IRISA

INSTITUT DE RECHERCHE
EN INFORMATIQUE
ET SYSTEMES ALEATOIRES
Campus Universitaire de Beaulieu
35042 RENNES CEDEX FRANCE
Tél.: 02 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 02 99 84 71 71

IFSIC
INSTITUT DE FORMATION SUPERIEURE en INFORMATIQUE et COMMUNICATION

Réalisation d'un analyseur polyèdrique pour un langage synchrone

Rapport de stage de maîtrise

Christelle LECOMTE

Projet LANDE

Septembre 2000

Encadreur :
Thomas JENSEN

Table des matières

Introduction

Ce stage a consisté à réaliser un analyseur pour un sous-ensemble du langage synchrone SIGNAL suivant l'analyse statique définie dans l'article "Polyhedral Analysis for Synchronous Languages" de Frédéric Besson, Thomas Jensen et Jean-Pierre Talpin [1]. Pour cela, j'ai utilisé SmartTools, un générateur d'environnement pour des langages de programmation. Ce logiciel m'a permis de définir une syntaxe abstraite pour un sous-langage de SIGNAL, ainsi que l'utilisation des concepts d'AST(Abstract Syntax Tree) et de Design Pattern Visitors. Cet analyseur produit des fichiers qui seront lus par REQS. Ce dernier permet de résoudre des systèmes d'équations récursives produites dans le cadre d'analyses statiques. La seconde partie de mon stage a consisté à implémenter un nouvel opérateur d'élargissement sur les polyèdres. Il a été défini dans l'article cité plus haut [1].

Dans un premier temps, je vais décrire le langage synchrone SIGNAL. Ce langage permet de développer des systèmes réactifs, dans lesquels les notions de temps et de synchronisation sont très importantes. Ensuite, je présenterai le générateur d'environnement de programmation SmartTools, développé par le projet Oasis à Sophia Antipolis. Puis, je décrirai l'analyse polyèdrique dont le but est de déduire des propriétés sur les valeurs que peuvent prendre au cours du temps les variables d'un programme SIGNAL. Cette analyse est originale, car elle se fait directement sur le programme source, contrairement aux autres analyses déjà définies pour les langages synchrones qui se font sur une représentation intermédiaire du programme. Puis, je présenterai succinctement le domaine abstrait "polyèdre", ainsi que l'opérateur d'élargissement définie sur ce domaine. Ensuite, je donnerai deux exemples (dont un ne terminant pas sans widening), pour lesquels je commenterai le résultat obtenu par cette analyse. Enfin, pour ceux qui voudraient reprendre mon programme, je décrirai en détails sa structure.

Chapitre 1

Le langage analysé SIGNAL

SIGNAL est un langage de programmation qui permet de synchroniser des événements, de définir une échelle de temps propre à un système, et d'accéder aux N dernières valeurs d'un événement. Ces événements sont, dans la sémantique de SIGNAL, des signaux; ils seront décrits dans la première section. Ensuite, les deux genres d'opérateurs (monochrones et polychrones) du langage seront présentés. Enfin, une grammaire représentative du langage analysé est donnée en dernière section.

1.1 Description des signaux

Un signal est constitué d'une suite de valeurs (de même type) et d'une horloge. L'horloge du signal défini les instants où le signal possède une valeur. Les autres instants, le signal possède la valeur $\perp$, on dit que le signal est absent. Ici, on ne considère que les signaux de type *event*, *boolean*, et *integer*. Les signaux de type *event* sont particuliers : l'intérêt de ces signaux n'est pas leurs valeurs, mais les instants auxquels ils se produisent. La seule valeur qu'ils peuvent prendre est *true*. Dans un processus SIGNAL, on peut accéder aux valeurs des signaux d'entrée et produire les valeurs des signaux de sorties. On peut, de plus, définir leur horloge. Grâce à l'horloge de chaque signal, on peut synchroniser les signaux : on peut spécifier qu'un signal $S1$ est présent seulement lorsque les signaux $S2$ et $S3$ sont présents tous les deux, ou lorsque l'un des deux est présent, ou lorsque $S2$ est présent, ...

Voyons maintenant les différents opérateurs qui permettent de définir l'horloge et/ou la valeur d'un signal.

1.2 Opérateurs du langage

Il existe deux sortes d'opérateurs : monochrones et polychrones. Les opérateurs monochrones imposent à ses opérandes d'être synchrones, c'est-à-dire d'avoir la même horloge : tous les opérandes doivent être présents aux mêmes instants. Par contre, les opérateurs polychrones donnent un résultat différent suivant que ses opérandes soient présents ou non.

1.2.1 Opérateurs monochrones

Les opérateurs monochrones comprennent tous les opérateurs sur les entiers et les booléens: $+, -, *, /, >, =, <=$, and, or, ... Il paraît évident qu'on ne peut calculer la valeur de X dans $X := Y + Z$ si Y ou Z est absent. Un autre opérateur est celui de l'horloge $\hat{}$ qui permet d'extraire l'horloge d'un signal. Enfin, l'opérateur \$ permet d'accéder aux valeurs précédentes d'un signal. Le signal retardé et le signal initial sont toujours synchrones.

1.2.2 Opérateurs polychrones

Les opérateurs polychrones comprennent le *when* unaire et binaire, et le *default*. Pour l'opérateur *when*, si son opérande droit est absent ou a la valeur *false*, alors le résultat est la valeur $\perp$. Par contre si son opérande droit a la valeur *true*, alors le résultat est, *true* dans le cas du *when* unaire, et l'opérande gauche dans le cas du *when* binaire. Pour l'opérateur *default*, si son opérande gauche est présent, alors le résultat est celui-ci, sinon son résultat est l'opérande droit.

1.3 Sous-langage traité

Le sous ensemble du langage SIGNAL que l'on peut analysé est défini par la grammaire suivante:

$$
\begin{array}{rcl}
prog & \rightarrow & \text{process } ident = \\
 & & (?signaux \quad !signaux) \\
 & & (\mid eqn \mid \ldots \mid eqn \mid) \\
 & & \text{where } mem \text{ end} \\
signaux & \rightarrow & type\ liste\_ident \\
type & \rightarrow & \text{integer} \mid \text{boolean} \\
eqn & \rightarrow & var := exp \mid var\ \hat{}= \ var \\
exp & \rightarrow & constant \mid var \mid delayvar \mid \ \hat{}\ var \mid var \text{ init } entier \mid \\
 & & delayvar \text{ init } valeur \mid \text{when } exp \mid exp \text{ when } exp \mid \\
 & & exp \text{ default } exp \mid f(exp \ldots exp) \\
delayvar & \rightarrow & var\ \$\ entier \\
var & \rightarrow & ident \\
mem & \rightarrow & mem; mem \mid type\ local\ (,local)^* \\
local & \rightarrow & var \mid var \text{ init } entier
\end{array}
$$

avec f qui représente les opérateurs monochrones tels que $+, >, =, \ldots$. Ici, le type *event* est représenté par le type *booléen*.

Cette partie a présentée très brièvement le langage synchrone SIGNAL. Le document [3] donne une introduction plus précise sur ce langage. Dans la suite de ce rapport, les signaux du programme seront appellés variables. Pour que l'analyse se passe bien, toutes les variables de retard doivent être initialisées.
Un des intérêts de ce stage a été l'utilisation de SmartTools pour la réalisation d'un analyseur statique. La partie suivante présente ce générateur d'environnement de programmation.

Chapitre 2

Le générateur d'environnement de programmation SmartTools

SmartTools est un générateur d'environnement de programmation développé en JAVA par Didier Parigot, Frank Chalaux, Carine Courbis, Pascal Degenne et Alexandre Fau du projet Oasis à Sophia Antipolis. Un générateur d'environnement de programmation est un logiciel qui permet de développer un environnement pour n'importe quel langage de programmation. Cet environnement peut comporter un éditeur, un compilateur et/ou un interpréteur, un débogueur, et toutes sortes d'applications. Centaur, qui est lui aussi un générateur d'environnement de programmation et dont SmartTools est le successeur, a été développé en LeLisp par l'ancienne équipe CROAP. SmartTools génère des environnements interactifs dédiés à un langage de programmation à partir de spécifications formelles. Il permet, entre autres, de développer des éditeurs structurés, des analyseurs, des compilateurs, des évaluateurs, des outils de mise au point, ...

Je vais maintenant présenter brièvement l'utilisation de SmartTools pour la réalisation d'un analyseur, les détails se trouve dans le manuel [4]. Il faut tout d'abord définir la syntaxe abstraite du langage à analyser, ce qui va permettre de définir l'architecture de l'arbre de syntaxe abstrait (ou AST). Cette première notion est présentée dans la première section. Ensuite, je décrirai le parser qui permet de faire l'analyse lexicale et syntaxique d'un programme source. Il sera décrit dans la seconde section. Puis, pour l'édition d'un programme source dans une fenêtre SmartTools, il est nécessaire d'avoir un pretty printer pour le langage analysé. Enfin, je parlerai des "Design Pattern Visitors" (ou modèles de conception de visiteurs) qui permettent de spécifier les actions de l'analyse sémantique.

2.1 La syntaxe abstraite

La syntaxe abstraite permet de définir l'architecture de l'AST (les noeuds) pour un langage donné. L'avantage de celle-ci par rapport à la syntaxe concrète est qu'elle permet de conserver seulement les informations dont on a besoin sous la forme souhaitée. Elle définit les types et les opérateurs d'un langage ainsi que

les dépendances entre ceux-ci. Cela permet de décrire la structure d'un langage sans s'emcombrer de la partie purement lexicale.

La syntaxe abstraite d'un langage doit être définie dans un fichier dont le nom est <*nom_du_langage*>.*ast*. Elle comporte deux notions importantes : phylum et opérateur. Un opérateur correspond au nom d'un noeud de l'arbre, et un phylum correspond au type du noeud.

La déclaration d'un opérateur consiste à donner le type de ses fils (s'ils existent) ainsi qu'un nom pour chacun d'eux. Lorsque l'on se trouve sur un noeud, ce nom permet d'obtenir l'arbre représentant le fils auquel le nom est associé (mais aussi d'affecter un arbre à ce fils). Voici un petit exemple de déclaration d'opérateur, l'opérateur arithmétique binaire "+" peut être déclaré par l'opérateur *plus* de la façon suivante :

plus(EXPRESSION left,EXPRESSION right)

La déclaration d'un phylum *ph* consiste à donner la liste des opérateurs qui sont du type *ph*. Cette liste peut également inclure d'autres phyla : cela signifie que le phylum *ph* comporte aussi tous les opérateurs des phyla se trouvant dans cette liste. Illustrons ceci par un exemple, le type (ou le phylum) *EXPRESSION* peut être défini ainsi :

EXPRESSION = plus(EXPRESSION left,EXPRESSION right), AUTRES_OPERATIONS;

ce qui signifie qu'une expression est soit une addition, soit une autre opération.

On peut aussi dire qu'un opérateur possède une liste de fils de même type. Une feuille de l'AST est soit un opérateur sans fils, soit un opérateur atomique : pour ce dernier cas, on doit préciser le type d'implémentation de sa valeur. Enfin, l'entête du fichier .ast est :

Formalism of <nom_du_langage> is
Root is <type_de_la_racine_de_l'AST>;

suivi de la liste des déclarations des phyla. En résumé, on peut voir un phylum comme un *ou* (une expression c'est une addition *ou* une autre opération) et un opérateur comme un *et* (une déclaration c'est un type *et* une liste de variables).

L'annexe A donne la syntaxe abstraite que j'ai choisie pour le langage SIGNAL. La section suivante donne quelques informations sur les générateurs de parsers utilisés dans SmartTools.

2.2 Le parser

On ne peut définir son parser qu'avec deux générateurs de parsers : JavaCC ou ANTLR, qui prennent en entrée une grammaire LL(k). La seule action de l'analyse sémantique du parser est le passage de la syntaxe concrète à la syntaxe abstraite. C'est à dire que les actions du parser construisent l'AST suivant la syntaxe abstraite définie dans le fichier .ast.

L'analyse sémantique du parser ne correspond donc pas aux actions que va effectuer l'analyseur. Voyons maintenant comment définir des pretty printers dans SmartTools.

2.3 Les pretty printers

Il faut savoir que SmartTools génère 3 pretty printers par défaut. Il n'est donc pas nécessaire d'en définir un soi-même. Toutefois, si l'on veut afficher les

programmes sources sous une autre forme que celles définies par défaut, il faut alors créer un fichier *<nom_du_langage>-std.ppml*.

C'est à l'issue de l'analyse lexicale et syntaxique qu'un programme source est affiché suivant la syntaxe du pretty printer. L'analyse sémantique d'un programme est effectuée par les visiteurs que je vais décrire dans la section suivante.

2.4 Les visiteurs

2.4.1 Généralités sur les *Design Pattern Visitors*

Un Visiteur est un modèle de conception appartenant au modèle "Design Pattern Visitors". Le modèle de conception définit une structure générale dans laquelle sont précisés : les classes et interfaces qui composent cette structure, leur rôle, leur coopération, ainsi que la hiérarchie de cette structure. Chaque modèle de conception se consacre à un problème ou à une solution particulière de conception orienté-objet. Un Visiteur représente une opération applicable aux éléments d'une structure d'objets liés entre eux par certaines relations (comme par exemple une structure arborescente). Il permet de définir une nouvelle opération, sans qu'il soit nécessaire de modifier la ou les classes définissant la structure et ses éléments. Ce modèle est avantageux lorsque la hiérarchie et les éléments de la structure ne changent pas ou peu. Si on prend l'exemple d'un compilateur (dont le programme source est sous forme d'AST), plutôt que de définir dans chaque noeud les méthodes *verifType* et *produireCode*, on créé un visiteur *verifTypeVisiteur* et un autre *produireCodeVisiteur*. Pour plus de détails sur ces notions, vous pouvez consulter le livre *Design patterns* [2]. Voyons maintenant l'implémentation des visiteurs dans SmartTools.

2.4.2 Les visiteurs de SmartTools

Dans SmartTools, les visiteurs offrent les fonctionnalités suivantes :

- ils autorisent à la fois les attributs synthétisés et hérités
- aucune contrainte n'est imposée sur l'ordre du parcours de l'arbre
- il est aussi possible de ne pas visiter certaines parties de l'arbre
- de plus, on peut laisser un nombre illimité d'objets dans chaque noeud de l'arbre

Dans chaque visiteur, il faut définir pour chaque noeud de l'AST que l'on veut parcourir, une méthode *visit*. Elle possède deux arguments : le premier est le noeud que l'on visite, et le second contient les attributs hérités. Le résultat de cette méthode *visit* contient les attributs synthétisés. Dans le corps de la méthode, on lance le parcours des fils que l'on souhaite visités (dans l'ordre que l'on veut) et l'on définit les actions associées à ce noeud.

Le passage des attributs synthétisés et hérités à travers l'arbre se fait sous forme d'un objet de la classe *Object*. Si on a plusieurs attributs à la fois, il suffit alors de construire un tableau contenant ces attributs pour obtenir un seul objet. De plus, il peut être intéressant de laisser des informations dans certains noeuds, informations que l'on souhaite retrouver plus tard lors d'une passe ultérieure sur l'arbre. Il suffit pour cela d'annoter l'AST. Les annotations sont en quelque sorte typées, car pour créer une annotation il faut lui donner un nom, ce qui permet de les regrouper.

A partir de la syntaxe abstraite d'un langage de programmation, SmartTools génère 6 visiteurs :

Abstract<nom_du_langage>Visitor : ce visiteur contient toutes les méthodes nécessaires pour parcourir l'arbre. Elle contient entre autres : une méthode *visit* par opérateur et des méthodes *<nom_du_phylum>visit* par phylum.

Default<nom_du_langage>Visitor : ce visiteur hérite du précédent, il est aussi la classe de base de tous les autres visiteurs qui ont été générés, et tout nouveau visiteur (créé par un utilisateur) doit étendre celui-ci. Le parcours qu'il définit est un parcours en profondeur de gauche à droite.

Dummy<nom_du_langage>Visitor : toutes les méthodes *visit* retourne *null*, il n'y a donc pas vraiment de parcours.

LookForMetaVariableVisitor : vérifie l'existence de métavariables dans l'AST.

SearchVisitor : cherche une valeur atomique dans l'AST.

ShowGTreeVisitor : permet d'afficher l'AST du programme source.

Pour créer son propre visiteur, il faut définir une classe qui hérite d'un de ces visiteurs.

Ainsi, avec le principe des visiteurs, il est très facile de faire plusieurs passes sur l'AST : il suffit de créer un visiteur par passe. Par exemple, si l'on veut générer un système de contraintes à partir d'un programme source, on peut faire une première passe qui récolte toutes les informations utiles sur les variables du programme, puis on donne ces informations au visiteur implémentant la seconde passe qui génère alors le système de contraintes.

Le chapitre suivant présente l'analyse polyèdrique réalisée avec SmartTools.

Chapitre 3

Description de l'analyse polyèdrique

Cette analyse statique permet de déduire des propriétés sur toutes les variables du programme qui s'expriment en fonction de leurs variables de retard. Ainsi, avec cette analyse, on peut vérifier qu'une variable est toujours comprise dans un intervalle donné. Le calcul de ces propriétés s'effectue en deux étapes. On extrait d'abord du programme source des contraintes linéaires qui décrivent à la fois les synchronisations entre les variables, et les valeurs qu'elles peuvent prendre. On obtient à l'issue de cette première étape, des ensembles de contraintes que l'on a calculé sous forme de polyèdres, et dont chacun décrit un comportement possible du programme SIGNAL. Puis, par itération de point fixe, on calcul pour chaque variable de retard l'intervalle de valeur dans lequel elle se situe, jusqu'à obtenir un intervalle stable.

Je ne présenterai pas la théorie sur la sémantique de l'analyse qui est détaillée dans l'article [1]. Dans la suite, je donne l'algorithme (extrait de [1]) que j'ai utilisé pour extraire du programme source les contraintes représentées sous forme de polyèdres. Puis je décrirai l'itération de point fixe sur les équations données au solveur REQS. Enfin, je donnerai quelques détails sur l'implémentation de cette analyse.

3.1 Extraction des contraintes

Pour chaque affectation ou synchronisation (équation), on définit les contraintes sous forme d'un ensemble d'ensembles de contraintes. Chaque ensemble de contraintes définit un comportement possible pour les variables de l'équation suivant qu'elles soient absentes ou présentes. Par exemple, l'affection de la variable y par la variable x peut avoir deux comportements possibles : soit x est présente et dans ce cas y est présente et $y = x$, soit x est absente et dans ce cas y est absente. Cet algorithme est donné par le tableau 3.1 page 14. Dans cet algorithme, on représente les booléens par des entiers : *1* pour *true* et *0* pour *false*. Par exemple, pour l'opérateur monochrone *not* dans $y :=$ not x, on obtient deux comportements possibles : soit y et x sont absents, soit ils sont présents et dans ce cas on a les 2 contraintes $y = 1 - x$ et $0 \leq x \leq 1$. On ne spécifie pas qu'un booléen ne peut prendre comme valeur que 0 ou 1, on le place seulement dans

l'intervalle [0..1]. Dans l'algorithme, je n'ai pas donné l'implémentation de l'extraction des contraintes pour chaque opérateur monochrone : le détail se trouve dans les commentaires de la classe *PolyhedronVisitor*. Dans le cas de la multiplication (sans constante), de la division (sans constante) et de la puissance, on ne peut rien dire à part que : soit toutes les variables sont présentes, soit elles sont toutes absentes, car on peut exprimer que des contraintes linéaires. Pour calculer tous les comportements possibles du programme source, on calcul toutes les combinaisons entre les ensembles de contraintes produits par l'extraction des contraintes de chaque affectation ou synchronisation. Ce nombre de comportements n'est pas aussi important qu'on pourrait le penser, car l'union d'un ensemble dans lequel la variable x est présente avec un ensemble dans lequel x est absente rend un ensemble vide (équivalent à un comportement impossible). Et ce cas se produit assez souvent.

TAB. 3.1 – *algorithme d'extraction des contraintes*

Synchro	$CE(y \hat{} = x)$	=	$\{\{y \neq \bot, x \neq \bot\}, \{y = \bot, x = \bot\}\}$
Const	$CE(y := c)$	=	$\{\{y = c, y \neq \bot\}, \{y = \bot\}\}$
Var	$CE(y := x)$	=	$\{\{y = x, y \neq \bot, x \neq \bot\}, \{y = \bot, x = \bot\}\}$
Delay	$CE(y := x \$ n)$	=	$\{\{y = x \$ n, y \neq \bot, x \neq \bot\}, \{y = \bot, x = \bot\}\}$
Clock	$CE(y := \hat{} x)$	=	$\{\{y = true, y \neq \bot, x \neq \bot\}, \{y = \bot, x = \bot\}\}$
MonoOp	$CE(y := f(x_1, \ldots, x_n))$	=	$\{\{y = f(x_1, \ldots, x_n), y \neq \bot, x_1 \neq \bot, \ldots, x_n \neq \bot\},$ $\{y = \bot, x_1 = \bot, x_n = \bot\}\}$
When	$CE(y :=$ when $x)$	=	$\{\{y = true, y \neq \bot, x = true, x \neq \bot\},$ $\{y = \bot, x = false, x \neq \bot\}, \{y = \bot, x = \bot\}\}$
When	$CE(y := z$ when $x)$	=	si z est une constante $\{\{y = z, y \neq \bot, x = true, x \neq \bot\}, \{y = \bot, x = false, x \neq \bot\},$ $\{y = \bot, x = \bot\}\}$ si z n'est pas une constante $\{\{y = z, y \neq \bot, x = true, x \neq \bot, z \neq \bot\},$ $\{y = \bot, x = true, x \neq \bot\ z = \bot\}, \{y = \bot, x = false, x \neq \bot\},$ $\{y = \bot, x = \bot\}\}$
Default	$CE(y := z$ default $x)$	=	si z est une constante $\{\{y = z, y \neq \bot\}, \{y = \bot\}\}$ si x est une constante $\{\{y = z, y \neq \bot, z \neq \bot\}, \{y = x, y \neq \bot, z = \bot\}, \{y = \bot, z = \bot\}\}$ si x et z ne sont pas des constantes $\{\{y = z, y \neq \bot, z \neq \bot\}, \{y = x, y \neq \bot, x \neq \bot, z = \bot\},$ $\{y = \bot, z = \bot, x = \bot\}\}$
Parallel	$CE(eq_1 \mid \ldots \mid eq_n)$	=	$\{C \mid C = \bigcup_{i=1}^{n} C_i \ , \ C_i \in CE(eq_i)\}$

Dans le cas de l'opérateur *when*, si x est une constante, alors cela provoque une erreur dans l'implémentation de l'algorithme. En effet, c'est équivalent à dire que y est toujours absent ($x = false$), ou équivalent à $y = z$ (ou à $y = true$) ($x = true$).

Voyons maintenant, comment à partir d'un ensemble de comportements possibles du programme, on calcul les intervalles de valeur dans lesquels se situent les variables de retard.

3.2 Itération de point fixe

Pour faire ce calcul, on modélise un programme SIGNAL par un ensemble de comportements $\{\mathcal{P}_i \mid 0 \leq i \leq n-1$,$n$ nombre de comportements possibles$\}$ et par l'état initial de la mémoire $init$. Un état de la mémoire associe à chaque variable de retard la valeur de cette variable. A chaque comportement $\mathcal{P}_i$ du programme, on associe un accumulateur Acc_i qui vaut initialement bottom (soit le polyèdre vide). Cet accumulateur représente un état de la mémoire à un instant donné. On le calcule avec l'équation récursive suivante :

$$Acc_i = \bigcup_{j=0}^{n-1} (Acc_j \downarrow \mathcal{P}_i) \bigcup (init \downarrow \mathcal{P}_i) \tag{3.1}$$

où $init$ donne la valeur initiale de chaque variable de retard, $\downarrow$ projette un état de la mémoire sur un ensemble de contraintes, et l'union correspond à l'union sur les polyèdres. Ici, $init$ et tous les Acc_i sont des polyèdres dont la dimension de l'espace où ils résident est d, le nombre de variables de retard du programme. La dimension de l'espace dans lequel résident les polyèdres $\mathcal{P}_i$ est de dimension v, le nombre de variables du programme ($d < v$).

L'opération de projection comporte trois étapes :

- on modifie le polyèdre Acc_j pour qu'il réside dans un espace de même dimension que celui d'un polyèdre $\mathcal{P}_i$. On obtient alors le polyèdre Acc'_j.
- on calcule l'intersection entre Acc'_j et $\mathcal{P}_i$. Ceci permet de tenir compte à la fois des contraintes sur les valeurs des variables de retard et à la fois des contraintes entre variables et variables de retard (union des contraintes).
- on transite ce nouveau polyèdre vers un nouvel état de la mémoire dans lequel, les variables de retard présentes, prennent la valeur de leur variable initiale, et les variables de retard absentes gardent leur valeur.

L'itération de point fixe se fait grâce au solveur d'équations récursives REQS. Ce solveur a été réalisé par Florimond Ployette de l'équipe LANDE. Plusieurs stratégies y sont utilisées pour résoudre les équations. Dans notre cas, on utilise la stratégie naïve : elle recalcule toutes les variables tant qu'au moins une d'entre elles a été modifiée. On a fait ce choix, car chaque variable dépend de toutes les autres variables. Pour utiliser REQS, il suffit de définir un domaine et les opérations nécessaires sur ce domaine, comme l'intersection, l'union, l'infériorité (ordre partiel), l'égalité, l'élément *bottom*, et la projection définie plus haut. Il faut aussi donner les équations à résoudre, qui ici correspondent aux équations 3.1. Mais en pratique, d'autres fichiers sont nécessaires pour ces calculs. Nous allons voir leur description ainsi que quelques détails sur l'implémentation de l'analyse dans la section suivante.

3.3 Implémentation de l'analyse

J'ai effectué l'analyse en 3 passes sur l'AST représentant le programme source. La première permet d'évaluer les expressions constantes. En effet, la génération d'équations intermédiaires (affectations SIGNAL) pour évaluer des expressions constantes pose problème en produisant beaucoup trop de comportements possibles pour le programme. On ne peut éviter la génération d'affectations intermédiaires, car le traitement des expressions complexes (com-

portant plus d'un opérateur) rend ces affectations intermédiaires nécessaires. Voyons ce problème sur l'exemple : $y := 3 + 4$ when z. La génération d'affectations intermédiaires donnent : $var := 3 + 4 \mid y := var$ when z. Normalement la constante $3 + 4$ n'est présente que lorsque y est présente, donc var devrait être présente seulement si y est présente. Or, d'après le traitement de l'affectation intermédiaire, on peut dire que var est présente même quand y est absente, ce qui va produire de nombreux comportements possibles (au moins deux fois plus) pour le programme qui ne sont pas vrais pour le programme source initial.

La seconde passe permet à la fois de générer les affectations intermédiaires, et de récolter toutes les informations nécessaires sur les variables du programme source et les variables intermédiaires. Je génère autant d'affectations et de variables intermédiaires qu'il est nécessaire pour obtenir des affectations comportant au plus un opérateur en partie droite. Les informations récoltées sur les variables sont : le nom, le type, la valeur initiale si elle existe, le rang dans le polyèdre $\mathcal{P}_i$ et le fait qu'elle soit ou non une variable de retard. Si c'est le cas, on conserve les informations supplémentaires suivantes : l'indice de retard, la variable initiale et le rang dans le polyèdre Acc_i.

Avec les informations récupérées lors de la seconde passe, on peut retrouver l'état initial de la mémoire (ie *init*), et on peut extraire les contraintes en suivant l'algorithme définit dans le tableau 3.1 page 14. Ceci constitue la troisième passe pendant laquelle on produit un ensemble d'ensembles de contraintes pour chaque équation (affectation ou synchronisation). A la fin, on récupère tous ces ensembles et on produit toutes les combinaisons possibles (traitement de l'opérateur **Parallel**) avec un calcul en profondeur. Ce calcul produit les polyèdres $\mathcal{P}_i$. Pour chacun de ces polyèdres, on produit une transition qui permet de projeter une variable de retard sur sa variable initiale si elle est présente, sinon elle est projetée sur elle-même. Je génère aussi une matrice de conversion qui permet d'implémenter la première étape de l'opération de projection spécifiée en 3.2. Enfin, je produis le fichier contenant les équations récursives pour REQS.

Tous les calculs sur les ensembles de contraintes sont faits grâce à une librairie de polyèdres. La partie suivante présente ces polyèdres, leurs représentations et leurs utilisations, ainsi qu'un opérateur d'élargissement définit sur ce domaine.

Chapitre 4

Polyèdre

Dans cette analyse un ensemble de contraintes linéaires est représenté par un polyèdre. On utilise ici, une librairie C de polyèdres, celle développée par l'équipe API de l'IRISA (cf document [5]). Il existe deux représentations duales pour les polyèdres. La première section va les présenter. Ensuite, je décrirai les opérations sur les polyèdres utilisées pour cette analyse. Enfin, je présenterai l'opérateur d'élargissement qui a été implémenté.

4.1 Représentations duales des polyèdres

On peut voir un polyèdre soit comme un ensemble de contraintes, soit comme un ensemble de lignes, de rayons et de sommets. Chaque contrainte (égalité ou inégalité) correspond à un point, à une droite ou à un plan à n dimensions. Une contrainte se représente tout simplement par un tableau d'entiers $(S,a_1,\ldots,a_n,K)$ correspondant aux coefficients (a_i) associés à chaque variable (X_i) décrivant le polyèdre. Une contrainte a donc la forme suivante :

$$S = 0 \ : \ a_1X_1 + \ldots + a_nX_n + K = 0$$
$$S = 0 \ : \ a_1X_1 + \ldots + a_nX_n + K \leq 0$$

Pour définir les contraintes, on a donc besoin d'affecter à une variable un rang dans le polyèdre. Dans la seconde représentation des polyèdres, une ligne correspond à une direction infinie (dans les deux sens) et un rayon correspond à une direction infinie (dans un seul sens). Voyons maintenant les opérations que l'on peut effectuer sur les polyèdres.

4.2 Opérations définies sur les polyèdres

La librairie C des polyèdres permet l'utilisation des opérations suivantes :

intersection : correspond à l'union sur les ensembles de contraintes.

union convexe : correspond à l'intersection sur les ensembles de contraintes. La différence de cette union avec l'union simple est que son résultat est un polyèdre convexe et non pas une liste de polyèdres. L'utilisation de cette union dans les calculs constituent bien sûr une approximation, puisque l'on ajoute dans le polyèdre résultat des points qui n'appartiennent normalement pas à ce résultat.

image : permet d'obtenir un polyèdre à m dimensions qui donne l'image d'un polyèdre à n dimensions (m<n). On peut ainsi projeter des variables sur d'autres variables.

préimage : permet d'obtenir un polyèdre à m dimensions qui donne l'image d'un polyèdre à n dimensions (m>n). On peut ainsi agrandir un polyèdre sur de nouvelles variables.

On a aussi des opérations qui permettent d'obtenir le polyèdre vide ou le polyèdre universel, qui permettent de construire un polyèdre à partir d'un ensemble de contraintes, ou à partir d'un ensemble de lignes, rayons et sommets. Je vais dans la section suivante présenter en détails l'opérateur d'élargissement réalisé.

4.3 Opérateur d'élargissement

La première partie donne la définition de l'opérateur implémenté. Puis la seconde présente son application sur deux exemples.

4.3.1 Définition de l'opérateur

Des opérateurs d'élargissement sur les polyèdres ont déjà été définis. Leur technique est de décroître strictement le nombre de contraintes. Par contre, celui définit dans l'article [1] travaille sur la seconde représentation des polyèdres : lignes, rayons et sommets. Pour déclencher cet opérateur, on prend en compte 3 paramètres : la dimension du polyèdre (dimension du plus petit sous-espace affine dans lequel il se trouve), le nombre de lignes et le nombre de sommets. Comme la dimension d'un polyèdre est bornée par la dimension de l'espace dans lequel il réside, si cette dimension augmente strictement d'itération en itération, cette augmentation finira par s'arrêter. Comme on peut avoir au plus n lignes linéairement indépendantes dans un espace à n dimensions, si ce nombre de lignes augmente strictement, cette augmentation finira par s'arrêter. Comme le nombre de sommets d'un polyèdre est borné par 1, si ce nombre diminue strictement d'itération en itération, cette diminution finira par s'arrêter. Dans une chaîne ascendante de polyèdres ($P_0 \subseteq P_1 \ldots \subseteq P_n \subseteq \ldots$), on déclenche l'opérateur d'élargissement seulement quand :

$$d_{P_i} \geq d_{P_{i+1}} \quad \text{et} \quad l_{P_i} \geq l_{P_{i+1}} \quad \text{et} \quad v_{P_i} \leq v_{P_{i+1}} \tag{4.1}$$

avec d la dimension, l le nombre de lignes et v le nombre de sommets. La stratégie utilisée par cet opérateur est de décroître le nombre de sommets et d'augmenter le nombre de rayons et de lignes. La technique consiste à :

- calculer l'ensemble des sommets V_i (V_{i+1}) qui contient les sommets de P_i (respectivement P_{i+1}) moins les sommets en commun entre P_i et P_{i+1} (ie les sommets qui n'ont pas bougé). Si V_{i+1} n'est pas vide :
  - on choisit un sommet v dans V_{i+1} (sommet qui a évolué) que l'on va repousser vers l'infinie. Dans l'implémentation ce sommet est choisi au hasard.
  - on retrouve le sommet de V_i lui "correspondant". On choisi le sommet v' de V_i ayant la plus petite distance avec v.

– on calcule le rayon correspondant à la direction qui va de v' à v, et on l'ajoute au nouveau polyèdre P' résultat de l'élargissement.

Si V_{i+1} est vide et que P_{i+1} possède un rayon :

– on choisit au hasard un rayon de P_{i+1} que l'on transforme en ligne dans P'.

Dans les autres cas, on rend une erreur.

Si le calcul s'est bien passé, on itère ce calcul sur P_i et P' tant que la condition 4.1 est vraie.

Voyons maintenant l'application de cet opérateur d'élargissement sur deux exemples extraits de l'article [1].

4.3.2 Application sur des exemples

Le premier exemple (figure 4.1) est un triangle dont le sommet ne cesse d'aller vers le haut. Le widening va alors repousser ce sommet vers l'infinie, et on va obtenir une demi-bande infinie.

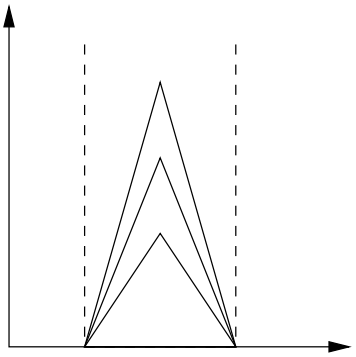

FIG. 4.1 – *Limit out of the scope of widening strategy*

Le second exemple (figure 4.2) est un triangle dont deux sommets s'éloignent de plus en plus l'un de l'autre. Après une première application du widening, on obtient une demi-bande infinie. Puis, comme le montre la figure 4.2, après une seconde application du widening, on obtient une bande infinie.

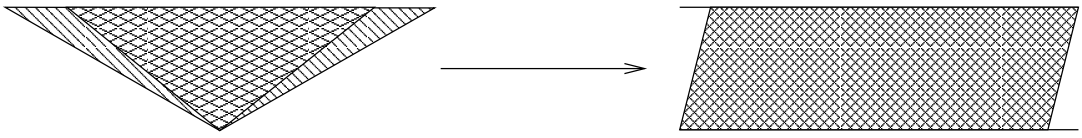

FIG. 4.2 – *An infinite band rather than an half-space*

Nous allons maintenant voir les résultats que produit cette analyse sur deux exemples, dont un nécessite l'opérateur d'élargissement pour converger.

Chapitre 5

Exemples

Dans cette analyse, pour savoir si une propriété est vérifiée, il est préférable de l'exprimer sous forme d'une équation SIGNAL, et ensuite de vérifier dans le résultat de l'analyse si elle est toujours vraie ou fausse pour tout les comportements possibles du programme. Le premier exemple porte sur une analyse qui converge, alors que le second porte sur une analyse qui converge seulement si l'on utilise un opérateur d'élargissement.

5.1 Bathtub

Dans ce premier exemple (extrait de [1]), on considère une baignoire dont on doit contrôler le niveau d'eau. On veut s'assurer que la baignoire ne soit jamais vide (level > 0) et qu'elle ne déborde pas (level < 9). Dans le cas contraire on déclenche une alarme. Pour remplir la baignoire, on possède un robinet. De plus, pour vider la baignoire, on possède une pompe. Au départ, on initialise le niveau de l'eau à 1, et le robinet et la pompe ne sont pas en marche. Ensuite, suivant le niveau de l'eau dans la baignoire, on met en marche le robinet et la pompe avec un débit plus ou moins important. Voici le programme SIGNAL correspondant à cet exemple :

```
process bathtub =
( ?
  ! boolean alarm )
(| level := zlevel + faucet - pump
 | faucet := zfaucet + ((1 when zlevel <= 4)
                        default (-1 when zfaucet > 0)
                        default 0 )
 | pump := zpump + ((1 when zlevel >= 7)
                    default (-1 when zpump >0 )
                    default 0 )
 | alarm := (0 >= level) or (level >= 9)                    (1)
 | zlevel := level $
 | zfaucet := faucet $
 | zpump := pump $
 | zalarm := alarm $
 |)
```

```
where
integer faucet, pump;
integer level, zlevel init 1;
integer zfaucet init 0, zpump init 0;
boolean zalarm init false
end
```

Dans ce programme, l'équation (1) exprime la propriété que l'on veut vérifier : la baignoire ne doit pas déborder et ne doit pas se vider complètement. On va donc vérifier pour tous les comportements possibles du programme que $zalarm$ est toujours faux ($zalarm = 0$). Voici le résultat de l'analyse de cet exemple :

```
Acc19: bottom
init:
Constraints :   ~   + 1 zlevel -1 = 0  ~   + 1 zpump = 0  ~
 + 1 zfaucet = 0  ~   + 1 zalarm = 0  ~   + 1 >= 0
Acc18: bottom
Acc17: bottom
Acc16: bottom
Acc15:
Constraints :   ~   + 1 zpump -1 = 0  ~   + 1 zalarm = 0  ~
 + 1 zlevel -1 zfaucet -6 >= 0  ~  -2 zlevel  + 1 zfaucet  + 14 >= 0  ~
 -1 zlevel  + 2 zfaucet  + 6 >= 0
Acc14: bottom
Acc13:
Constraints :   ~   + 1 zalarm = 0  ~
 + 1 zlevel  + 1 zpump -2 zfaucet -7 >= 0  ~   + 1 zfaucet >= 0  ~
 -2 zlevel -3 zpump  + 3 zfaucet  + 17 >= 0  ~   + 1 zpump -2 >= 0
Acc12: bottom
Acc11: bottom
Acc10:
Constraints :   ~   + 1 zlevel -2 = 0  ~   + 1 zpump = 0  ~
 + 1 zfaucet -1 = 0  ~   + 1 zalarm = 0  ~   + 1 >= 0
Acc31:
Constraints :   ~   + 1 zpump = 0  ~   + 1 zfaucet = 0  ~
 + 1 zalarm = 0  ~  -1 zlevel  + 6 >= 0  ~   + 1 zlevel -5 >= 0
Acc30: bottom
Acc9: bottom
Acc8: bottom
Acc7:
Constraints :   ~   + 1 zfaucet -1 = 0  ~   + 1 zalarm = 0  ~
 -1 zlevel -1 zpump  + 5 >= 0  ~   + 5 zlevel  + 4 zpump -23 >= 0  ~
  + 1 zlevel  + 2 zpump -5 >= 0
Acc6: bottom
Acc5:
Constraints :   ~   + 1 zlevel -3 zfaucet  + 2 = 0  ~   + 1 zpump = 0  ~
  + 1 zalarm = 0  ~  -1 zfaucet  + 3 >= 0  ~   + 1 zfaucet -2 >= 0
Acc4: bottom
Acc3: bottom
Acc2:
Constraints :   ~   + 1 zalarm = 0  ~
```

```
 -1 zlevel -2 zpump  + 1 zfaucet  + 4 >= 0  ~   + 1 zpump >= 0  ~
  + 5 zlevel  + 9 zpump -6 zfaucet -17 >= 0  ~   + 1 zfaucet -2 >= 0
Acc1: bottom
Acc0: bottom
Acc29: bottom
Acc28:
Constraints :   ~   + 1 zfaucet = 0  ~   + 1 zalarm = 0  ~
 + 1 zlevel  + 1 zpump -5 >= 0  ~  -2 zlevel -5 zpump  + 14 >= 0  ~
 -2 zlevel -3 zpump  + 12 >= 0  ~   + 1 zlevel  + 2 zpump -6 >= 0
Acc27: bottom
Acc26: bottom
Acc25: bottom
Acc24: bottom
Acc23: bottom
Acc22: bottom
Acc21: bottom
Acc20:
Constraints :   ~   + 1 zpump = 0  ~   + 1 zalarm = 0  ~
 + 1 zlevel -4 zfaucet -1 >= 0  ~   + 1 zlevel -1 zfaucet -5 >= 0  ~
 + 1 zfaucet >= 0  ~  -1 zlevel  + 1 zfaucet  + 6 >= 0
```

init représente l'état initial de la mémoire, et les variables Acc_i représentent les différents comportements du programme. Certains comportements ne sont pas possibles : on le voit au fait que la résolution de l'équation récursive aboutit à bottom. Sur ce premier exemple, l'analyse a produit 32 comportements différents. Dans le résultat de chacune des équations récursives, on voit que *zalarm* est toujours faux, donc on est sûr que la propriété est vérifiée, l'alarme ne se déclenchera jamais.

Par contre, si dans le même programme, on modifie l'équation SIGNAL (1) de la façon suivante :

```
alarm := (0 >= level) or (level >= 7)
```

le résultat de l'analyse montre que l'alarme peut se déclencher. Voici le résultat :

```
Acc19: bottom
init:
Constraints :   ~   + 1 zlevel -1 = 0  ~   + 1 zpump = 0  ~
 + 1 zfaucet = 0  ~   + 1 zalarm = 0  ~   + 1 >= 0
Acc18: bottom
Acc17: bottom
Acc16: bottom
Acc15: bottom
Acc14:
Constraints :   ~   + 1 zlevel -8 = 0  ~   + 1 zpump -1 = 0  ~
 + 1 zfaucet -2 = 0  ~   + 1 zalarm -1 = 0  ~   + 1 >= 0
Acc13:
Constraints :   ~   + 1 zlevel -4 = 0  ~   + 1 zpump -3 = 0  ~
 + 1 zfaucet = 0  ~   + 1 zalarm = 0  ~   + 1 >= 0
Acc12:
Constraints :   ~   + 1 zlevel -7 = 0  ~   + 1 zpump -2 = 0  ~
 + 1 zfaucet -1 = 0  ~   + 1 zalarm -1 = 0  ~   + 1 >= 0
Acc11: bottom
```

```
Acc10:
Constraints :   ~   + 1 zlevel -2 = 0  ~   + 1 zpump = 0  ~
 + 1 zfaucet -1 = 0  ~   + 1 zalarm = 0  ~   + 1 >= 0
Acc31: bottom
Acc30: bottom
Acc9: bottom
Acc8: bottom
Acc7:
Constraints :   ~   + 1 zlevel -3 = 0  ~   + 1 zpump -2 = 0  ~
 + 1 zfaucet -1 = 0  ~   + 1 zalarm = 0  ~   + 1 >= 0
Acc6: bottom
Acc5:
Constraints :   ~   + 1 zlevel -4 = 0  ~   + 1 zpump = 0  ~
 + 1 zfaucet -2 = 0  ~   + 1 zalarm = 0  ~   + 1 >= 0
Acc4:
Constraints :   ~   + 1 zlevel -7 = 0  ~   + 1 zpump = 0  ~
 + 1 zfaucet -3 = 0  ~   + 1 zalarm -1 = 0  ~   + 1 >= 0
Acc3: bottom
Acc2:
Constraints :   ~   + 1 zlevel -4 = 0  ~   + 1 zpump -1 = 0  ~
 + 1 zfaucet -2 = 0  ~   + 1 zalarm = 0  ~   + 1 >= 0
Acc1:
Constraints :   ~   + 1 zlevel -7 = 0  ~   + 1 zpump = 0  ~
  + 1 zfaucet -3 = 0  ~   + 1 zalarm -1 = 0  ~   + 1 >= 0
Acc0: bottom
Acc29: bottom
Acc28: bottom
Acc27: bottom
Acc26: bottom
Acc25: bottom
Acc24: bottom
Acc23: bottom
Acc22: bottom
Acc21: bottom
Acc20: bottom
```

Pour les comportements *Acc13*, *Acc10*, *Acc7*, *Acc5* et *Acc2* l'alarme ne se déclenche pas, alors que pour les comportements *Acc14*, *Acc12*, *Acc4* et *Acc1* elle se déclenche.

Voyons maintenant un exemple qui utilise l'opérateur d'élargissement.

5.2 Train

Dans ce second exemple, on considère un train qui parcours un nombre infini de mètres et qui doit traverser un passage à niveau. Au passage du train, le passage à niveau doit bien sûr être fermé. Un détecteur se situe avant le passage à niveau. Lorsque ce détecteur détecte un train, soit il fait fermé la barrière, soit il fait arrêter le train par une signalisation. Voici le programme SIGNAL correspondant à cet exemple :

```
% modelisation de l'exemple du train-gate controler revisisted (Halbwachs)
```

```
process train =
( ?
  ! )
% exterieur
(| second := true
 | meter := true
 | speed_bound := true

% 1 <= speed <= 3 => nb meter - nb second >= 0 and 3 nb second - nb meter >=0
 | speed_bound := meter_counter - second_counter>=0
       and 3* second_counter - meter_counter>=0
 | clk := meter default second

% Compute the average speed
 | meter_counter :=  zmeter_counter when brake default
       zmeter_counter + 1 when meter
        default zmeter_counter
 | zmeter_counter := meter_counter $ 1 init 0
 | meter_counter^= clk
 | second_counter := zsecond_counter +1 when second default zsecond_counter
 | zsecond_counter := second_counter $ 1 init 0
 | second_counter^= clk

% reperes
 | signal := 10
 | gate := 11

% Check is_closed
 | is_closed := true when not failure
        default zis_closed
 | zis_closed := is_closed $ 1 init false
 | is_closed ^= clk
 | t2 := true when (second_counter=2)
 | failure^= t2

 | red_signal := not is_closed and (second_counter>=3) default zred_signal
 | zred_signal := red_signal $ 1 init false
 | red_signal ^= clk

 | brake := zmeter_counter=signal when red_signal

% consequences
 | pte := is_closed when meter_counter=gate                (2)
 | zpte := pte $ 1 init true
 |)
where
 integer meter_counter, zmeter_counter, second_counter, zsecond_counter;
 boolean second,meter,borne;
 integer signal,gate;
 boolean is_closed,zis_closed;
```

```
 boolean red_signal,zred_signal,brake,speed_bound;
 boolean failure,t2,pte,clk,zpte
end
```

Dans ce programme, l'équation (2) exprime la propriété que l'on souhaite vérifier : la barrière doit être fermée lorsque le train traverse le passage à niveau. On va le vérifier grâce au résultat de l'analyse :

```
Acc19: bottom
init:
Constraints :   ~   + 1 zmeter_counter = 0  ~   + 1 zis_closed = 0  ~
 + 1 zpte -1 = 0  ~   + 1 zsecond_counter = 0  ~   + 1 zred_signal = 0  ~
 + 1 >= 0
Acc18:
Constraints :   ~   + 1 zis_closed -1 = 0  ~   + 1 zpte -1 = 0  ~
 + 1 zred_signal = 0  ~   + 1 zmeter_counter -1 zsecond_counter >= 0  ~
 -1 zmeter_counter  + 3 zsecond_counter -3 >= 0  ~
 + 1 zsecond_counter -3 >= 0  ~  -1 zmeter_counter  + 9 >= 0
Acc17:
Constraints :   ~   + 1 zis_closed -1 = 0  ~   + 1 zpte -1 = 0  ~
 + 1 zred_signal = 0  ~  -1 zmeter_counter  + 3 zsecond_counter -3 >= 0  ~
 + 1 zmeter_counter -1 zsecond_counter >= 0  ~
 + 1 zmeter_counter -12 >= 0  ~   + 1 >= 0
Acc16:
Constraints :   ~   + 1 zmeter_counter -11 = 0  ~   + 1 zis_closed -1 = 0  ~
  + 1 zpte -1 = 0  ~   + 1 zred_signal = 0  ~
 -1 zsecond_counter  + 11 >= 0  ~   + 3 zsecond_counter -14 >= 0
Acc15:
Constraints :   ~   + 1 zmeter_counter -10 = 0  ~   + 1 zis_closed -1 = 0  ~
  + 1 zpte -1 = 0  ~   + 1 zred_signal = 0  ~
 -1 zsecond_counter  + 10 >= 0  ~   + 3 zsecond_counter -13 >= 0
Acc14:
Constraints :   ~   + 1 zis_closed = 0  ~   + 1 zpte -1 = 0  ~
  + 1 zsecond_counter -2 = 0  ~   + 1 zred_signal = 0  ~
 + 1 zmeter_counter -2 >= 0  ~  -1 zmeter_counter  + 3 >= 0
Acc13:
Constraints :   ~   + 1 zis_closed -1 = 0  ~   + 1 zpte -1 = 0  ~
 + 1 zsecond_counter -2 = 0  ~   + 1 zred_signal = 0  ~
  + 1 zmeter_counter -2 >= 0  ~  -1 zmeter_counter  + 3 >= 0
Acc12:
Constraints :   ~   + 1 zis_closed = 0  ~   + 1 zpte -1 = 0  ~
 + 1 zred_signal -1 = 0  ~   + 1 zmeter_counter -1 zsecond_counter >= 0  ~
 -1 zmeter_counter  + 9 >= 0  ~  -1 zmeter_counter  + 3 zsecond_counter -3 >= 0  ~
  + 1 zsecond_counter -3 >= 0
Acc11: bottom
Acc10: bottom
Acc9:
Constraints :   ~   + 1 zmeter_counter -10 = 0  ~   + 1 zis_closed = 0  ~
  + 1 zpte -1 = 0  ~   + 1 zred_signal -1 = 0  ~
 -1 zsecond_counter  + 10 >= 0  ~   + 3 zsecond_counter -13 >= 0
Acc8:
```

```
Constraints :   ~   + 1 zmeter_counter -1 = 0  ~   + 1 zis_closed = 0  ~
  + 1 zpte -1 = 0  ~   + 1 zsecond_counter -1 = 0  ~
 + 1 zred_signal = 0  ~   + 1 >= 0
Acc7:
Constraints :   ~   + 1 zis_closed -1 = 0  ~   + 1 zpte -1 = 0  ~
  + 1 zred_signal = 0  ~   + 1 zsecond_counter -3 >= 0  ~
  + 1 zmeter_counter -1 zsecond_counter >= 0  ~  -1 zmeter_counter  + 10 >= 0  ~
 -1 zmeter_counter  + 3 zsecond_counter -2 >= 0
Acc6:
Constraints :   ~   + 1 zis_closed -1 = 0  ~   + 1 zpte -1 = 0  ~
  + 1 zred_signal = 0  ~   + 1 zmeter_counter -1 zsecond_counter >= 0  ~
  + 1 zmeter_counter -12 >= 0  ~
 -1 zmeter_counter  + 3 zsecond_counter -2 >= 0  ~   + 1 >= 0
Acc5:
Constraints :   ~   + 1 zmeter_counter -11 = 0  ~   + 1 zis_closed -1 = 0  ~
  + 1 zpte -1 = 0  ~   + 1 zred_signal = 0  ~
 -1 zsecond_counter  + 11 >= 0  ~   + 3 zsecond_counter -13 >= 0
Acc4:
Constraints :   ~   + 1 zis_closed = 0  ~   + 1 zpte -1 = 0  ~
  + 1 zsecond_counter -2 = 0  ~   + 1 zred_signal = 0  ~
  + 1 zmeter_counter -2 >= 0  ~  -1 zmeter_counter  + 4 >= 0
Acc3:
Constraints :   ~   + 1 zis_closed -1 = 0  ~   + 1 zpte -1 = 0  ~
  + 1 zsecond_counter -2 = 0  ~   + 1 zred_signal = 0  ~
  + 1 zmeter_counter -2 >= 0  ~  -1 zmeter_counter  + 4 >= 0
Acc2:
Constraints :   ~   + 1 zis_closed = 0  ~   + 1 zpte -1 = 0  ~
  + 1 zred_signal -1 = 0  ~  -1 zmeter_counter  + 10 >= 0  ~
  + 1 zmeter_counter -1 zsecond_counter >= 0  ~   + 1 zsecond_counter -3 >= 0  ~
 -1 zmeter_counter  + 3 zsecond_counter -2 >= 0
Acc1: bottom
Acc0:
Constraints :   ~   + 1 zmeter_counter -10 = 0  ~   + 1 zis_closed = 0  ~
 + 1 zpte -1 = 0  ~   + 1 zred_signal -1 = 0  ~
 -1 zsecond_counter  + 10 >= 0  ~   + 3 zsecond_counter -13 >= 0
Acc28:
Constraints :   ~   + 1 zis_closed = 0  ~   + 1 zpte -1 = 0  ~
  + 1 zsecond_counter -1 = 0  ~   + 1 zred_signal = 0  ~
  + 1 zmeter_counter -2 >= 0  ~  -1 zmeter_counter  + 3 >= 0
Acc27:
Constraints :   ~   + 1 zis_closed -1 = 0  ~   + 1 zpte -1 = 0  ~
  + 1 zred_signal = 0  ~  -1 zmeter_counter  + 10 >= 0  ~
  + 1 zmeter_counter -1 zsecond_counter -1 >= 0  ~
  + 1 zsecond_counter -3 >= 0  ~  -1 zmeter_counter  + 3 zsecond_counter >= 0
Acc26:
Constraints :   ~   + 1 zis_closed -1 = 0  ~   + 1 zpte -1 = 0  ~
  + 1 zred_signal = 0  ~   + 1 zmeter_counter -1 zsecond_counter -1 >= 0  ~
  + 1 zmeter_counter -12 >= 0  ~
 -1 zmeter_counter  + 3 zsecond_counter >= 0  ~   + 1 >= 0
Acc25:
```

```
Constraints :   ~   + 1 zmeter_counter -11 = 0  ~   + 1 zis_closed -1 = 0  ~
  + 1 zpte -1 = 0  ~   + 1 zred_signal = 0  ~
 -1 zsecond_counter  + 10 >= 0  ~   + 3 zsecond_counter -11 >= 0
Acc24:
Constraints :   ~   + 1 zpte -1 = 0  ~   + 1 zsecond_counter -2 = 0  ~
  + 1 zred_signal = 0  ~  -1 zis_closed  + 1 >= 0  ~
  + 1 zmeter_counter -3 >= 0  ~   + 1 zis_closed >= 0  ~
 -1 zmeter_counter  + 6 >= 0
Acc23:
Constraints :   ~   + 1 zis_closed -1 = 0  ~   + 1 zpte -1 = 0  ~
  + 1 zsecond_counter -2 = 0  ~   + 1 zred_signal = 0  ~
  + 1 zmeter_counter -3 >= 0  ~  -1 zmeter_counter  + 6 >= 0
Acc22:
Constraints :   ~   + 1 zis_closed = 0  ~   + 1 zpte -1 = 0  ~
  + 1 zred_signal -1 = 0  ~  -1 zmeter_counter  + 10 >= 0  ~
 + 1 zmeter_counter -1 zsecond_counter -1 >= 0  ~
  + 1 zsecond_counter -3 >= 0  ~  -1 zmeter_counter  + 3 zsecond_counter >= 0
Acc21: bottom
Acc20:
Constraints :   ~   + 1 zmeter_counter -10 = 0  ~   + 1 zis_closed = 0  ~
  + 1 zpte -1 = 0  ~   + 1 zred_signal -1 = 0  ~
 + 3 zsecond_counter -10 >= 0  ~  -1 zsecond_counter  + 10 >= 0
```

Sur cet exemple, l'analyse a produit 29 comportements différents. Dans le résultat de chacune des équations récursives, on voit que *zpte* est toujours vrai, donc la propriété est vérifiée, le train ne traversera jamais le passage à niveau sans que la barrière ne soit fermée. Dans cet exemple, l'utilisation de l'opérateur d'élargissement est nécessaire, car le train parcours un nombre infini de mètres. Cet opérateur a modifié les comportements *Acc*17, *Acc*6 et *Acc*26. Voyons les contraintes qu'ils possédaient avant que l'opérateur n'ait été appliqué :

```
Acc17 :

Constraints :   ~   + 1 zis_closed -1 = 0  ~   + 1 zpte -1 = 0  ~
  + 1 zred_signal = 0  ~   + 1 zmeter_counter -12 >= 0  ~
  + 1 zmeter_counter -1 zsecond_counter >= 0  ~
 -1 zmeter_counter  + 3 zsecond_counter -3 >= 0  ~
 -1 zmeter_counter  + 1 zsecond_counter  + 16 >= 0  ~
 -1 zmeter_counter  + 31 >= 0  ~
 -1 zmeter_counter -1 zsecond_counter  + 60 >= 0
Acc6 :

Constraints :   ~   + 1 zis_closed -1 = 0  ~   + 1 zpte -1 = 0  ~
  + 1 zred_signal = 0  ~   + 1 zmeter_counter -12 >= 0  ~
 + 1 zmeter_counter -1 zsecond_counter >= 0  ~
 -1 zmeter_counter -1 zsecond_counter  + 62 >= 0  ~
 -1 zmeter_counter  + 3 zsecond_counter -2 >= 0  ~
 -1 zmeter_counter  + 1 zsecond_counter  + 17 >= 0  ~
 -1 zmeter_counter  + 32 >= 0
Acc26 :
```

```
Constraints :   ~   + 1 zis_closed -1 = 0  ~   + 1 zpte -1 = 0  ~
  + 1 zred_signal = 0  ~  -1 zmeter_counter  + 1 zsecond_counter  + 18 >= 0  ~
  -1 zmeter_counter -1 zsecond_counter  + 63 >= 0  ~
 -1 zmeter_counter  + 33 >= 0  ~
  + 1 zmeter_counter -1 zsecond_counter -1 >= 0  ~
 + 1 zmeter_counter -12 >= 0  ~
 -1 zmeter_counter  + 3 zsecond_counter >= 0
```

On peut voir que le nombre de mètres parcouru (*zmeter_counter*) est compris entre 12 et 31 pour *Acc*17, entre 12 et 32 pour *Acc*6, et entre 12 et 33 pour *Acc*26. Après l'application du widening, on remarque que le nombre de mètres parcouru est supérieur à 12 pour *Acc*17, *Acc*6 et *Acc*26 : il n'y a pas de borne supérieure. Si on n'avait pas appliqué l'opérateur d'élargissement, alors la borne supérieure n'aurait pas cessé d'augmenter.

Chapitre 6

Description de la structure du programme

Cette partie donne en détails la structure du programme, ainsi que le rôle de chaque classe que j'ai codé. Cette partie n'est intéressante à lire seulement si vous avez l'intention de reprendre mon code.

6.1 L'analyseur réalisé avec SmartTools

J'ai défini 4 visiteurs :

ConstantExpressionVisitor : il évalue les expressions constantes. Lorsque l'on se trouve sur un noeud de type *ConstantTree*, on remonte ce noeud. Lorsque l'on se trouve sur un noeud de type *MONO_OPPhylum* et que l'expression est constante, on l'évalue et on retourne un arbre *ConstantTree* lui correspondant. Au niveau des arbres de type *AffectTree*, *MonoOpTree*, *WhenTree* et *DefaultTree*, lorsque la visite de leurs fils rend un arbre (*ConstantTree*), on remplace ce fils par le résultat de la visite.

MemVarVisitor : il permet à la fois de récolter des informations sur les variables du programme, et de générer des variables intermédiaires lorsque les affectations comportent plus d'un opérateur. Pour générer les variables intermédiaires, on leur donne un nom non encore utilisé (du type $var \sim i$) grâce à la classe *FreshName*. De plus, dans les noeuds de type *EXPPhylum*, si le père n'est pas de type *AffectTree*, alors on se trouve dans le cas d'une sous-expression et on annote le noeud par une variable intermédiaire. En outre, dans les noeuds de type *DelayTree* et *InitTree* (dont le premier fils est de type *DelayTree*) on génère une variable dont le nom est de la forme <variable_initiale>$<indice_de_retard> et on annote le noeud avec ce nom. Enfin, pour récolter les informations sur les variables du programme, une table de variables est utilisée comme attribut de ce visiteur. Voyons maintenant les structures de données qui compose cette table.

Variable : les objets de cette classe permettent de conservées les informations suivantes : nom, type, valeur initiale si elle existe et rang dans le polyèdre ($\mathcal{P}_i$) d'une variable. Une méthode permet d'obtenir son rang dans les matrices de conversion et de projection, car dans

un polyèdre, le rang dans une contrainte commence à 1, par contre il commence à 0 dans une matrice.

DelayVariable : les objets de cette classe héritent de la précédente. On y mémorise des informations supplémentaires propres aux variables de retard : variable initiale, indice de retard et rang dans le polyèdre (Acc_i). Comme pour les objets *Variable*, une méthode permet d'obtenir le rang (spécifique aux variables de retard) dans les matrices de conversion et de projection. Une autre méthode (*toDelayVariable*) permet à partir d'une variable de retard et d'une variable, de remplacer le nom de la variable de retard par le nom de la variable. On utilise cette méthode dans le cas où on affecte à une variable une variable de retard. Par exemple, pour l'affectation $x := y \$ 1$ init 4, on annote le noeud *InitTree* avec la variable fraîche *y$1*, mais comme cette variable correspond aussi à x, on modifie le nom de la variable *y$1* par le nom x.

VariableTable : cette classe représente une table contenant des objets de type *Variable* : la clé correspond au nom de la variable mémorisée. On conserve dans cette classe le dernier rang affecté à une variable : la méthode *newRank* permet d'obtenir un nouveau rang dans le polyèdre (rang non encore affecté). Les deux méthodes *addRank* et *addDelayRank* permettent, une fois la table remplie, de donner les deux rangs aux variables de retard qui n'en n'avaient pas encore. Cette classe possède bien d'autres méthodes dont leur rôle est expliqué dans les commentaires du code.

PolyhedronVisitor : ce visiteur permet d'extraire les contraintes sous forme de polyèdres suivant l'algorithme présenté en 3.1. Les principaux attributs utilisés sont : un tableau (*Vector*) d'ensembles d'ensembles de contraintes et une variable qui est la variable à affecter. Lorsque l'on se trouve sur un noeud *AffectTree*, on donne à son second fils les attributs : la variable à affecter et le tableau des ensembles déjà produits. Lorsque l'on se trouve sur un noeud de type *EXPPhylum*, si le ou les opérandes sont de type *EXPPhylum*, on récupère la valeur de l'annotation qui se trouve dans le ou les fils pour connaître les variables sur lesquels l'opérateur est appliqué. Dans ce même cas, on donne aussi comme attribut (variable) aux fils, la valeur de l'annotation de ces fils, qui correspond à la variable à affecter dans la sous-expression. Pour ce visiteur, dans les noeuds cités précédemment, on construit les ensembles d'ensembles de contraintes grâce aux structures de données que je vais décrire.

SetSetsConstraints : cette classe représente un ensemble d'ensembles de contraintes, avec un ensemble de contraintes qui est représenté par la classe *SetConstraints*.

SetConstraints : cette classe contient les contraintes de l'ensemble, le polyèdre correspondant à ces contraintes, les ensembles de variables absentes et présentes, et la dimension du polyèdre, ie le nombre total de variables du programme source plus les variables intermédiaires. Un objet de type *SetConstraints* peut être soit sous la forme d'un ensemble de contraintes, soit sous la forme d'un polyèdre, soit sous les deux formes. Cette classe possède entre autres les méthodes : *polyToSet* pour qu'à partir de la forme sous polyèdre on puisse obtenir la

forme sous ensemble de contraintes, *union* qui calcule l'union de deux ensembles de contraintes (ie intersection de polyèdres). Cette dernière méthode rend un objet de type *EmptySetConstraints* si les intersections entre les ensembles de variables présentes et les ensembles de variables absentes ne sont pas vides.

EmptySetConstraints : représente l'ensemble vide. C'est à dire qu'il n'y a pas de solution qui satisfasse les contraintes contenues dans cet ensemble.

Polyhedron : cette classe représente un polyèdre. Elle constitue l'interface entre Java et la librairie C de polyèdres. L'information qui y est conservée, est un pointeur (sous forme d'entier) sur un polyèdre C. Les méthodes la constituant, permettent de créer le polyèdre vide et le polyèdre universel, de créer un polyèdre à partir d'un tableau de contraintes, d'obtenir un tableau de contraintes à partir d'un polyèdre, et d'effectuer des opérations sur les polyèdres comme l'union, l'intersection, le test d'inclusion, ... La seconde partie de cette classe permet d'implémenter l'opérateur d'élargissement, elle est décrite dans la section suivante. Le fichier *analyseur_signal_Polyhedron.h* et *PolyhedronImp.c* constitue le code C des méthodes natives contenues dans cette classe.

Memory : cette classe hérite de *SetConstraints*. Elle permet de représenter un état de la mémoire à la fois sous la forme d'un polyèdre et sous la forme de contraintes. Dans cette classe les contraintes ne portent que sur les variables de retard, on exprime seulement le fait qu'une variable de retard est égale à sa valeur initiale.

SignalEquationsVisitor : ce visiteur permet d'afficher les équations SIGNAL sur lesquels l'analyseur travaille. C'est à dire, qu'il permet d'afficher les équations du programme, modifiées par la génération des affectations intermédiaires. Pour pouvoir lancer ce visiteur, il faut tout d'abord que les variables intermédiaires aient été générées, et donc que le visiteur *MemVarVisitor* ait été lancé avant.

De plus, j'ai défini 4 actionneurs (lanceurs de visiteurs) :

VariableInformationVisitorAction : affiche dans le fichier *variables.txt* la table des variables produite par le visiteur *MemVarVisitor*.

SignalEquationsVisitorAction : affiche dans le fichier *equations.txt* les équations SIGNAL récupérées grâce au visiteur *SignalEquationsVisitor*.

AnalyserVisitorAction : lance tout d'abord le visiteur *ConstantExpressionVisitor*, puis lance le visiteur *MemVarVisitor*, puis lance le visiteur *PolyhedronVisitor* avec comme paramètre la table des variables, et enfin, produit tous les fichiers nécessaires pour pouvoir lancer REQS. Certains fichiers sont le résultat de la sérialisation d'objets créés lors de cette analyse. Les fichiers fournis à REQS sont :

init.tmp : représente l'état initial de la mémoire (variables de retard) sous forme de contraintes. Le rang dans ces contraintes des variables de retard est *DelayRank*.

constraints.tmp : représente les ensembles de contraintes qui décrivent chacun un comportement possible du programme source.

projections.tmp : représente les projections sous forme de matrice. Il

y en a une par ensemble de contraintes. Les variables de retard sont sur les lignes et les variables sur les colonnes.

conversion.tmp : représente la matrice de conversion. Elle permet de passer d'un polyèdre de dimension, le nombre de variables de retard, à un polyèdre de dimension, le nombre de variables.

equations.tmp : contient les équations que REQS doit résoudre.

rankTable.tmp : tableau de correspondance entre le nom d'une variable de retard et le rang de cette variable de retard dans un polyèdre représentant un état de mémoire.

PrintConstraintSetVisitorAction : permet d'afficher dans le fichier *result.txt* les ensembles de contraintes produits par le visiteur *PolyhedronVisitor*.

Je finirai par présenter les classes dont je n'ai pas encore fait allusion :

Projection : représente une projection sous forme d'une matrice. Cette classe possède des méthodes qui permettent de construire une projection à partir d'un ensemble de contraintes.

Functions : cette classe contient diverses méthodes utiles. La méthode *allCombinations* permet de calculer les comportements possibles d'un programme source à partir des ensembles d'ensembles de contraintes produits en analysant les équations SIGNAL (application de l'algorithme 3.1 sur l'opérateur *Parallel*). La méthode *makeArray* transforme un *Vector* de *SetConstraints* en un tableau de *SetConstraints* duquel on retire l'ensemble de contraintes dans lequel toutes les variables sont absentes. La méthode *addDelayVar* permet, pour les variables de retard dont l'indice est supérieur à 1, de générer autant de variables de retard intermédiaires nécessaires pour que toutes les variables de retard soient à la fin avec un retard de 1 sur une variable initiale. Si l'on ne fait pas appel à cette méthode, les projections poseront un problème. La méthode *makeRankTable* permet de créer le tableau de correspondance entre rang et variables de retard. Enfin, les méthodes *makeEquations* et *makeEquationsW* rendent sous forme de chaînes de caractères les équations pour REQS. La seconde méthode permet l'utilisation d'un opérateur d'élargissement.

En ce qui concerne les classes *ConstantData*, *DoubleEntryException*, *ErrorClass* et *VisitorData*, elle sont très simples.

6.2 Le solveur REQS

Dans REQS, il faut définir le nouveau domaine sur lequel on veut travailler. Il faut spécifier dans la classe *operateurStrictness* (du répertoire *domain*) les opérations définies sur le domaine *Polyhedron*. Il faut aussi définir la classe *treillisPolyhedron* (du répertoire *treillis*) qui décrit ce qu'est un treillis de polyèdres. On y donne le code pour chaque opérations. Les opérations définies sur les polyèdres (éléments du treillis) sont :

inferieur : inclusion de polyèdre.

equals : égalité de polyèdre.

INTER : intersection de polyèdre.

UNION : union de polyèdre.

NEXT : projection définie en 3.2 page 15.

INIT : création du polyèdre représentant l'état initial de la mémoire.

bottom : polyèdre vide.

WIDEN : élargissement d'un polyèdre à partir d'un certain nombre d'itérations. Il possède 4 arguments : une variable (Acc_i), le numéro de cette variable (i), l'expression qui représente ceux à quoi est égale la variable et le nombre d'itérations après lequel on applique l'opérateur d'élargissement. Par exemple : $Acc0 =$ (WIDEN $Acc0$ 0 (UNION (...) ($ldots$)) 10)

La première fois que l'on construit un treillis de polyèdres, il faut récupérer toutes les informations nécessaires : l'état initial de la mémoire, les polyèdres décrivant les comportements possibles du programme, les projections, la matrice de conversion, la table de correspondance entre les variables de retard et leur rang dans le polyèdre, et les équations que REQS doit résoudre. Je vais maintenant décrire les classes utilisées pour implémenter cet opérateur :

Vertex : représente un sommet de polyèdre. Cette classe conserve les coordonnées d'un point. On y trouve entre autres une méthode permettant de construire un rayon à partir de deux sommets.

SetVertex : représente un ensemble de sommets. Cette classe possède la méthode *evolvedVertices* qui permet de calculer deux ensembles de sommets correspondant à la différence entre deux ensembles. Elle possède aussi la méthode *nearestVertex* qui permet de trouver dans un ensemble de sommets, le sommet le plus proche (au niveau de la distance) d'un sommet donné.

Enfin, la seconde partie de la classe *Polyhedron* conserve les données suivantes : l'ensemble des sommets, l'ensemble des lignes et l'ensemble des rayons d'un polyèdre. Cette classe possède des méthodes pour passer de ces trois ensembles à un polyèdre, pour passer d'un polyèdre à une représentation sous forme d'ensembles de sommets, rayons et lignes, des méthodes pour créer un nouveau polyèdre à partir d'un autre en lui ajoutant soit un rayon, soit une ligne, ...

Pour exécuter ce programme, il suffit d'utiliser la commande *./launch* qui lance la production des contraintes, puis lance la résolution des équations REQS. La classe *LaunchAnalyser* permet de lancer SmartTools sans passer par l'interface graphique.

Conclusion

En conclusion, il est assez facile de réaliser un analyseur avec SmartTools. Ici (dans SmartTools), la syntaxe abstraite joue le rôle de représentation intermédiaire des équations du programme sur laquelle on travaille pour extraire les contraintes. De plus, l'utilisation des visiteurs permet de décomposer les actions en plusieurs étapes. La communication entre SmartTools et REQS s'est faite ici, seulement par fichiers. Dans l'avenir, il est envisagé de communiquer directement par objets Java pour utiliser REQS (ou peut-être de définir une interface xml). Cette analyse a été testée sur quelques programmes assez simples, car il est toujours difficile de trouver des exemples complexes définis seulement sur des domaines simples (entiers et booléens). Lors de la résolution des équations récursives, il y a parfois des problèmes de débordement pour la multiplication. Ce cas s'est produit sur un exemple (*subway*). L'implémentation de l'opérateur d'élargissement est correcte, mais elle n'est pas aussi précise qu'on le souhaiterait. En effet, il faudrait éviter le choix arbitraire d'un sommet, et il faudrait aussi prendre en compte l'évolution des rayons.

Je remercie, pour le temps accorder à répondre à mes questions, Thomas Jensen, Frédéric Besson et Florimond Ployette de l'équipe LANDE. Je remercie aussi l'équipe des Smarties: Didier Parigot, Frank Chalaux, Carine Courbis, Pascal Degenne et Alexandre Fau, pour m'avoir accueillie deux jours à Sophia Antipolis.

Bibliographie

[1] Frédéric Besson, Thomas Jensen, and Jean-Pierre Talpin. Polyhedral analysis for synchronous languages. In Agostino Cortesi and Gilberto Filé, editors, *Static Analysis*, volume 1694 of *Lecture Notes in Computer Science*, pages 51–68. Springer, 1999.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns, elements of reusable object-oriented software.* Addison-Wesley, 1995.

[3] Bernard Houssais. Cours de programmation en langage temps-réel signal. Technical report, Irisa, Institut de Recherche en Informatique et Systèmes Aléatoires - projet EP-ATR, 1999.

[4] Christelle Lecomte. Manuel d'utilisation de smarttools pour la réalisation d'analyseurs. Projet Lande, août 2000.

[5] D. K. Wilde. A library for doing polyhedral operations. Technical Report RR-2157, Inria, Institut National de Recherche en Informatique et en Automatique, 1993.

Annexe A

Syntaxe abstraite de SIGNAL

```
Formalism of signal is
Root is PROG;
   PROG =
      process(INSIGLIST inVariables, OUTSIGLIST outVariables,
                EQNLIST equations, LOCALLIST localsVariables);

  INSIGLIST =
        SIGLIST;
  OUTSIGLIST =
        SIGLIST;
  SIGLIST =
      NONE,
      signals(SIG[] variables);
  SIG =
      decl(TYPE type, VARLIST varList);
  VARLIST =
      varl(VAR[] variables);

  LOCALLIST =
      NONE,
      locals(LOCAL[] variables);
  LOCAL =
      declLocals(TYPE type, IDENTLIST varList);
  IDENTLIST =
      initIdentl(IDENT[] variables);
  IDENT =
      VAR,
      initIdent(VAR variable, CONST value);

  EQNLIST =
      equationList(EQN[] eqlist);
  EQN =
```

```
    synchro(VAR left, VAR right) is {"^=", , },
    affect(VAR variable, EXP exp) is {":=", , };
  EXP =
    constant(CONST value),
    VAR,
    clock(VAR variable) is {"^", , },
    delay(VAR variable, NUMBER number) is {"$", , },
    monoOp(MONO_OP oper, ARGUMENT args),
    when(ARGUMENT args) is {"when", , },
    default(EXP left, EXP right) is {"default", , },
    init(EXP variable, CONST value) is {"init", , };
  ARGUMENT =
    unary(EXP exp),
    binary(EXP left, EXP right);
  MONO_OP =
    not() is {"not", , },
    or() is {"or", , },
    and() is {"and", , },
    plus() is {"+", , },
    minus() is {"-", , },
    multiply() is {"*", , },
    divide() is {"/", , },
    modulo() is {"modulo", , },
    power() is {"**", , },
    less() is {"<", , },
    lesseq() is {"<=", , },
    greater() is {">", , },
    greatereq() is {">=", , },
    equal() is {"=", , },
    diff() is {"/=", , };
  VAR =
    var as STRING;
  CONST =
    number as STRING,
    bool as STRING;
  NUMBER =
    numConst as STRING;
  TYPE =
    intType as STRING,
    boolType as STRING;
  NONE =
    none();
End
```

En ce qui concerne l'opérateur *when* et l'opérateur *monoOp*, il aurait été préférable de les séparer en deux et d'élimimer le phylum *ARGUMENT* : des opérateurs pour les opérations binaires (*binaryWhen* et *binaryMonoOp*) et des opérateurs pour les opérations unaires (*unaryWhen* et *unaryMonoOp*). En effet, lors du parcours de ces opérateurs, pour le fils de type *ARGUMENTPhylum*, il faut à chaque fois faire deux cas : unaire ou binaire.