FRACTION : analyseur de propriétés par des techniques de résolution de contraintes

Fabrice Parrennes

SURLOG S.A. VELIZY PLUS - Bat E
1 bis, rue du petit-clamart
78140 Velizy-Villacoublay
Fabrice.Parrennes@surlog.com
LIP6, Thème SPI
Fabrice.Parrennes@lip6.fr

19 juin 2001

Résumé

Ce papier décrit FRACTION, un outil de vérification de propriétés de programme par utilisation de contraintes. L'utilisation de techniques liées aux contraintes pour vérifier des propriétés n'est pas véritablement nouvelle [11, 12]. L'originalité de l'approche exposée est d'avoir défini une traduction d'un langage de style impératif en une série de systèmes de contraintes. Ce sont les algorithmes de résolution de contraintes qui permettent de valider ou non les propriétés recherchées. Cette traduction passe par une étape intermédiaire où l'on retrouve l'aspect fonctionnel des programmes.
L'utilisation du langage OBJECTIVE CAML a permis de réaliser facilement un prototype de la méthode dont l'architecture rappelle celle d'un compilateur. La définition d'un *solveur* de contraintes modulaire rend l'outil paramétrable par rapport à des modules de résolution de contraintes sur des domaines différents (les entiers et les booléens par exemple).

1 Introduction

La problématique de la vérification de propriétés d'un programme informatique peut se ramener à valider des prédicats sur l'ensemble des états possibles du programme. Il est assez facile de vérifier qu'une propriété est vraie lorsque le programme informatique est exécuté dans un environnement particulier. Il suffit d'évaluer le programme pour l'environnement spécifié puis de vérifier la propriété sur le résultat de l'exécution du programme. La propriété n'est alors valide que sous l'hypothèse décrite par l'environnement dans lequel est évalué le programme. Il est nettement plus difficile de vérifier la propriété lorsque nous ne connaissons pas l'ensemble des valeurs possibles des variables (par exemple lorsque l'environnement d'exécution associe plusieurs valeurs à une même variable).

La vérification de propriétés peut se faire par interprétation abstraite, c'est-à-dire en donnant une vue abstraite des domaines des données. Une interprétation abstraite intéressante pour la validation de propriétés est définie en découpant un domaine de données en intervalles[7]. L'image des différentes opérations réalisées par ces données est fournie par une sémantique abstraite basée sur une sémantique concrète d'évaluation.

Il reste alors à calculer sur ces intervalles. Un intervalle peut être représenté par une contrainte, une propriété est ainsi représentée par un système de contraintes. Il faut alors utiliser des algorithmes de réduction de domaine, pour vérifier qu'un ensemble de contraintes est satisfiable et ainsi exhiber une ou l'ensemble des solutions de la résolution.

Une approche semblable a déjà été utilisée pour déterminer des jeux de test de programmes [12] ou pour vérifier des programmes par "model checking" avec contraintes [11].

Ce papier présente un analyseur de propriétés, appelé FRACTION, qui permet d'obtenir une abstraction sous forme de contraintes d'un programme écrit dans un langage impératif. L'abstraction de ce programme passe par une étape intermédiaire où est retrouvée l'aspect fonctionnel de celui-ci. Les systèmes de contraintes associés au programme de départ s'obtiennent par traduction de cet aspect fonctionnel du programme.

A partir des systèmes de contraintes et des propriétés recherchées, un module de résolution permet de valider ou non les différentes propositions. La technique de traduction permet de calculer plus finement les différents domaines de données associés aux variables car le langage des contraintes est beaucoup plus riche que la simple abstraction des données en intervalle.

L'outil FRACTION a été implémenté en utilisant le langage OBJECTIVE CAML[14, 18]. La définition de phases de traduction bien distinctes a permis l'implémentation d'une structure modulaire proche des différentes phases d'un compilateur. La modularité et les foncteurs d'OBJECTIVE CAML ont permis d'obtenir un paramétrage du module de résolution de contraintes par des modules spécifiques aux domaines de calcul.

Cet outil FRACTION[1] est destiné à établir la communication entre les outils développés par SURLOG S.A., à savoir AGFL®[2] et RCSPL®[3]. L'outil AGFL® permet d'obtenir une vision fonctionnelle d'un programme écrit dans un langage de style impératif[3]. L'outil RCSPL® est un outil permettant la réalisation d'analyses de sûreté du logiciel de type AMDE[4].

Cet article présente la problématique, la technique de vérification par contraintes et la partie relative au codage, dans le langage OBJECTIVE CAML de la méthode d'analyse de propriétés par utilisation de techniques de contraintes.

2 Présentation de la méthode

2.1 Présentation de la problématique

Nous cherchons à définir une approximation des programmes sur des domaines de valeurs, pour cela un certain nombre de définitions sont introduites.

Soit $\mathcal{V}ar$ un ensemble de variables et $\mathcal{V}al$ un ensemble de valeurs (les entiers, les booléens, les réels par exemple). $\mathcal{P}$ dénote l'ensemble des programmes d'un certain langage L et $\mathcal{Q}$ représente l'ensemble des prédicats possibles sur des expressions définies à l'aide des variables $\mathcal{V}ar$ et des opérateurs de L.
Soit $\mathcal{E}nv$ l'ensemble des couples $(\mathcal{V}ar \times \mathcal{V}al)$ ($i.e$ $\mathcal{E}nv = \wp(\mathcal{V}ar \times \mathcal{V}al)$) définissant la notion d'environnement d'évaluation d'un programme $\mathcal{P}$ et $\mathcal{E}[\![P]\!]_e = e'$ la fonction sémantique d'évaluation d'un programme dont la signature est :

$$\mathcal{E} : \mathcal{P} \to \mathcal{E}nv \to \mathcal{E}nv$$

$\mathcal{E}[\![P]\!]_e = e'$ signifie que e' est l'environnement résultant de l'évaluation de P dans e.
$\mathcal{E}_Q : \mathcal{Q} \to \mathcal{E}nv \to \{true; false\}$ représente une procédure de décision pour l'ensemble des prédicats $\mathcal{Q}$.
Vérifier qu'une propriété Q est vraie à la suite de l'évaluation d'un programme P dans un environnement e revient à vérifier que :

$$\mathcal{E}[\![P]\!]_e = e' \ et \ \mathcal{E}_Q[\![Q]\!]_{e'} = true$$

Vérifier que la propriété Q est toujours vraie, c'est vérifier que :

$$\forall e \in \mathcal{E}nv \ \mathcal{E}[\![P]\!]_e = e' \ et \ \mathcal{E}_Q[\![Q]\!]_{e'} = true$$

2.1.1 Interprétation abstraite de programmes

L'interprétation abstraite de programmes permet de calculer des propriétés bien choisies sur des programmes informatiques [7, 17].

On définit un certain domaine abstrait $\mathcal{V}al^\sharp$ pour représenter l'ensemble $\mathcal{V}al$ des valeurs possibles d'une variable[4] et ce domaine est en général bien plus simple que ce dernier[5].

[1]Ce travail fait partie des travaux réalisés dans le cadre d'une thèse CIFRE entre la société SURLOG S.A. et le Laboratoire d'Informatique de Paris 6 `http ://www-spi.lip6.fr`.
[2]AGFL® : Analyseur par Graphe Fonctionnel du Logiciel.
[3]RCSPL® Recherche des Combinaisons Significatives de Pannes du Logiciel.
[4]Le domaine doit respecter des critères bien particuliers comme celui d'être un treillis complet [10].
[5]Une interprétation abstraite bien connue est le signe des variables d'un programme. Les entiers sont découpés en trois groupes : les entiers négatifs, zéro et les entiers positifs.

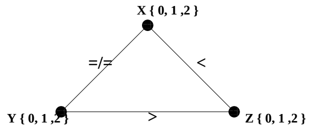

FIG. 1 – Un exemple de contraintes tri-parties.

Pour cela, une certaine abstraction des valeurs concrètes vers les valeurs abstraites est définie sous la forme d'une fonction d'abstraction $\alpha : \mathcal{V}al \rightarrow \mathcal{V}al^\sharp$.
Les propriétés et les fonctions d'évaluation sont aussi interprétées sous une forme abstraite. $\mathcal{E}^\sharp$ et ${\mathcal{E}_Q}^\sharp$ dénotent les fonctions sémantiques d'évaluation abstraite des programmes et des prédicats sur des environnements abstraits $\mathcal{E}nv^\sharp$.

L'interprétation abstraite $P^\sharp$ d'une propriété P sur $\mathcal{V}al$ est correcte si la validité de $P^\sharp$ sur $\mathcal{V}al^\sharp$ implique la validité de P sur $\mathcal{V}al$.

Le traitement des conditionnelles s'effectue de la manière suivante : le calcul est effectué pour les deux branches de la conditionnelle et une fusion des deux résultats s'effectue de manière à garantir la validité de la propriété sur les deux branches.

Cette technique permet de générer des résultats valides quelle que soit l'exécution possible du programme[6] (si aucune information relativement à une variable n'est calculée alors son domaine est l'ensemble des valeurs possibles). Il est plus difficile d'obtenir de l'information lorsque les prédicats des programmes ou des conditionnelles sont de la forme $x < y$ (par exemple si $x = 1..100$ et $y = 1..100$ on ne sait pas dire si la condition est vraie ou fausse).

La technique par contraintes utilise un traitement différent des conditionnelles. L'ensemble des branches possibles issues de l'exécution du programme est intégré dans le modèle fonctionnel. La traduction des contraintes s'effectuant sur ce modèle garantit la validité des prédicats sur les exécutions possibles du programme.

2.1.2 Programmation logique par contraintes

La programmation logique est un paradigme de programmation dont les fondements ont été posés dans le début des années 1970.

La programmation logique par contraintes intègre des domaines de calculs particuliers comme les domaines réels, les domaines finis ou les formules booléennes (on étend le domaine de discours), pour définir sur ces ensembles une notion de contrainte représentée par des clauses de Horn[7]. Résoudre un programme logique par contraintes c'est principalement résoudre les différentes contraintes en utilisant des solveurs spécifiques aux différents domaines de définition [5, 6, 13, 15].

Exemple de technique de résolution sur les entiers Les techniques de calcul de résolution de contraintes sur les domaines entiers peuvent être basées sur la notion d'arc consistance [15] : à chaque étape du calcul, nous essayons de vérifier *l'ensemble des contraintes*, entraînant la réduction des domaines des différentes variables sur lesquelles portent les contraintes. Soit l'exemple de la figure 1 : nous voulons satisfaire pour les trois variables X, Y, Z (de domaine $[0..2]$) les inéquations suivantes $X \neq Y \quad X < Z$ et $Y > Z$. Un exemple de résolution de cet exemple est le suivant :

	$Y \neq X$	$Y > Z$	$X < Z$	*resultat*
$Y = 0$	$X = 1, 2$	$Z = \emptyset$		*impossible*
$Y = 1$	$X = 0, 2$	$Z = 0$	$X = \emptyset$	*impossible*
$Y = 2$	$X = 0, 1$	$Z = 0, 1$	$X = 0$	$X = 0, Y = 2, Z = 1$

[6]On parle aussi d'analyse statique des programmes.
[7]Les clauses de Horn sont un sous-ensemble des formules logiques du premier ordre.

Sémantique des langages	Résolution de contraintes
un composant logiciel	un ensemble de systèmes de contraintes sur les entrées/sorties du composant(1)
un environnement d'exécution	un système de contraintes sur les entrées du logiciel (2)
analyser un composant	résoudre les systèmes de contraintes (1)+(2)
analyser deux composants logiciels en séquence	fusionner leurs systèmes de contraintes et résoudre
une propriété = un prédicat	un système de contraintes sur les entrées/sorties du logiciel
vérifier une propriété	fusionner les systèmes et la propriété puis résoudre

TAB. 1 – Liens entre analyse par contraintes et sémantique de langage

Dans le système CLP(BNR) [5], les auteurs utilisent des approximations et créent des fonctions de rétrécissements et d'élargissements (*widening and narrowing)* des contraintes pour réduire, étape par étape, les différents domaines de valeurs.

Soit $\mathcal{C}$ l'ensemble de toutes les contraintes basées sur les domaines $\mathcal{V}al$ et les variables $\mathcal{V}ar$. Soit $\mathcal{S}$ l'ensemble des systèmes de contraintes (ce sont des collections de contraintes $\mathcal{C}$). Soit $\circ : \mathcal{S} \rightarrow \mathcal{S} \rightarrow \mathcal{S}$ un opérateur de fusion de systèmes de contraintes. Un système $S \in \mathcal{S}$ est dit inconsistant ou insatisfiable s'il contient une contrainte C de domaine vide[8].

L'utilisation de contraintes permet de pouvoir parler de l'expression $X < Y$ lorsque $X \in [1..100]$ et $Y \in [1..100]$ par l'ajout de la contrainte $X < Y$ aux deux ensembles de valeurs définis par l'appartenance de X et Y à $[1..100]$.

Le point précédent, allié aux techniques d'interprétation abstraite de programmes, nous donne le schéma global de l'analyse de propriétés par contraintes. Les opérateurs abstraits sont dans notre technique les opérations sur les systèmes de contraintes.

2.2 Méthode de la vérification de propriétés par contraintes

A partir de notre problématique initiale, nous essayons d'établir des liens entre la sémantique des programmes et l'utilisation de contraintes, le tableau 1 présente une synthèse des opérations que nous aimerions définir.

Nous essayons de tirer profit des contraintes pour interpréter finement les différentes expressions présentes dans les programmes. C'est à dire définir une interprétation des programmes par des systèmes de contraintes et utiliser les techniques de résolution pour obtenir la procédure de vérification des propriétés.

Une méthode en quatre phases a été définie pour traduire et résoudre des propriétés par utilisation de contraintes.

1. lecture du programme, vérification de type et calcul du flot des variables ;
2. modélisation du programme sous la forme d'un modèle fonctionnel ;
3. traduction du modèle fonctionnel en systèmes de contraintes ;
4. composition des systèmes de contraintes précédents et des propriétés à vérifier.

La figure 2, commentée ci-après, montre un exemple de programme écrit dans un langage de type pascal, la deuxième colonne présente la traduction sous forme de contraintes des fonctionnalités du programme.

La contrainte $X \in D$ exprimée par rapport à une variable X représente le domaine de valeur D que peut prendre la variable X. Fonctionnellement, l'exemple de la figure 2 se ramène à trois cas particuliers :

- les prédicats P et Q sont vrais : les variables affectées sont $J \leftarrow I$, $L \leftarrow 2$ et $K \leftarrow K+1$,
- le prédicat P est vrai et le prédicat Q n'est pas vrai, les variables affectées sont $J \leftarrow I$, $L \leftarrow 3$ et $K \leftarrow K+1$,

[8]Cela signifie qu'il n'existe pas de valeur associée à une variable telle que le système de contraintes admette une solution.

```
if (P = true)
 then
   begin
     J:= I;
     if (Q = true)
        then L := 2
        else L := 3;
     K := K+1;
   end
 else
   K := K+2
```

$$\text{Cas 1} \begin{cases} P_0\ in\ \{true\}\ Q_0\ in\ \{true\} \\ J_1\ in\ dom(I_0)\ I_0\ in\ dom(J_1) \\ L_1\ in\ \{2\} \\ K_1\ in\ dom(K_0)+1\ K_0\ in\ dom(K_1)-1 \end{cases}$$

$$\text{Cas 2} \begin{cases} P_0\ in\ \{true\}\ Q_0\ in\ \neg\{true\} \\ J_1\ in\ dom(I_0)\ I_0\ in\ dom(J_1) \\ L_1\ in\ \{3\} \\ K_1\ in\ dom(K_0)+1\ K_0\ in\ dom(K_1)-1 \end{cases}$$

$$\text{Cas 3} \begin{cases} P_0\ in\ \neg\{true\} \\ J_1\ in\ dom(J_0)\ J_0\ in\ dom(J_1) \\ I_1\ in\ dom(I_0)\ I_0\ in\ dom(I_1) \\ L_1\ in\ dom(L_0)\ L_0\ in\ dom(L_1) \\ K_1\ in\ dom(K_0)+2\ K_0\ in\ dom(K_1)-2 \end{cases}$$

FIG. 2 – Exemple de programme et systèmes de contraintes associés

– le prédicat P n'est pas vrai : la seule variable affectée est $K \leftarrow K+2$.

Les systèmes de contraintes utilisés expriment les valeurs des variables avant et après leurs affectations. Pour cela K_0 et K_1 représentent la même variable avant et après l'évaluation du programme. Cette distinction est importante car il n'est possible de définir le domaine d'une variable qu'en fonction de ses valeurs précédentes[9].

A partir de cet exemple nous cherchons à déterminer si, à partir de l'environnement

$$[\langle P, \{true, false\}\rangle\ \langle Q, false\rangle\ \langle K, 10\rangle\ \langle I, \mathbb{N}\rangle\ \langle J, \mathbb{N}\rangle\ \langle L, \mathbb{N}\rangle\ ]$$

il est possible d'obtenir

$$[\langle K, 12\rangle\ \langle L, 10\rangle\ \langle P, ?\rangle\ \langle Q, ?\rangle\langle I, ?\rangle\ \langle J, ?\rangle]^{10}$$

Par rapport à nos définitions *avant* et *après* des variables, ces deux assertions se traduisent en systèmes de contraintes par $S_1 = \{K_0\ in\ \{10\},\ Q\ in\ \{false\}\}$ et $S_2 = \{K_1\ in\ \{12\},\ L_1\ in\ \{10\}\}$ (on remarquera que seules les informations pertinentes apparaissent dans les systèmes de contraintes).

A partir des systèmes de contraintes relatifs à notre propriété, la validité de celle-ci se ramène à calculer si une des trois compositions suivantes est possible (ou satisfiable) :

$$\begin{array}{lcl} S_1 \circ \text{Cas } 1 \circ S_2 & \Rightarrow & Insatisfiable \\ S_1 \circ \text{Cas } 2 \circ S_2 & \Rightarrow & Insatisfiable \\ S_1 \circ \text{Cas } 3 \circ S_2 & \Rightarrow & Satisfiable \end{array}$$

La conclusion de la composition est que la propriété voulue est valide sur le chemin 3 et que la solution est $[\langle P_0, false\rangle\ \langle Q_0, false\rangle\ \langle K_0, 10\rangle\ \langle K_2, 12\rangle\ \langle L_0, 10\rangle\ \langle L_1, 10\rangle\ ...]$.

2.2.1 Traduction vers le modèle fonctionnel

La traduction des programmes en systèmes de contraintes nécessite l'obtention d'une forme intermédiaire appelée modèle fonctionnel. Ce modèle est constitué d'équations où chacune exprime une fonctionnalité du programme considéré (l'ensemble des équations pour un programme représente sa *fonction caractéristique*).

L'intérêt de ce modèle est d'exprimer *à plat* les relations entre les différentes variables du programme sous la forme d'une transition en une étape : différentes affectations d'une même variable sont propagées dans les équations pour ne retenir que l'affectation finale d'une variable par rapport à un prédicat (éventuellement

[9]Le problème est identique à celui de la causalité dans les langages de flot de données comme dans lucid-synchrone par exemple `http ://www-spi.lip6.fr/~pouzet/lucid-synchrone`.

[10]$\langle I, \mathbb{N}\rangle$ et $\langle J, ?\rangle$ signifie qu'aucune valeur particulière n'est donnée à J.

égal à vrai) dépendant des expressions du composant logiciel analysé.

Pour les besoins de la traduction vers les systèmes de contraintes ce modèle fonctionnel est représenté par une forme équationnelle propre à l'outil FRACTION. Cette forme équationnelle correspond au modèle fonctionnel, résultat de l'utilisation de l'outil AGFL©, particularisé à la traduction en contraintes.

Une telle traduction est possible en effectuant deux transformations du programme initial :

- les instructions d'un programme sont traduites en une forme *SSA* pour n'avoir que des occurrences différentes des variables du programme [1, 9] ;
- les différentes affectations des variables sont collectées à travers les fonctions de choix issues de la traduction des conditionnelles de la forme SSA.

Le tableau 2 présente le modèle fonctionnel de l'exemple de la figure 2.

Chemins fonctionnels	condition sur les entrées	Affectations des sorties
Chemin fonctionnel 1	Q_0 = true P_0 = true	$J_1 := I_0$ $L_1 := 2$ $K_1 := K_0 + 1$
Chemin fonctionnel 2	Q_0 <> true P_0 = true	$J_1 := I_0$ $L_1 := 3$ $K_1 := K_0 + 1$
chemin fonctionnel 3	P_0 <> true	$J_1 := J_0$ $L_1 := L_0$ $K_1 := K_0 + 2$

TAB. 2 – Les différents chemins du programme de la figure 2

2.2.2 Traduction vers la forme "contraintes"

La forme "contraintes" de la forme équationnelle est obtenue à partir d'une traduction des expressions du langage. Cette forme équationnelle permet de se défaire des différentes commandes du langage de base. Il en résulte une série de relations entre variables exprimées simplement sous la forme de liens entre une pré-condition et différentes affectations *simultanées* représentée sous la forme d'un chemin fonctionnel du programme. Une condition nécessaire pour la traduction vers les contraintes est ne ne pas avoir d'occurrence de l'opérateur Not et de n'avoir que des formes conjonctives dans les prédicats d'équations (cette condition permet d'obtenir des approximations plus fines des expressions).
L'utilisation d'une forme SSA à la base de la traduction nous permet de s'assurer que deux variables portant le même nom sont bien liées par une notion *avant-après*.

Le tableau 3 représente une partie de la traduction en terme de contraintes des expressions de la forme équationnelle. Les systèmes de contraintes de la figure 2 s'obtiennent par application des règles du tableau 3 sur les chemins du tableau 2.

2.2.3 Résolution des formes "contraintes"

A partir des systèmes de contraintes précédents, un algorithme de type arc-consistance [6, 13, 15] permet de réduire itérativement les systèmes de contraintes.
Si la composition des systèmes de contraintes issus du programme initial et de la propriété recherchée est inconsistante alors la propriété n'est pas valide. Par contre, l'interprétation des expressions étant réalisée de manière supérieure (lors de la traduction d'un $<$ par exemple), si la résolution nous donne une solution, alors il n'est pas toujours possible d'affirmer que la propriété est vraie ou non.

Par rapport aux différents éléments définis précédemment, la technique de modélisation par contraintes

Expression	Traduction en contraintes	Remarques
$X = Y$	$\begin{cases} X\ in\ Dom(Y) \\ Y\ in\ Dom(X) \end{cases}$	Egalité de deux variables
$X \neq Y$	$\begin{cases} X\ in\ \neg Dom(Y) \\ Y\ in\ \neg Dom(X) \end{cases}$	Inégalité de deux variables
$X \leq Y$	$\begin{cases} X\ in\ -\infty..Max(Y) \\ Y\ in\ Min(X)..+\infty(X) \end{cases}$	Inégalité de deux variables
$X < Y$	$\begin{cases} X\ in\ -\infty..(Max(Y)-1) \\ Y\ in\ (Min(X)+1)..+\infty \end{cases}$	Inégalité stricte de deux variables
X := Y	$\{\ X\ in\ Dom(Y)$	Affectation
X := Y + Z	$\{\ X\ in\ Dom(Y) + Dom(Z)$	Addition
X := Y - Z	$\{\ X\ in\ Dom(Y) - Dom(Z)$	Soustraction
X := - Y	$\{\ X\ in\ -Dom(Y)$	Opération unaire

TAB. 3 – Traduction des expressions des équations en contraintes

peut se ramener à lier les éléments suivants :

$$\begin{array}{lcl@{\qquad\qquad}lcl} Val^\sharp & \approx & \mathcal{C} & Q^\sharp & \approx & \mathcal{S} \\ Env^\sharp & \approx & \wp(\mathcal{S}) & \mathcal{E}^\sharp & \approx & \circ \\ {\mathcal{E}_Q}^\sharp & \approx & \text{Satisfiabilité dans } \mathcal{S} & & & \end{array}$$

3 Réalisation d'un prototype

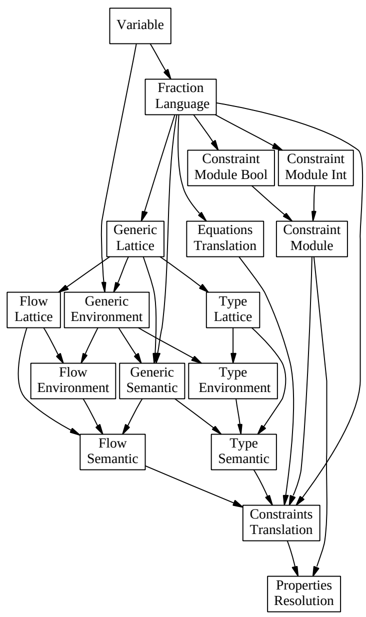

FIG. 3 – Les différents modules de l'analyseur FRACTION

L'utilisation du langage OBJECTIVE CAML[11] et de ses modules a permis d'obtenir un outil modulaire où il est possible d'ajouter rapidement de nouveaux domaines de définition pour la résolution de contraintes. Le schéma de la figure 3 présente l'architecture des modules de l'analyseur.

[11]`http ://caml.inria.fr`

Le prototype FRACTION a besoin d'informations sur les variables présentes dans le programme analysé pour la phase de traduction vers les contraintes. Il était intéressant de regrouper les techniques de calcul de ces informations dans un seul module sous la forme d'une sémantique générique. Pour cela, l'utilisation des foncteurs de OBJECTIVE CAML a permis de définir une architecture basée sur les valeurs possibles des variables. Ces valeurs (pouvant être différentes selon l'analyse désirée) sont regroupées dans un seul module treillis.
La définition d'un module général de contraintes fondée sur des solveurs de domaines différents (`Constraint module Int` et `constraint module Bool` permet aussi une plus grande souplesse lors de la résolution de contraintes.

3.1 Lecture et typage du programme

La première phase de la traduction vers les contraintes est de lire et de typer les programmes. L'information de type est importante car c'est elle qui permet d'orienter la traduction vers les différents domaines de résolution de contraintes. A partir d'un module de sémantique générique paramétré par un treillis de valeurs (voir [8]) nous pouvons évaluer les différents types du programme et calculer de l'information sur les entrées/sorties du programme.

3.1.1 Le module de variable

Le module `Variable_signature` exprime les opérations nécessaires à la manipulation des variables d'un programme, sa signature est la suivante :

```
module type Variable_signature =
  sig
    type var
    val is_equal        : var -> var -> bool
    val next_of_var     : var -> var
    val new_var         : unit -> var
    ...
  end
```

Il est important de différencier les variables à droite et à gauche des affectations, ou encore en entrée et en sortie de programme. Pour cela la différence entre une variable v et la variable `next_of_var(v)` réside dans le fait que `next_of_var(v)` désigne la valeur suivante de la variable v[12].

Implémentation L'implémentation du module est effectuée en définissant le type `var` comme un couple $(string * int)$. Ainsi la fonction `next_of_var (s,i)` se résume à renvoyer le couple $(s, i + 1)$.

3.1.2 Le module de langage

Le module de langage paramétré par le module de variable (voir figure 3) représente le langage de base de notre application. Ce module est importé par tous les composants de l'analyseur et par les modules de traduction. Le type le plus important de ce module est le type `expr` car c'est celui qui est la base de la traduction en contraintes. Le type `expr` permet aussi de définir des propriétés sous la forme de prédicats sur les variables du programme (c'est notre langage de propriété).

```
module type Language_fraction =
  sig
  module V : Variable_signature
  type commands =
        Skip
      | Affect        of (V.var * expr)
      | Seq           of (commands * commands)
      | While         of (expr * commands)
```

[12]Il faut comprendre par cela que l'affectation `v := v+1` où `v` est une variable équivaut à `next_of_var(v) := v+1`.

```
      | Ifthenelse_c  of (expr * commands * commands)
  and  expr =
        Const        of values
      | Ident        of V.var
      | Add          of (expr * expr)
      | Sub          of (expr * expr)
      | Equal        of (expr * expr)
      | Inf          of (expr * expr)
      | Or           of (expr * expr)
      ...
  and  values =
        Int  of int
      | Bool of bool

  val fraction_true  : expr
  val fraction_false : expr
  val equationnal_simplification : expr -> expr
  val normalize_expr_not         : expr -> expr
  val commands_to_equations      : commands -> (expr * ((V.var * expr) list)) list
  ...
```

Implémentation Le codage du module de langage est classique pour un langage de style impératif. La fonction `commands_to_equations` implémente la technique de traduction par l'intermédiaire d'une forme SSA des programmes vers la forme équationnelle.

3.1.3 Le module de sémantique générique

Le module de sémantique générique est basé sur une sémantique standard de type opérationnelle du langage de base [16]. Le typage du programme et la détermination du flux des variables s'effectuent par l'intermédiaire de ce module paramétré par un module treillis de valeurs et un module d'environnement abstrait.
L'évaluation du programme pour un treillis donné est une interprétation abstraite simple des expressions et des commandes du langage [7, 8, 17].

Le module générique des treillis Le module des treillis représente les opérations élémentaires sur les éléments que nous voulons calculer. Il faut poser la traduction des valeurs de bases du langage dans le treillis des valeurs (ou encore écrire la fonction d'abstraction α), et donner deux fonctions `is_true_of_lat` et `is_false_of_lat` pour l'évaluation des conditionnelles et des boucles. L'intérêt de ces deux fonctions est de pouvoir valider ou non un prédicat. Lorsque celui-ci est indéterminable alors nous utilisons les techniques de valeurs supérieures dans le treillis pour garantir la validité des résultats.

```
module type Lattice_algebra_signature =
  sig
    module L : Language_fraction
    type lat

    val bot : unit -> lat      (* bottom value                     *)
    val top : unit -> lat      (* top value                        *)
    val eq  : lat->lat->bool   (* egality test on lattice elements  *)
    val leq : lat->lat->bool   (* order function on lattice         *)
    val lat_upper : lat -> lat -> lat (* upper bound *)
    val val_of_constant  : L.values -> lat
    val is_true_of_lat   : lat -> bool (** Is the value of lat is true ?  **)
    val is_false_of_lat  : lat -> bool (** Is the value of lat is false ? **)
```

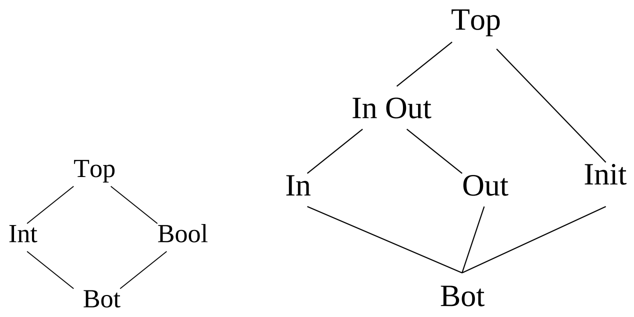

FIG. 4 – Le treillis des types et du flot de données

```
    val lat_Add     : lat -> lat -> lat
    val lat_Sub     : lat -> lat -> lat
    val lat_Equal   : lat -> lat -> lat
    ...
  end
```

Le treillis des signes Le treillis des signes présenté dans la figure 4 permet de *typer* les variables présentes dans les expressions et les commandes.

Implémentation Voici un extrait du module implémentant le treillis des types. La fonction de valeur supérieure utilise le treillis donné à la figure 4, les opérations arithmétiques et logiques s'écrivent naturellement selon le type de donnée attendu.

```
module Lattice_algebra_signature_type (Lang : Language_fraction)
             : (Lattice_algebra_signature with module L = Lang) =
  struct
    ...
    type lat = Bot | Int | Bool | Top

    let val_of_constant c =
       match c with
         Lang.Int _ -> Int
       | Lang.Bool _ -> Bool
    let lat_Add l1 l2 =
       match (l1,l2) with
         (Int,Int) -> Int
      | (Bot,Int) -> Int
      | (Int,Bot) -> Int
      | (Bot,Bot) -> Bot
      | (_,_)     -> Top
      ...
end
```

Le treillis des entrées/sorties Les entrées et sorties sont utilisées pour vérifier le flux entre différentes parties de programme. L'ajout d'une notion de procédure permet d'utiliser le flux pour lier les valeurs des variables aux entrées des procédures : les variables sont nommées et il faut alors mettre à jour les noms des variables pour la procédure considérée.
La figure 4 présente le treillis des valeurs entrées/sorties. `In` signifie que la variable est en entrée des programmes, `Out` signifie que la variable est en sortie (ce qui veut dire que sa valeur en sortie ne dépend pas

de sa valeur en entrée. In_Out correspond à une variable lue et écrite dans le corps du programme. Init est une valeur utilisée lorsque la variable est simplement affectée dans le corps d'un programme.

Le module générique d'environnements A partir de la définition d'un treillis de valeurs, une notion générique d'environnement est définie. Elle associe aux variables des valeurs du treillis paramètre.

```
module type Env_algebra_signature =
  sig
    module V   : Variable_signature
    module Lat : Lattice_algebra_signature
    type env
    val empty         : unit -> env
    val env_get       : env -> V.var -> Lat.lat
    val env_put       : env -> V.var -> Lat.lat -> env
    val upper_env_put : env -> V.var -> Lat.lat -> env
    val upper_env     : env -> env -> env
    ...
end
```

Implémentation Une liste (V.var * Lat.lat) list permet d'utiliser les fonctions d'association du langage Objective Caml. On remarquera que la fonction upper_env permet de fusionner deux environnements par application de l'opérateur supérieur dans le treillis des valeurs lat.
L'environnement de flot et de type s'obtiennent en utilisant le module de treillis des flots, le module de type et le module d'environnement générique d'environnement.

Le module de sémantique générique Le module de sémantique générique se paramètre par trois modules : un module de langage *Lang*, un treillis de valeurs *Lattice* et un environnement *Env* associant des variables du langage *Lang* aux valeurs du treillis *Lattice*. A partir de la définition du langage, et d'une sémantique de type opérationnelle [16], les programmes sont évalués en utilisant les opérations définies dans le treillis *Lat*. La signature du module de sémantique générique est la suivante :

```
module type Generic_forward_semantic_signature =
    sig
      module Lang    : Language_fraction
      module Lattice : (Lattice_algebra_signature with module L = Lang)
      module Env     : (Env_algebra_signature with module V   = Lang.V
                                              and  module Lat = Lattice)
      val generic_forward_expr     : Env.env -> Lang.expr -> (Lattice.lat * Env.env)
      val generic_forward_commands : Lang.commands -> Env.env -> Env.env
  end
```

Implémentation L'implémentation du module est basée sur une sémantique standard du langage. La définition des fonctions is_true_of_lat et is_false_of_lat du treillis des valeurs et la fonction upper_env du module d'environnement permettent de s'assurer que les résultats obtenus sont valides quelle que soit l'évaluation du programme.

3.1.4 Forme équationnelle du programme

La forme équationnelle du programme est obtenue en deux phases de réécriture des commandes de base. La fonction de traduction est présente dans le module de langage afin de ne pas exporter la structure interne des commandes.
Le module représentant la forme équationnelle des programmes noté Language_fraction_equation_signature utilise la signature suivante :

```
module type Language_fraction_equation_signature =
sig
  module Var  : Variable_signature
  module Lang : (Language_fraction with module V = Var)
  type recursive_affectation
  and  fraction_path
  and  fraction_path_program = fraction_path list
  val name_of_path         : fraction_path -> string
  val precond_of_path      : fraction_path -> Lang.expr
  val affects_list_of_path : fraction_path -> (Var.var * Lang.expr) list
  val path_of_commands                 : Lang.commands -> fraction_path_program
  val normalize_path                   : fraction_path -> fraction_path_program
  val normalized_path_of_program_path : fraction_path_program -> fraction_path_program
  ...
```

La forme équationnelle des programmes doit respecter certaines propriétés. Par exemple les expressions ne doivent pas contenir de construction *not*. Pour cela une série de règles de réécriture est utilisée pour normaliser les expressions (c'est la fonction `normalize_expr_not` du module de langage [8]).

Le terme chemin (path) désigne une fonctionnalité du programme (c'est une branche de l'arbre d'exécution du programme). Il peut exister des branches non accessibles, pour cela une fonction de normalisation `normalize_path` permet de retirer ces branches (le prédicat est évalué partiellement).

3.1.5 Traduction vers la forme "contraintes"

A partir de la forme normale équationnelle, un module de traduction vers les systèmes de contraintes est défini. Ce module est basé sur la traduction d'une expression arithmétique ou logique du langage de base. L'utilisation d'une forme *atomique* X *in* D de contraintes [6] permet d'exprimer des équations sous la forme d'une collection de contraintes. Le tableau 3 présente des règles de traduction d'expressions en système de contraintes.

Il est possible d'ajouter des variables aux expressions afin de faciliter la traduction. Ainsi l'expression $(X+2)=(Y+3)$ peut se réécrire en $U=(X+2); V=(Y+3); U=V$. Cette opération est semblable à une mise en forme *code 3 adresses* des expressions [1].

La signature du module de traduction utilise le module d'équation et le module de résolution de contraintes.

```
module type Language_fraction_constraint =
  sig
    module Eqn : Language_fraction_equation_signature
    module Const_module : (Constraint_signature_module with module Lang = Eqn.Lan
g)
    type fraction_store_path
    and  fraction_store_program = fraction_store_path list
    val store_path_of_name_store : string -> Const_module.store -> fraction_store_path
    val store_of_store_path : fraction_store_path -> Const_module.store
    val name_of_store_path  : fraction_store_path -> string
    val store_of_path : Eqn.fraction_path -> Const_module.store
    ...
  end
```

A partir des types des expressions (obtenus à partir du module de typage), les systèmes de contraintes sont générés par utilisation de la fonction de traduction présente dans le module de contraintes.

3.1.6 Le module de contraintes

Le module de contraintes permet de regrouper dans une seule construction différentes techniques de résolution sur des domaines différents. La signature du module de contrainte est la suivante :

```
module type Constraint_signature_module =
```

```
  sig
    module Lang       : Language_fraction
    module Const_int  : (Constraint_signature_type_int with module Var_int = Lang
.V and module Lang_int = Lang)
    module Const_bool : (Constraint_signature_type_bool with module Var_bool = La
ng.V and module Lang_bool = Lang)

    type contr
    type range
    type tdef
    type store
    ...
    val store_from_int  : Lang.expr -> store
    val store_from_bool : Lang.expr -> store
    ...
    val tell         : contr -> store -> store
    val fusionne     : store -> store -> store
    val elim_store   : store -> bool
    ...
  end
```

Implémentation L'implémentation du module de contraintes s'effectue en différenciant les types de contraintes utilisés en interne dans le module. Lors de la résolution ou lors de la traduction l'information de type sur les contraintes permet d'appeler la fonction correspondant au domaine défini par le type.

3.2 Conclusion sur l'implémentation du prototype

L'implémentation du prototype a été facilitée par l'utilisation des *lexers* et *parsers* fournis avec Objective Caml. Le prototype fraction obtenu permet l'analyse de programmes impératifs écrits dans un langage proche du langage pascal. D'autres analyses ont été définies à partir du module de sémantique générique (comme l'analyse par intervalles) et différentes techniques de résolution de contraintes ont été utilisées par implémentation de modules spécifiques de résolution.
Un travail en cours est d'augmenter la puissance d'expression du langage de base pour intégrer des constructions "procédures" et "fonctions" ainsi que d'ajouter des types énumérés sur les variables des programmes.

4 Conclusion

Nous avons présenté une méthode de validation de propriétés par utilisation de techniques liées à la résolution de contraintes. Cette technique a l'avantage de se servir de la puissance de résolution des algorithmes de contraintes, et peut par l'intermédiaire d'un module *contraintes* s'interfacer avec des outils existants de résolution de contraintes.

L'utilisation du langage Objective Caml a permis de définir rapidement une architecture *compilateur* de notre outil de vérification de propriétés par contraintes. La facilité pour les langages de la famille ML de traiter les structures de données définies récursivement est un atout essentiel pour l'implémentation du prototype. De plus l'utilisation des modules du langage a permis de paramétrer la résolution et la traduction des expressions vers les systèmes de contraintes.

La forme équationnelle peut être étendue lorsque le langage de base contient des procédures ou fonctions. Le programme est alors décomposé en différents sous-programmes sur lesquels s'applique la technique de traduction par contraintes. Cette décomposition permet de réduire la taille de la forme équationnelle (il est plus simple de traiter deux chemins suivis de deux chemins que quatre chemins simultanément).

La traduction des contraintes n'a été effectuée pour l'instant que pour un sous ensemble du langage de base ne contenant pas de construction (`while`). Une piste de recherche est d'utiliser les techniques semblables à celles utilisées dans l'interprétation abstraite pour obtenir une évaluation supérieure des valeurs des variables en sortie de boucle. Cela peut modifier les techniques de contraintes par ajout d'une construction point fixe sur la résolution de contraintes. L'implémentation du prototype devra alors intégrer ces différentes modifications par ajout de constructions dans le langage des équations et lors de la phase de résolution de contraintes.

L'analyseur a été utilisé pour valider les propriétés d'un régulateur de vitesse [2]. L'apport des contraintes permet aussi de mieux caractériser les différentes opérations réalisées dans un système logiciel par rapport à une interprétation par intervalle des domaines de valeurs.
L'intégration de techniques de contraintes dans l'outil RCSPL© permettra ainsi une modélisation fine des composants logiciels. L'apport de l'outil FRACTION est aussi de donner une forme normalisée des programmes, facilitant ainsi le travail lors des analyses de sûreté du logiciel.

Références

[1] AHO, A., SETHI, R., ULLMAN, J *Compilers Principles, techniques and tools.* Addison-Wesley Publishing Compagny, Inc, 1986.

[2] Joanne M. ATLEE and John GANNON, *State-Based Model Checking of Event-Driven System Requirement.* IEEE Transaction on Software Engineering, 1993.

[3] P. AYRAULT T. HARDIN M. GUESDON *Méthodologie de développement d'un outil d'évaluation de la sûreté du logiciel en langage OCaml*, JFLA 2000.

[4] B. LE TRUNG M-C. MONÉGIER DU SORBIER ,B. SOUBIÈS, O. ELSENSOHN, *Logiciels de sécurité*,Veille technologique, 1995.

[5] Frédéric. BENHAMOU, William J. OLDER, *Applying Interval Arithmetic to Real, Integer, and Boolean Constraints*, ournal of Logic Programming, Volume 32, 1997.

[6] Philippe CODOGNET and Daniel DIAZ, *Compiling Constraints in clp(FD).* J. Logic Programming, 1996.

[7] Patrick COUSOT and Radhia COUSOT, *Static determination of dynamic properties of programs.* Proceedings of the Second International Symposium on Programming, 1976.

[8] Patrick COUSOT, *The Calculational Design of a Generic Abstract Interpreter.* M. Broy and R. Steinbrüggen (eds.), Calculational System Design NATO ASI Series F. Amsterdam : IOS Press, 1999.

[9] Ron Cytron Jeanne Ferrante Barry K. Rosen Mark N. Wegman and F. Kenneth Zadeck, *An efficient method of computing static single assignment form.* 16th ACM Symposium on Principles of Programming Languages, 1989.

[10] B.A. DAVEY and H.A. PRIESTLEY, *Introduction to Lattices and Order.* Cambridge Mathematical Textbooks, Cambridge university press, 1990.

[11] Giorgio DELZANNO, Andreas PODELSKI *Model Checking in CLP.* TACAS 1999.

[12] Arnaud GOTLIEB, Bernard BOTELLA, Michel RUEHER, *Automatic Test Data Generation using Constraint Solving Techniques.* International Symposium on Software Testing and Analysis, 1998.

[13] Joxan JAFFAR and Michael MAHER, *Constraint Logic Programming : A Survey.* Journal of Logic Programming, 1994.

[14] Xavier LEROY, *The Objective Caml system (release 3.00) Documentation and user's manual.* `http ://caml.inria.fr/ocaml/htmlman` 2000.

[15] Alan K. MACKWORTH, *Consistency in Networks of Relations.* Artificial Intelligence, 1977.

[16] G. D. PLOTKIN, *A structural approach to operational semantics.* Aarhus University, 1981.

[17] David A. SCHMIDT, *Abstract Interpretation of Small-Step Semantics.* 5th LOMAPS Workshop on Analysis and Verification of Multiple-Agent Languages, LNCS 1192, 1997.

[18] Pierre WEIS et Xavier LEROY, *Le langage Caml* (2^e édition). Dunod, 1999.