAS'99*. Venezia (Italy), sep 1999.

[8] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation (HOSC)*, 19(1):31–100, 2006. `Http://www.di.ens.fr/~mine/publi/article-mine-HOSC06.pdf`.

Annexes

1 Exemple d'analyse de programme SCADE

Reprenons notre exemple canonique :

```
node updown() returns(x: int)
var up: bool;
let
  up = true -> (if pre up then pre x < 10 else pre x <= 0);
  x = 0 -> (if up then pre x + 1 else pre x - 1);
tel
```

Une analyse sans disjonction de cas se ferait comme suit :

itération	**init**	**pre up**	**pre x**	**up**	**x**
1	tt	$\bot$	$\bot$	tt	0
2	$\top$	tt	0	tt	$[0,1]$
3	$\top$	tt	$[0,1]$	tt	$[0,2]$
4	$\top$	tt	$[0,+\infty]$	$\top$	$[-1,+\infty]$
5	$\top$	$\top$	$[-1,+\infty]$	$\top$	$[-2,+\infty]$
6	$\top$	$\top$	$\top$	$\top$	$\top$
7	$\top$	$\top$	$\top$	$\top$	$\top$

Tableau 1. Analyse sans disjonction de cas de l'exemple `updown`. À partir de la 4e itération, les informations ne sont plus pertinentes ; à la fin nous n'avons plus aucune information.

Une analyse plus fine, par itérations chaotiques en considérant $\mathbb{X}_d = \{\text{init}, \text{pre up}, \text{up}\}$ donne :

itération		**init**	**pre up**	**up**	**pre x**	**x**
1	$\star$	tt	$\bot$	tt	$\bot$	0
2		tt	$\bot$	tt	$\bot$	0
	$\star$	ff	tt	tt	0	1
3		tt	$\bot$	tt	$\bot$	0
		ff	tt	tt	$[0,9]$	$[1,10]$
	$\star$	ff	tt	ff	10	9
4		tt	$\bot$	tt	$\bot$	0
		ff	tt	tt	$[0,9]$	$[1,10]$
		ff	tt	ff	10	9
	$\star$	ff	ff	ff	9	8
5		tt	$\bot$	tt	$\bot$	0
		ff	tt	tt	$[0,9]$	$[1,10]$
		ff	tt	ff	10	9
		ff	ff	ff	$[1,9]$	$[0,8]$
	$\star$	ff	ff	tt	0	1
6		tt	$\bot$	tt	$\bot$	0
		ff	tt	tt	$[0,9]$	$[1,10]$
		ff	tt	ff	10	9
		ff	ff	ff	$[1,9]$	$[0,8]$
		ff	ff	tt	0	1

Tableau 2. Analyse avec disjonctions de cas de l'exemple `updown`. La valeur abstraite de la 6e itération est un point fixe pour lequel $x \in [0,10]$ est vrai : la propriété est prouvée.