View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by INRIA a CCSD electronic archive server

Vérification sémantique de spécifications métallurgiques

Francis Klay, Eric Domenjoud, Claude Kirchner

▶ To cite this version:

Francis Klay, Eric Domenjoud, Claude Kirchner. Vérification sémantique de spécifications métallurgiques. [Rapport de recherche] RR-2226, INRIA. 1994. inria-00074444

HAL Id: inria-00074444
https://hal.inria.fr/inria-00074444

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Vérification sémantique de spécifications métallurgiques

Francis Klay
Eric DOMENJOUD · Claude KIRCHNER

N° 2226
Mars 1994

PROGRAMME 2

Calcul symbolique,
programmation
et génie logiciel

Rapport de recherche

1994

Vérification sémantique de spécifications métallurgiques

Semantic verification of metallurgic specifications

Francis Klay, Eric Domenjoud, Claude Kirchner.

INRIA Lorraine & CRIN
BP 101 F-54602 Villers-Lès-Nancy
FRANCE
E-mail: Francis.Klay@loria.fr, Eric.Domenjoud@loria.fr, Claude.Kirchner@loria.fr

12 mars 1994

Résumé

Dans ce document nous présentons d'une part une méthode pour vérifier certaines propriétés sémantiques sur des spécifications décrites par des règles, et d'autre part l'implantation de cette méthode dans un programme appelé Versem. Ce travail a été réalisé à la demande de l'entreprise Sollac qui utilise de telles spécifications pour formaliser des processus de transformation métallurgique.

Abstract

In this report we are presenting a method for verifying some properties of rule based specifications, and its implementation in a program called Versem. This work has been realized to answer the need of the Sollac compagny that uses such specifications for describing metallurgic transformation processes.

Table des matières

1 Introduction 3

2 Présentation informelle syntaxe et sémantique 6
2.1 Règles 6
2.2 Spécification 6
2.2.1 Déclaration du module 6
2.2.2 Types 7
2.2.3 Variables 7
2.2.4 Termes 7
2.2.5 Formule logique 8
2.2.6 Préconditions 8
2.2.7 Règles 8
2.2.8 Généralités 8
2.3 Sémantique 9
2.4 Vérifications effectuées 10
2.5 Hypothèses de travail 11

3 Syntaxe formelle 12

4 Aspects théoriques de la méthode 14
4.1 Définitions 14
4.2 Syntaxe abstraite d'une spécification 15
4.3 Sémantique d'une spécification 18
4.4 Formule logique et spécification 20
4.5 Vérification 22
4.5.1 Test d'ambiguïté 22
4.6 Test de non définition 22
4.7 Table de dépendances 23
4.8 Test de satisfaisabilité des contraintes réelles linéaires. 24
4.8.1 Réduction du problème par les égalités 24
4.8.2 Recherche d'une solution aux inéquations 24
4.8.3 Recherche des équations implicites 25

5 Exemples d'exécution 28
5.1 Exemple pour le test de variable ambiguë 28
5.2 Exemple pour le test de variable indéfinie 33

6 Description de l'implantation 34

7 Erreurs non traitées actuellement 37

8 Conclusion 38

A Description des fichiers 39

B Primitives des librairies g++ utilisé 41

1 Introduction

Dans les aciéries, le minerai subit une suite de transformations qui le convertissent en acier. Ce processus est contrôlé par des programmes qui décrivent les opérations nécessaires pour chacune des étapes. Afin de fixer les idées considérons la figure 1, dans ce cas d'école le but est de refroidir un bloc de métal en fonction de son poids avec de l'eau. Le département de production définit comment la quantité d'eau est calculée, et ensuite le département informatique écrit le programme qui contrôlera le processus de refroidissement. Une fois que ces opérations sont effectuées la production peut débuter.

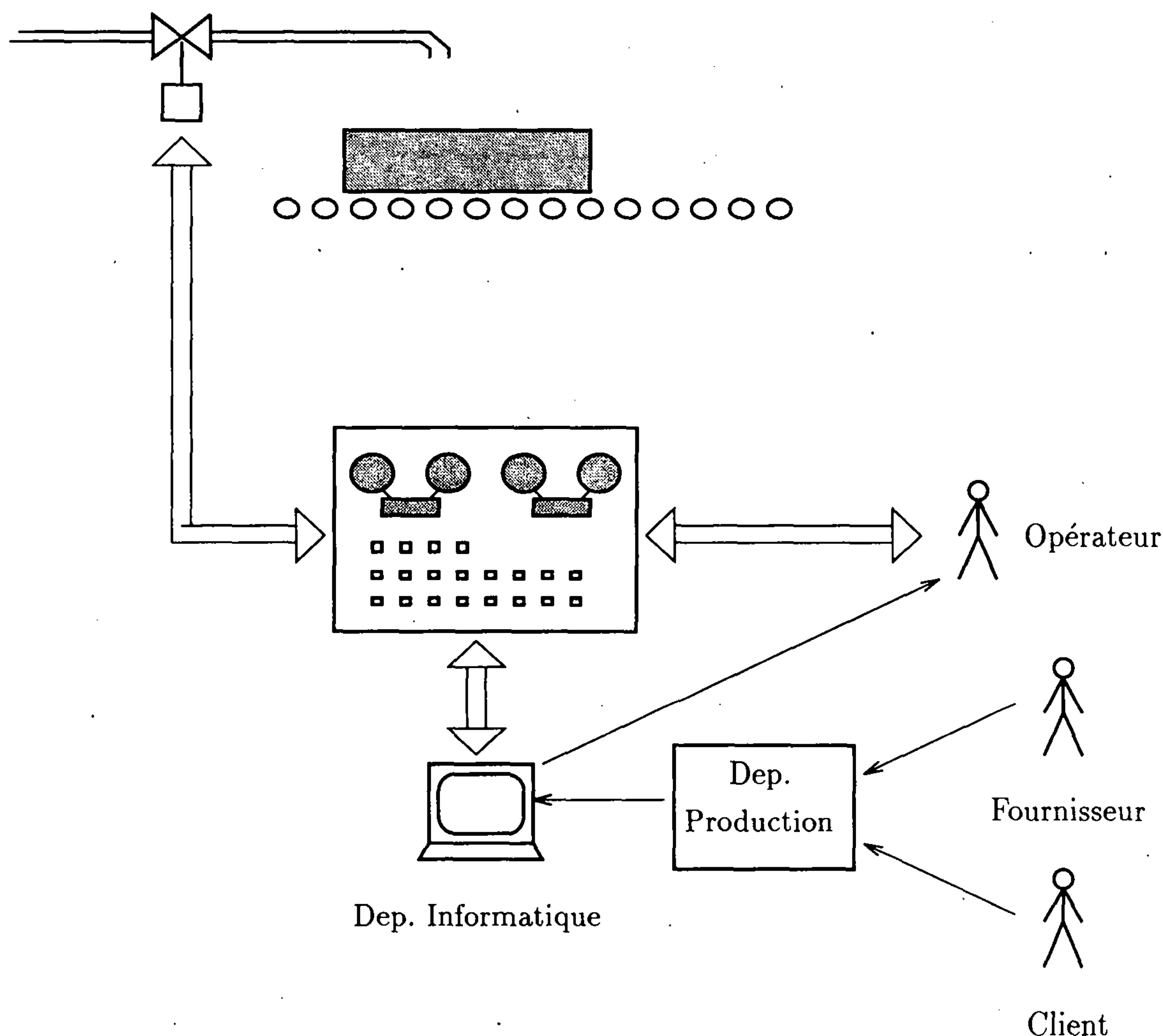


FIG. 1 - *Situation initiale*

Les différentes transformations impliquées dépendent étroitement du produit initial et du produit final désiré. C'est pourquoi, pour chaque cas de production, il est nécessaire de modifier les programmes informatiques. Cet état de fait pose de nombreux problèmes en pratique : la maintenance et la modification des différents cas de productions existants est difficile, il n'est pas possible de vérifier automatiquement la correction des modifications, tout changement de la production demande l'intervention d'un informaticien, etc.
Pour palier à ces inconvénients l'entreprise Sollac à effectué le travail suivant. Le processus de contrôle a été analysé de façon à en extraire les parties sujettes à de fréquentes modifications.

Après cela, une syntaxe à base de règles à été définie pour décrire ces parties du contrôle de façon naturelle et agréable. Finalement, un outil pour éditer et traduire les règles en programme informatique a été développé. Pour notre cas d'école la nouvelle situation est représentée dans la figure 2 ou le calcul du volume d'eau apparaît clairement.

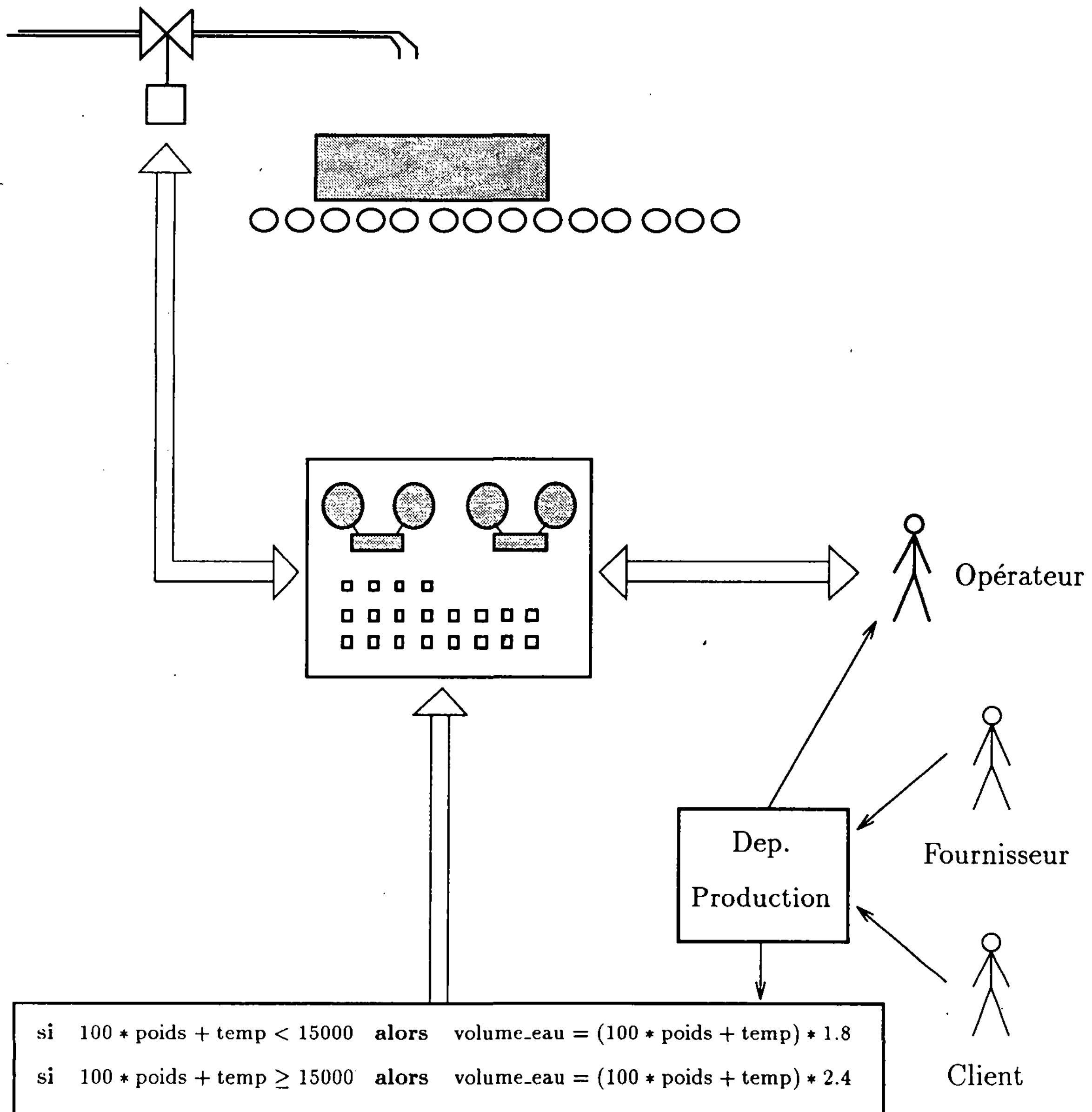


FIG. 2 - *Nouvelle situation*

Intuitivement, exécuter une liste de règles revient à les interpréter **séquentiellement**. A ce stade il serait intéressant de pouvoir **garantir que l'exécution d'une liste de règles ne portera atteinte ni aux personnes ni au matériel**, c'est précisément ce point qui est l'objet du travail présenté dans ce document. Si on reprend notre exemple, il y a principalement deux types d'erreurs à détecter.

- Si la condition de la première règle était $100 * \text{poids} + \text{temp} < 18000$, il y aurait problème puisque pour $15000 \leq 100 * \text{poids} + \text{temp} < 18000$ le volume d'eau serait **ambigu** car deux règles se déclencheraient et fixeraient 2 volumes d'eau différents, ce qui a priori est

probablement le résultat d'une erreur de spécification.

- Si la condition de la première règle était $100 * \text{poids} + \text{temp} < 12000$, il y aurait problème puisque pour $12000 \leq 100 * \text{poids} + \text{temp} < 15000$ le volume d'eau serait **indéfini** car aucune règle ne se déclencherait.

Comme on le verra plus loin, les listes de règles définies ont une expressivité plus faible qu'un langage informatique classique. Cette restriction est intéressante car elle rend possible la détection statique de ces deux types d'erreur.
Ce document est structuré comme suit. La section 2 contient une présentation informelle du langage de spécification utilisé, dans cette même section les différents tests à effectuer et les hypothèses de travail sont présentées. Une vision intuitive du problème étant acquise à ce stade, nous donnons dans les sections 3 et 4 une description formelle du problème et de la méthode utilisée pour le résoudre. Afin de fixer les idées un exemple est complètement déroulé dans la section 5 et dans la section 6 la structure générale du programme est décrite. Finalement l'annexe A contient une description des fichiers et l'annexe B la liste des primitives de la librairie g++ utilisées.

2 Présentation informelle syntaxe et sémantique

Comme nous l'avons vu dans l'introduction, la description des processus de transformation est faite essentiellement avec des règles. Dans cette partie nous allons décrire plus finement comment une telle spécification est structurée et quel est l'effet de son exécution.

2.1 Règles

Une **règle** est une expression de la forme :

si *condition* **alors** $action_1$ **et** $action_2$ **et** ... **et** $action_n$;

où *condition* est une expression booléenne et où pour tout $i \in [1..n]$, $action_i$ est soit une affectation soit un appel de procédure. Notons que dans ce cas tout contrôle sémantique est exclu sur ce que fait la procédure, mais on pourrait penser à ajouter des pré et post conditions. Par la suite, on appellera **conclusion** de la règle, la conjonction $action_1$ **et** $action_2$ **et** ... **et** $action_n$. Le comportement opérationnel d'une règle correspond à l'évaluation de la conclusion si la condition est vraie, si la condition est fausse alors l'exécution de la règle est sans effet. Toute action est évaluée pour les valeurs des variables avant le déclenchement de la règle. De plus, remarquons que le ";" fait office de terminateur de règle. Une règle peut se résumer à sa conclusion, dans ce cas elle sera toujours déclenchée.

Exemple 1 *Dans la règle ci-dessous Proc est une procédure, f est une fonction,* **N2_visé** *et* **Type_fonte** *sont des variables d'entrée de la règle alors que* **VNR2** *et* **N2** *sont des variables de sortie.*

si *Type_fonte = 'H'* $\wedge$ *N2_visé* > 40 **alors** *VNR2 = VNR2* $+ 10$ **et** *N2 = f(N2_visé)* **et** *Proc(N2);*

2.2 Spécification

Les règles seules ne suffisent pas à décrire complètement un processus de transformation. En fait il faut rajouter d'autres informations telles que les déclarations de types, de variables, etc. Un module comportant toute l'information décrivant un processus est appelé une **spécification**, cette dernière a la structure générale suivante :

Déclaration du module
Déclaration des types
Déclaration des variables
Préconditions
Règles

Dans la suite le caractère " délimite les symboles introduits par l'utilisateur. Les expressions introduites par l'utilisateur sont quant à elles délimitées par le caractère '.

2.2.1 Déclaration du module

Une déclaration de module est une expression de la forme :

module: "nom du module"

La déclaration de module sert essentiellement à donner un nom à la spécification afin de pouvoir la manipuler dans une extension future du langage (inclusion de module, etc.).

2.2.2 Types

Toute spécification contient le type prédéfini **reel** qui symbolise l'ensemble des nombres réels. L'ensemble des déclarations de types est préfixé par le mot réservé **types:**. Une déclaration de type peut avoir trois formes :

"Type2" = "Type1"

Après une déclaration de cette forme "Type2" est un synonyme de "Type1". La seconde forme de déclaration permet de définir les ensembles **finis** par énumération de leurs éléments, un tel type est parfois appelé **type énuméré**:

"Type" = < "a", "b", "c", ... >

Après une déclaration de cette forme le symbole "Type" est connu et son domaine est {"*a*", "*b*", "*c*", ...}. La dernière forme de déclaration permet de déclarer des **types contraints**, c'est à dire un type dont le domaine est un sous-ensemble du domaine d'un type déjà connu :

"Type2" = "Type1" [**'Formule logique anonyme'**]

Après une déclaration de cette forme, le domaine de "Type2" contient les éléments de "Type1" qui satisfont la 'Formule logique anonyme' (ces formules sont définies dans la sous-section 2.2.5 ci-dessous). Si on désire par exemple définir les réels positifs, la déclaration est de la forme :

positif = **reel**[0 <= @]

Le type **reel** et les types énumérés sont des types **initiaux.** Le **type initial** d'un type T est T si T est un type initial. Le type initial de T est le type initial de T1 si la déclaration de T est de la forme :

"T" = "T1" ou **"T" = "T1"** [**'Formule logique anonyme'**]

2.2.3 Variables

Toute spécifications a une variable prédéfinie par type, cette variable appelée **variable anonyme** est notée @. Une variable anonyme ne peut être utilisée que dans la formule logique définissant un type contraint. Une variable est appelée **variable d'entrée** si sa valeur est supposée connue au début de l'exécution de la spécification, dans le cas contraire on dit qu'il s'agit d'une **variable de sortie.** L'ensemble des déclarations des variables d'entrée (resp. sortie) est préfixé par le mot réservé **entrees:** (resp. **sorties:**). Une déclaration de variables a la forme suivante :

"x", "y", ... : 'Type'

où 'Type' est soit un symbole de type déjà déclaré soit une expression légale à droite du symbole = dans les déclarations de type. Après cette déclaration les variables "x", "y", ...sont connues et de type 'Type'.

2.2.4 Termes

Un **terme** est un terme de type énuméré ou un terme de type réel. Un **terme énuméré** est soit un élément d'un type énuméré soit une variable dont le type initial est un type énuméré. Un **terme réel** est soit une constante réelle soit une variable dont le type initial est **reel** soit une forme linéaire dont toutes les variables sont des termes réels. La notion de type initial est naturellement étendue aux termes.

2.2.5 Formule logique

Une **formule logique** est une expression booléenne classique construite à l'aide des **connecteurs** réservés **et**, **ou**, **non** et des **prédicats** réservés =, ! =, <, <=, >, >=. Les deux arguments d'un prédicat doivent avoir le même type initial et seuls les prédicats =, ! = sont autorisés sur les termes énumérés.
Une formule logique est dite **anonyme** si la seule variable qu'elle contient est @. Une formule anonyme est **satisfaite** pour a si la formule est vraie lorsque toute les occurences de @ sont remplacées par a. Notons que dans les déclarations de type contraint le type de @ préfixe la formule logique anonyme.

2.2.6 Préconditions

Une précondition est une propriété qui est supposée vérifiée lorsque la spécification est exécutée. L'ensemble des préconditions est préfixée par le mot réservé **preconditions:**. Une préconditions est toujours de la forme suivante :

'Formule logique' ;

où ";" est le terminateur de précondition. Les préconditions servent essentiellement à alléger la rédaction des conditions de règle. D'un autre point de vue elles permettent d'améliorer l'efficacité du vérificateur dont il sera question plus loin.

2.2.7 Règles

Pour suivre le contexte dans lequel elles sont utilisées par Sollac, la liste des règles est subdivisée en trois groupes. On a d'abord le groupe des **règles de base** qui décrivent le traitement "standard". Le second groupe est celui des **règles d'exception**, ces dernières indiquent comment le traitement "standard" est modifié pour des données particulières. Finalement le dernier groupe est constitué des **règles de validation** qui ont un rôle de "garde fou", une règle de validation assure par exemple que la quantité de matière déversée dans un silo ne dépassera pas la capacité de ce dernier. La partie de la spécification contenant les règles a la structure suivante :

bases:
'liste de règles'
exceptions:
'liste de règles'
validations:
'liste de règles'

où 'liste de règles' est une suite de règles qui ont la forme décrite plus haut.

2.2.8 Généralités

Quelques remarques supplémentaires peuvent être faites à propos de la syntaxe :

- Tout symbole doit être déclaré avant de pouvoir être utilisé. Bien entendu cette remarque ne s'applique pas aux mots réservés et aux symboles prédéfinis.
- Les noms de symbole doivent commencer par une lettre suivie d'une chaîne de caractères composée de lettres de chiffres et du caractère "_".
- Un commentaire est inclus entre /* et */

- La déclaration d'un symbole S provoque un conflit de symbole si :
  - S est un mot réservé.
  - S est un type et il existe déjà un type S.
  - S est un élément de type énuméré ou une variable et il existe déjà un élément ou une variable S.
- Les déclarations d'intervalles sur les réels peuvent être abrégée par $[a..b]$, $[..b]$, $[a..]$ où a et b sont des constantes réelles. Si par exemple x est une variable plus grande que 10 on la déclare par $x : [10..]$.
- Les déclarations de types et de variables peuvent être mélangées à condition de préfixer chaque ensemble de déclarations par le mot clé approprié.

2.3 Sémantique

D'un point de vue opérationnel, **exécuter une spécification** revient à exécuter séquentiellement les règles de la spécification dans l'ordre où elles apparaissent. Notons que par définition il n'est pas possible d'appeler une règle depuis une conclusion donc chaque règle est exécutée au plus une fois pour toute exécution de la spécification.

Exemple 2 *La spécification ci-dessous calcule la mise au mille de la chaux à partir de l'analyse visée. En d'autres termes cette spécification décrit la valeur de MMCao en fonction de Si, P, S, P_visé et type_fonte.*

module: *exemple*

entrees:
$Si, P, S, P_visé : [0..]$
$type_fonte \quad :< H, P >$

sorties:
$MMCao : [0..]$

bases:
$MMCao = 0.0075 * Si + 0.06 * P - 16;$

exceptions:

si $S > 25$ **et** $S \leq 45$	**alors**	$MMCao = MMCao + 10;$
si $S > 45$	**alors**	$MMCao = MMCao + 20;$
si $P_visé \leq 17$	**alors**	$MMCao = MMCao + 15;$
si $P_visé > 17$ **et** $P_visé \leq 25$	**alors**	$MMCao = MMCao + 5;$
si $P_visé > 25$	**alors**	$;$

validations:

si $type_fonte = 'H'$ **et** $MMCao < 40$	**alors**	$MMCao = 40;$
si $type_fonte = 'H'$ **et** $MMCao > 80$	**alors**	$MMCao = 80;$
si $type_fonte = 'P'$ **et** $MMCao < 60$	**alors**	$MMCao = 60;$
si $type_fonte = 'P'$ **et** $MMCao > 120$	**alors**	$MMCao = 120;$

Supposons que cette spécification soit exécutée pour les valeurs suivantes :

$$S = 15 \qquad Si = 1200 \qquad P_visé = 10$$
$$P = 2100 \qquad type_fonte = 'P'$$

Dans ces conditions, la règle de base retourne MMCao = 119, *ensuite la troisième règle d'exception retourne MMCao* = 134 *finalement la dernière règle de validation ramène la valeur de MMCao à* 120.

Remarquons qu'en général, la valeur d'une variable d'entrée d'une spécification représente le résultat retourné par un module de calcul, une mesure physique ou une consigne donnée par un opérateur. La valeur d'une variable de sortie représente dans la plupart des cas la consigne donnée à un automate de contrôle.

Exemple 3 *Dans cet exemple nous montrons une spécification un peu plus compliquée syntaxiquement.*

```
/* Petit exemple de démonstration */

module:          exemple
types:           positif = reel [0 <= @]
entrees:         x, y : positif
                 z    :< a, b, c >
sorties:         v, w : positif
preconditions:   x < y + 2;
bases:
                 v = 2;
exceptions:
                 si x < 10               alors z = a et y = x - 1 et w = x;
                 si y > 10 et z != c     alors w = x + v;
                 si z = c                alors w = y;
```

2.4 Vérifications effectuées

Une idée de la sémantique des spécifications qui nous intéressent étant donnée ; dans cette partie nous décrivons les deux vérifications qui nous intéressent sur un cas moins trivial que celui de l'introduction. Soit x une variable de sortie d'une spécification S, de façon un peu plus formelle les deux sortes d'erreur sont données ci-dessous.

- Il a **erreur d'ambiguïté** sur x s'il existe une exécution de S telle que x est assignée par au moins deux règles d'exception.
- Il a **erreur de non définition** sur x s'il existe une exécution de S telle que x n'est assignée par aucune règle de base et de validation.

Exemple 4 *Considérons la spécification suivante :*

module: *exemple*
entrees:
  $C, N2$: [0..]
sorties:
  Fer, CaO : [0..]
bases:

si	$N2 > 30$ **et** $C > 8$	**alors**	$Fer = 10;$
			$CaO = 5;$

exceptions:

si	$C < 10$	**alors**	$Fer = 4$ **et** $CaO = 1;$
si	$N2 > 40$	**alors**	$Fer = 5$ **et** $CaO = 2;$
si	$C < 8$ **et** $N2 < 20$	**alors**	$CaO = 4;$

Pour cet exemple la figure 3 donne les échecs de vérification relatifs à la variable de sortie Fer.

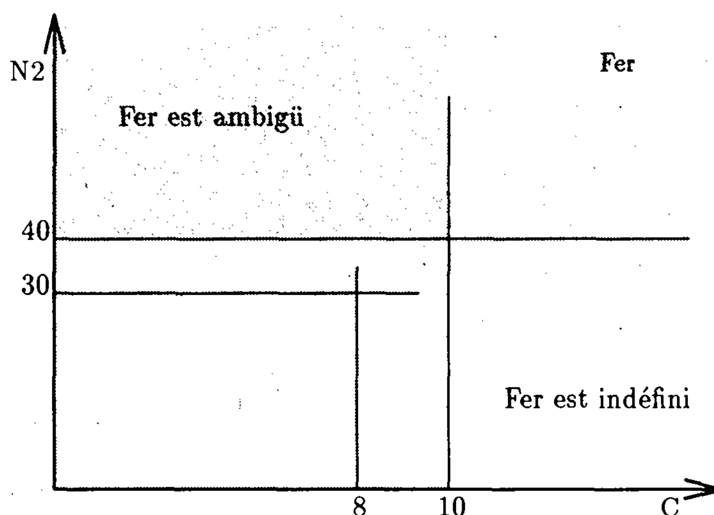


FIG. 3 - *Erreur pour la variable de sortie Fer*

2.5 Hypothèses de travail

Dans tout le reste de ce document les hypothèses de travail suivantes sont faites :

- L'ensemble de règles est exécuté séquentiellement. Cette hypothèse a déjà été donnée précédemment mais elle est répétée ici en raison de son importance.
- Les procédures et les fonctions sont ignorées pour l'instant par le vérificateur.

3 Syntaxe formelle

D'un point de vue syntaxique une spécification est un mot du langage algébrique décrit ci-dessous sous forme Backus Naur (grammaire BNF) où les non terminaux sont en italique et les terminaux en gras. Une définition des grammaires BNF pourra être trouvée dans tout livre traitant de la compilation ou dans certains ouvrages sur le langage Pascal.

module → **module:** {*decl_section*} *rules_sections*

decl_section → *type_sec* | *input_sec* | *output_sec* | *precond_sec*;

type_sec → **types:** {*type_decl*}
type_decl → **ident** = *type*
type → *real_range_type* | *cstr_type* | *enu_type* | **ident**
real_range_type → [$\textbf{real_const}_{opt}$ .. $\textbf{real_const}_{opt}$]
cstr_type → **ident** [*formula*]
enu_type → <*ident_list* >
ident_list → **ident** {, **ident**}

input_sec → **entree:** {*var_decl*}
output_sec → **sorties:** {*var_decl*}
var_decl → *ident_list* **:** *type*

precond_sec → **preconditions:** {*precond_decl*}
precond_decl → *formula* **;**

rules_sections → $base_sec_{opt}$ $excep_sec_{opt}$ $valid_sec_{opt}$
base_sec → **bases:** {*rule*}
excep_sec → **exceptions:** {*rule*}
valid_sec → **validations:** {*rule*}

rule → *actions* **;** | **si** *formula* **alors** *actions* **;**

actions → *action* {**et** *action*}
action → **ident** = *expr* | *function*

formula → *formula* **ou** *conjunction* | *conjunction*
conjunction → *conjunction* **et** *litteral* | *litteral*
litteral → **non** *litteral* | *atom* | (*formula*)
atom → *expr pred expr*
pred → = | < | > | <= | >= | ! =

expr → *expr* + *term* | *expr* − *term* | *term*
term → *term* * *fact* | *term* / *fact* | *fact*
fact → *ident* | **real_const** | − *fact* | (*expr*) | *function*
function → **ident** (*term_list*) | **ident** ()
term_list → *term* {, *term*}

Notons que cette grammaire autorise des formes non linéaires sur les réels, il est possible de modifier la grammaire pour tenir compte de ce fait mais il en coûte un alourdissement inutile. Une grammaire du même type peut être utilisée pour définir les terminaux qui ne sont pas donnée explicitement dans la grammaire précédente.

ident → *letter* {*alpha_num*}
alpha_num → *letter* | *digit*

letter	$\rightarrow$	**a** | **b** | ... | **z** | **A** | **B** | ... | **Z** | _
real_const	$\rightarrow$	*int_const* $frac_part_{opt}$ exp_part_{opt}
int_const	$\rightarrow$	*digit* {*digit*}
frac_part	$\rightarrow$	. *int_const*
exp_part	$\rightarrow$	it exp_prefix $sign_{opt}$*int_const*
exp_prefix	$\rightarrow$	**e** | **E**
sign	$\rightarrow$	− | +
digit	$\rightarrow$	**0** | **1** | ... | **9**

4 Aspects théoriques de la méthode

Dans la section 2 nous avons introduit informellement la sémantique d'une spécification, puis nous avons décrit intuitivement la méthode permettant d'effectuer les différents tests. Dans ce qui suit nous allons préciser ces notions, dans un premier temps nous définirons plus formellement la sémantique d'une spécification puis nous montrerons comment la formule logique associée à une spécification est obtenue. Nous prouverons ensuite que sous certaines conditions, toute solution de la formule correspond à une exécution de la spécification. Ce résultat étant établi nous pourrons montrer que la vérification d'une propriété sémantique se réduit à un test de satisfaisabilité.

4.1 Définitions

Avant de définir formellement la sémantique d'une spécification, quelques définitions préliminaires sont nécessaires. Le formalisme utilisé peut paraître lourd pour formaliser des notions qui s'expriment facilement en langage naturel. Ceci ne signifie pas que la formalisation soit inutile car elle a permis par exemple de mettre en évidence des cas qui avaient initialement été oubliés et c'est bien là un des buts essentiels de la formalisation !
Comme dans toute la suite de ce document, nous nous plaçons uniquement dans un cadre **multi-sorté** certaines conventions usuelles de notation sont parfois omises.
Soit $\mathcal{S} = \{s_1, \ldots, s_k\}$ un ensemble de **symboles de sortes** et $\mathcal{O} = \{f, g, h, \ldots\}$ un ensemble de **symboles d'opérateurs**. On appelle ensemble des **déclarations d'opérateurs** l'ensemble $\mathcal{F}$ contenant pour chaque opérateur f de $\mathcal{O}$ un unique **profil** de la forme $f : s_1, \ldots, s_n \rightarrow s$ tel que $s, s_i \in \mathcal{S}$. Le cardinal n définit **l'arité** de f qui sera notée $ar(f)$. Un opérateur d'arité 0 est appelé une **constante** et $\mathcal{F}_n$ désignera l'ensemble des déclarations d'opérateurs d'arité n. Etant donnés $\mathcal{S}$ et $\mathcal{F}$, on appelle **signature** Σ le couple $(\mathcal{S}, \mathcal{F})$. Les extensions naturelles des opérateurs ensemblistes seront utilisées sur les signatures et par abus de notation on écrira parfois $f \in \mathcal{F}$, $f \in \Sigma$ et $s \in \Sigma$ au lieu $f \in \mathcal{O}$ et $s \in \mathcal{S}$. Une **interprétation** de Σ est une **Σ-algèbre** $\mathcal{A}$ décrite par le couple $(\mathcal{S}^{\mathcal{A}}, \mathcal{F}^{\mathcal{A}})$ où :

- $\mathcal{S}^{\mathcal{A}}$ est le **support de l'algèbre** défini par l'union des éléments d'une famille $\{s^{\mathcal{A}} \mid s \in \mathcal{S}\}$.
- $\mathcal{F}^{\mathcal{A}}$ est une **famille de fonctions** $\{f^{\mathcal{A}} : s_1^{\mathcal{A}} \times \cdots \times s_n^{\mathcal{A}} \rightarrow s^{\mathcal{A}} \mid (f : s_1, \ldots, s_n \rightarrow s) \in \mathcal{F}\}$. Par la suite $Dom(f^{\mathcal{A}})$ désignera le **domaine** de $f^{\mathcal{A}}$.

Soit $\mathcal{A} = (\mathcal{S}^{\mathcal{A}}, \mathcal{F}^{\mathcal{A}})$ et $\mathcal{B} = (\mathcal{S}^{\mathcal{B}}, \mathcal{F}^{\mathcal{B}})$ deux Σ-algèbres. Une application $h : \mathcal{S}^{\mathcal{A}} \rightarrow \mathcal{S}^{\mathcal{B}}$ est une **Σ-application** si pour tout $s \in \mathcal{S}$ et pour tout $a \in s^{\mathcal{A}}$ on a $h(a) \in s^{\mathcal{B}}$. Une **Σ-application** $h : \mathcal{S}^{\mathcal{A}} \rightarrow \mathcal{S}^{\mathcal{B}}$ est un **homomorphisme** de Σ-algèbre si pour tout $f \in \mathcal{F}_n$ et pour tout $(a_1, \ldots, a_n) \in Dom(f^{\mathcal{A}})$ on a $h(f^{\mathcal{A}}(a_1, \ldots, a_n)) = f^{\mathcal{B}}(h(a_1), \ldots, h(a_n))$. Un homomorphisme bijectif est un **isomorphisme** et un homomorphisme d'une Σ-algèbre dans elle même est un **endomorphisme**.
Soit Σ une signature et $\mathcal{X}$ un ensemble dénombrable de variables tel que chaque variable x a une unique sorte s. On dit que t est un **Σ-terme** à variables dans X, de sorte s, noté $t : s$, si t est une variable de sorte s ou si $t = f(t_1, \ldots, t_n)$ et $(f : s_1, \ldots, s_n \rightarrow s) \in \mathcal{F}$ et $t_i : s_i$. Par définition l'ensemble des Σ-termes peut clairement être muni d'une structure de Σ-algèbre que l'on désignera par $\mathcal{T}(\Sigma, \mathcal{X})$, les variables seront notées v, x, y, z et les termes l, r, t, u. Par la suite on fera parfois la confusion entre algèbre et support pour les termes. L'**ensemble des variables** qui apparaissent dans un terme t sera désigné par $Var(t)$.
Etant donnés $\mathcal{T}(\Sigma, \mathcal{X})$ et une Σ-algèbre $\mathcal{A}$ on appelle **valuation** de X toute Σ-application $\rho : \mathcal{X} \rightarrow \mathcal{S}^{\mathcal{A}}$. L'ensemble des valuations de X sera désigné par $\mathcal{S}_{\mathcal{X}}^{\mathcal{A}}$. On peut montrer que ρ a un unique prolongement $\overline{\rho}$ par morphisme pour toute Σ-algèbre. Cette propriété s'exprime en disant que $\mathcal{T}(\Sigma, \mathcal{X})$ est à isomorphisme près **la Σ-algèbre libre** engendrée par $\mathcal{X}$ de la classe des Σ-algèbres. Etant donné un terme t de $\mathcal{T}(\Sigma, \mathcal{X})$ on appelle **interprétation** de t dans $\mathcal{A}$ la fonction

$t^{\mathcal{A}} : \mathcal{S}^{\mathcal{A}}_{\mathcal{X}} \rightarrow \mathcal{S}^{\mathcal{A}}$ telle que $t^{\mathcal{A}}(\rho) = \overline{\rho}(t)$. Par la suite une valuation sera désignée par une lettre grecque minuscule et on fera parfois la confusion entre une valuation et son prolongement par morphisme. La **composition** de valuations sera notée par juxtaposition et la **restriction** d'une valuation ρ à un ensemble de variables $\mathcal{V} \subseteq \mathcal{X}$, sera désignée par $\rho_{|V}$.
Etant donnée ρ une valuation $\mathcal{X} \rightarrow \mathcal{T}(\Sigma, \mathcal{X})$ qui est l'identité presque partout on appelle **substitution** l'endomorphisme qui prolonge ρ. *Id* désigne la **substitution identité**, une substitution bijective est un **renommage** et une substitution σ est **idempotente** si $\sigma^2 = \sigma$. Le **domaine** d'une substitution σ, noté $Dom(\sigma)$, est l'ensemble des variables x telles que $\sigma(x) \neq x$. Son **codomaine**, noté $Cod(\sigma)$ est l'ensemble $\{\sigma(x) | x \in Dom(\sigma)\}$, et son **image**, notée $Range(\sigma)$, est l'ensemble des variables apparaissant dans les termes de son codomaine, c'est-à-dire $\cup_{t \in Cod(\sigma)} Var(t)$.
La **restriction** de σ à un ensemble de variables $\mathcal{V} \subseteq \mathcal{X}$ est notée $\sigma_{|\mathcal{V}}$ et définie par $\sigma_{|\mathcal{V}}(x) = \sigma(x)$ si $x \in \mathcal{V}$ et $\sigma_{|\mathcal{V}}(x) = x$ sinon. Par la suite, une substitution σ sera souvent représentée par son **graphe** $\{x \mapsto \sigma(x) \mid x \in Dom(\sigma)\}$. Si $\mathcal{V} = (v_1, \ldots, v_n)$ est un n-uplet de variables toutes distinctes et $\mathcal{T} = (t_1, \ldots, t_n)$ est un n-uplet de termes, on utilisera parfois la notation condensée $\vec{v} \mapsto \vec{t}$ pour désigner la substitution $\{v_1 \mapsto t_1, \ldots, v_n \mapsto t_n\}$.
Soient Σ une signature telle que $bool \in \mathcal{S}$ et f un opérateur tel que $f : s_1, \ldots, s_n \rightarrow s \in \mathcal{F}$. Si pour tout i, $s_i = bool$ et $s = bool$ alors f est un **connecteur**, si pour tout i, $s_i \neq bool$ et $s = bool$ alors f est un **prédicat** et f est une **fonction** si pour tout i, $s_i \neq bool$ et $s \neq bool$. Par la suite les ensembles de déclarations de connecteurs, de prédicats et de fonctions seront désignés respectivement par $\mathcal{F}_c$, $\mathcal{F}_p$ et $\mathcal{F}_f$, ces ensembles induisent les signatures Σ_c, Σ_p et Σ_f dont les sortes sont celles qui apparaissent dans les déclarations d'opérateurs.
Soit Σ_c où $\mathcal{F}_c$ est un ensemble de déclarations de connecteurs usuels (faux : $\bot$, vrai : $\top$, non : $\neg$, et : $\wedge$, ou : $\vee$, implique : $\Longrightarrow$, ...). Pour exprimer les sens de ces connecteurs, il faut utiliser, non pas une Σ_c-algèbre quelconque, mais une algèbre qui définit le sens classique des connecteurs. Considérons par exemple la **signature booléenne** $\Sigma_b = (\{bool\}, \mathcal{F}_b)$ ou $\mathcal{F}_b$ est l'ensemble des déclarations de $\top$, $\bot$, $\neg$, $\wedge$ et $\vee$ dans ce cas l'interprétation de Σ_b doit être une **algèbre de Boole**, dans la suite on représentera le support de cette algèbre par $\mathbf{2} = \{0, 1\}$ qui suffit à exprimer une sémantique à deux valeurs (0 pour faux et 1 pour vrai).
Soit $\Sigma = \Sigma_b \cup \Sigma'$ une signature telle que Σ' ne contient pas de connecteur, une **interprétation booléenne** de Σ est une interprétation dont la restriction à Σ_b est une algèbre de Boole. Si $\mathcal{X}$ est un ensemble dénombrable de variables de sorte *bool* alors on appelle **formules de calcul des propositions** sur X les termes de $\mathcal{T}(\Sigma_b, \mathcal{X})$. Le choix de la signature utilisée pour définir les formules du calcul des propositions n'est pas unique. En fait la seule condition à satisfaire est que Σ_c soit une base, ce qui signifie essentiellement que pour toute application $f : \mathbf{2}^k \rightarrow \mathbf{2}$ il existe un terme t de $\mathcal{T}(\Sigma_c, \mathcal{X})$, tel que f soit codé par l'interprétation de t. Soient $\Sigma = \Sigma_b \cup \Sigma_r \cup \Sigma_f$ une signature et $\mathcal{X}$ un ensemble dénombrable de variables de sortes différentes de *bool*. Les **formules du premier ordre sans quantificateur** sont les éléments de $\mathcal{T}(\Sigma, \mathcal{X})$, comme précédemment le choix des connecteurs n'est pas unique. Par la suite, les éléments de $\mathcal{T}(\Sigma, \mathcal{X})$ seront simplement appelés formules logiques et désignés par des lettres grecques minuscules, de plus, l'ensemble de variables ou la signature seront parfois sous entendus. Soit α une formule logique de $\mathcal{T}(\Sigma, \mathcal{X})$. Une interprétation $\mathcal{A}$ de Σ_c **satisfait** α si pour toute valuation ρ on a $\alpha^{\mathcal{A}}(\rho) = 1$, α est **valide** si toute interprétation booléenne de Σ_c satisfait α et α est **satisfaisable** dans l'interprétation $\mathcal{A}$ de Σ_c s'il existe une valuation ρ telle que $\alpha^{\mathcal{A}}(\rho) = 1$. Dans ce cas on dit que ρ est une **solution** de α dans $\mathcal{A}$.

4.2 Syntaxe abstraite d'une spécification

Dans la section précédente nous avons donné la syntaxe concrète d'une spécification. Celle ci décrit les chaînes de caractères qui sont une spécification bien construite. On peut donc voir

la syntaxe concrète comme l'outil qui permet la définition d'une spécification par l'utilisateur. La syntaxe abstraite est un autre type de représentation des spécifications ; son rôle essentiel est de permettre une définition simple de la **sémantique d'une spécification** ou en d'autres termes de son comportement. Dans le même ordre d'idées, la syntaxe concrète permet de décrire naturellement les différents objets construits à partir d'une spécification comme par exemple les formules logiques dont il est question dans l'introduction de cette section.
Des Σ-algèbres seront utilisées pour définir la syntaxe abstraite car une spécification est essentiellement composée de termes qui sont évalués dans des structures classiques (corps des réels, ensemble fini pour les types énumérés, etc.). Dans ce qui suit nous allons décrire la signature de notre Σ-algèbre. Afin d'alléger la lecture, certains opérateurs sont surchargés, ce qui ne porte pas à conséquence puisque ces derniers peuvent être renommés en fonction de leur profil. Supposons que la spécification considérée ait les propriétés suivantes :

- Les types énumérés utilisés sont :

$$\begin{aligned} \mathcal{E}^1 &= \{b_1^1, \ldots, b_{k1}^1\} \\ \mathcal{E}^2 &= \{b_1^2, \ldots, b_{k2}^2\} \\ &\vdots \\ \mathcal{E}^l &= \{b_1^l, \ldots, b_{kl}^l\} \end{aligned}$$

  Remarquons que les types énumérés anonymes (déclaration de la forme : **entrees:** $x, y :< a, b, c >$) ne posent pas problème puisqu'il suffit de leur donner un nom arbitraire. Par la suite on appellera type de **base** tout type qui n'est pas un type contraint.

- Y est l'ensemble des variables d'entrée. Cet ensemble est l'union des ensembles suivants : Y^r pour les variables de type réel et Y^{ei} pour celles de type $\mathcal{E}^i$. De façon analogue pour les variables de sortie on a les ensembles Z, Z^r et Z^{ei}. L'ensemble de toutes les variables de la spécification sera noté $X = Y \cup Z$, X^r et X^{ei} désignerons les sous-ensembles de X de façon similaire à Y et Z.

Dans ces conditions, la signature Σ est définie par $\Sigma = \Sigma_f \cup \Sigma_p \cup \Sigma_b$ où :

Σ_f déclare les opérateurs fonctionnels. L'ensemble des sortes de **base** est défini par :

$$\mathcal{S}_f = \{reel\} \bigcup_{j=1}^{l} \{enum^j\}$$

et l'ensemble des déclarations d'opérateurs par :

$$\mathcal{F}_f = \left\{ \begin{array}{lll} * & : reel \times reel & \rightarrow reel \\ + & : reel \times reel & \rightarrow reel \\ - & : reel \times reel & \rightarrow reel \\ - & : reel & \rightarrow reel \end{array} \right\} \bigcup_{r \in \mathcal{R}} \{r :\rightarrow reel\} \bigcup_{j=1}^{l} \bigcup_{i=1}^{k_j} \{b_i^j :\rightarrow enum^j\}$$

où $\mathcal{R}$ est l'ensemble des nombres réels. Remarquons qu'à chaque nombre réel est associée une déclaration de constante dans la signature. L'ensemble des déclarations n'est donc pas récursivement énumérable comme c'est usuellement le cas. Cette remarque ne porte pas à conséquence car les résultats utilisés par la suite ne font pas appel à cette propriété.

Σ_p déclare les prédicats, dans notre cas l'ensemble des sortes est :

$$\mathcal{S}_p = \{bool, reel\} \bigcup_{j=1}^{l} \{enum^j\}$$

et pour l'ensemble des déclarations d'opérateurs on a:

$$\mathcal{F}_p = \left\{ \begin{array}{lll} = & : reel \times reel & \rightarrow bool \\ \neq & : reel \times reel & \rightarrow bool \\ > & : reel \times reel & \rightarrow bool \\ \geq & : reel \times reel & \rightarrow bool \\ < & : reel \times reel & \rightarrow bool \\ \leq & : reel \times reel & \rightarrow bool \end{array} \right\} \bigcup_{j=1}^{l} \left\{ \begin{array}{lll} = & : enum^j \times enum^j & \rightarrow bool \\ \neq & : enum^j \times enum^j & \rightarrow bool \end{array} \right\}$$

Σ_b déclare les connecteurs booléens classiquement. Pour les sortes on a $\mathcal{S}_b = \{bool\}$ et pour les déclarations d'opérateurs on a:

$$\mathcal{F}_b = \left\{ \begin{array}{lll} \top & : bool \times bool & \rightarrow bool \\ \perp & : bool \times bool & \rightarrow bool \\ \neg & : bool \times bool & \rightarrow bool \\ \vee & : bool \times bool & \rightarrow bool \\ \wedge & : bool \times bool & \rightarrow bool \end{array} \right\}$$

Si on considère les éléments de X^r comme des variables de type **reel** et ceux de X^{e_i} comme des variables de type $enum^i$ alors on peut définir la Σ-algèbre libre $\mathcal{T}(\Sigma, X)$. Dans ces conditions, la syntaxe abstraite d'une spécification à $n+1$ règles est définie par le n-uplet $S = (pre, CT, LR)$ où

- pre est la conjonction des préconditions.
- CT est un ensemble de couples qui décrit les contraintes de types. Le couple (x, t) est élément de CT si et seulement si t est une formule de $\mathcal{T}(\Sigma, X)$ qui code le domaine de x par rapport à son type de base.
- $LR = [r_0, r_1, \ldots, r_n]$ est une liste de couples $r_i = (c_i, \sigma_i)$ où $c_i \in \mathcal{T}(\Sigma, X)$ est la condition de la règle numéro i et ou σ_i et une substitution de $\mathcal{T}(\Sigma, X)$ représentant l'ensemble des assignations de la règle i. De façon plus explicite $(x \mapsto t)$ est dans le graphe de σ_i si et seulement si $x = t$ est une assignation de la règle i. Une règle non conditionnelle a comme condition la constante $\top$.

Remarquons que le support de $\mathcal{T}(\Sigma, X)$ contient des termes illicites dans une spécification puisque par exemple $\mathcal{T}(\Sigma, X)$ contient des termes non linéaires alors que ceux-ci sont interdits dans une spécification. Cette remarque ne porte pas à conséquence car l'important est que tout terme apparaissant dans une spécification soit un terme de $\mathcal{T}(\Sigma, X)$.

Exemple 5 *La relation entre syntaxe concrète et syntaxe abstraite est relativement simple comme le montre cet exemple. Si on considère la spécification suivante*

module: *exemple*		
types:	*positif* : **reel**$[0 \leq @]$	
	intervalle : *positif*$[@ \leq 20]$	
entrees:	x, y : *intervalle*	
sorties:	u, v : *positif*	
preconditions:	$x < y$;	
bases:	$v = 2$;	
exceptions:	**si** $x + 2 * y \leq 10$	**alors** $u = 2$ **et** $v = 1.4 * y$;
	si $x + 2 * y > 10$	**alors** $u = 12$ **et** $v = 1.6 * y$;
validations:	**si** $v > 10$	**alors** $v = 10$;

alors avec la syntaxe abstraite la spécification est décrite par (pre, CT, LR) *où :*

$$\begin{array}{rcl} pre & = & x < y \\ CT & = & \left\{ \begin{array}{lll} (& x, & 0 \leq x \wedge x \leq 20 \;), \\ (& y, & 0 \leq y \wedge y \leq 20 \;), \\ (& u, & 0 \leq u) \;), \\ (& v, & 0 \leq v) \;) \end{array} \right\} \\ LR & = & \left[\begin{array}{lll} (& \top & ,\{v \mapsto 2\} \;), \\ (& x + 2 * y \leq 10 & ,\{u \mapsto 2, \; v \mapsto 1.4 * y\} \;), \\ (& x + 2 * y > 10 & ,\{u \mapsto 12, \; v \mapsto 1.6 * y\} \;), \\ (& v > 10 & ,\{v \mapsto 10\} \;) \end{array} \right] \end{array}$$

4.3 Sémantique d'une spécification

Maintenant que nous avons une syntaxe appropriée, nous allons pouvoir définir le comportement d'une spécification pour une valuation de ses variables d'entrée.
Dans un premier temps, nous allons définir une interprétation de la signature utilisée dans la syntaxe abstraite, puis nous utiliserons cette dernière pour définir la sémantique d'une spécification. Si $\mathcal{R}$ est l'ensemble des nombres réels et si $\mathcal{E}^j$ est l'ensemble des constantes du $j^{ème}$ type énuméré, alors le support de l'interprétation $\mathcal{A}$ de Σ est défini comme suit :

$$reel^{\mathcal{A}} = \mathcal{R} \quad \text{et} \quad (enum^j)^{\mathcal{A}} = \mathcal{E}^j \quad \text{et} \quad bool^{\mathcal{A}} = \mathbf{2}$$

Le support $\mathcal{S}^{\mathcal{A}}$ de la Σ-algèbre étant défini nous pouvons donner l'interprétation des fonctions de $\mathcal{F}$.

$\mathcal{F}_f^{\mathcal{A}}$: Pour toute les constantes r : *reel* on a $r^{\mathcal{A}} = r$ avec $r \in \mathcal{R}$, pour les fonctions binaires $\{+^{\mathcal{A}}, -^{\mathcal{A}}, *^{\mathcal{A}}\}$ et unaire $\{-^{\mathcal{A}}\}$ la sémantique est la même que dans le corps des réels. Pour toute constante b_i^j : $enum^j$ on a $(b_i^j)^{\mathcal{A}} = b_i^j$ avec $b_i^j \in enum^j$.

$\mathcal{F}_p^{\mathcal{A}}$: Si les arguments du prédicat sont de sorte *reel* alors la sémantique de son interprétation est la même que celle du prédicat correspondant sur les réels. Si tous ses argument sont de sorte $enum^j$ alors $=^{\mathcal{A}}$ est l'identité et $\neq^{\mathcal{A}}$ la négation de l'identité.

$\mathcal{F}_b^{\mathcal{A}}$: La sémantique des fonctions de $\mathcal{F}_b^{\mathcal{A}}$ est celle des fonctions booléennes correspondantes.

Soit une spécification $S = (pre, CT, LR)$ et une valuation $\rho : Y \rightarrow \mathcal{S}^{\mathcal{A}}$ appelée **valuation initiales**. Le résultat de l'exécution de S pour ρ, notée $S(\rho)$, est soit le symbole ∇ soit une valuation $\rho' : X \rightarrow \mathcal{S}^{\mathcal{A}}$ appelée **valuation finale**. Intuitivement si le résultat est ∇ alors un échec s'est produit durant l'exécution et si le résultat est une valuation ρ' alors ρ' indique la valeur des variables après l'exécution de la spécification. Plus formellement la sémantique d'une spécification $S = (pre, CT, LR)$ est donnée ci-dessous, les parties droites des égalités se lisent de haut en bas et les indices des symboles ∇ ne sont pas significatifs, ils serviront à décrire la cause

de l'échec :

$$(pre, CT, LR)(\rho) = \begin{cases} \nabla_1 & \text{si } \exists(x,t) \in CT,\ \rho(t) = 0 \text{ sinon} \\ \nabla_2 & \text{si } Var(pre) \not\subseteq Dom(\rho) \text{ sinon} \\ \nabla_3 & \text{si } \rho(pre) = 0 \text{ sinon} \\ (CT, LR)(\rho) & \end{cases}$$

$$(CT, [r_0, \ldots, r_n])(\rho) = \begin{cases} & \text{soit } \rho' = (CT, r_0)(\rho) \\ \nabla & \text{si } \rho' = \nabla \text{ sinon} \\ (CT, [r_1, \ldots, r_n])(\rho') & \end{cases}$$

$$(CT, [])(\rho) = \rho$$

$$(CT, (c, \sigma))(\rho) = \begin{cases} \nabla_4 & \text{si } Var(()c) \not\subseteq Dom(\rho) \text{ sinon} \\ \rho & \text{si } \rho(c) = 0 \text{ sinon} \\ \nabla_5 & \text{si } Range(\sigma) \not\subseteq Dom(()\rho) \text{ sinon soit } \rho' = \rho(\sigma) \\ \nabla_6 & \text{si } \exists(x,t) \in CT,\ \rho'(t) = 0 \text{ sinon} \\ \rho' & \end{cases}$$

Rappelons que les variables de sortie ont une valeur indéterminée au début de l'exécution de la spécification. Dans ces conditions, les causes d'échec sont les suivantes :

∇_1 La valeur d'au moins une variable d'entrée est hors domaine.

∇_2 Au moins une variable de sortie apparaît dans une précondition.

∇_3 Au moins une précondition n'est pas satisfaite.

∇_4 Au moins une variable de la condition d'une règle est indéterminée quand elle est évaluée.

∇_5 Au moins une variable d'un membre droit d'une assignation est indéterminée quand il est évalué.

∇_6 Une variable est assignée hors de son domaine.

A partir de ce point nous ne considérerons que des valuations initiales qui excluent les échecs ∇_1 et que des spécifications S qui excluent les échecs ∇_2, ∇_3, ∇_4, ∇_5 et ∇_6. Nous verrons par la suite comment il est possible de tester si une spécification satisfait cette propriété.

En donnant la sémantique d'une spécification S, nous avons décrit quelle était la valuation finale en fonction d'une valuation initiale. Par la suite une autre information qui nous intéressera sera : quelles sont les règles qui se sont déclenchées lors de l'exécution. Si S est une spécification contenant $n+1$ règles et si $\mathcal{N}$ dénote les entiers naturels alors un **chemin d'exécution** pour S est un mot du monoïde libre $(\{+,-\} \cup \mathcal{N})^*$ de la forme :

$$s_0 0 s_1 1 s_2 2 \ldots s_n n$$

telle que pour tout $i \in [0..n]$ on ait $s_i \in \{+,-\}$. Si $s_i = +$ alors cela signifie que la règle numéro i s'est déclenchée et si ce n'est pas le cas alors $s_i = -$. Si S est une spécification et si ρ est

la valuation initiale alors le chemin d'exécution obtenu peut être vu comme une sémantique particulière de S notée $Ch(S,\rho)$. Dans la définition suivante . dénote la concaténation de mots et ε le mot vide.

$$\begin{array}{lcl}
Ch((pre,CT,LR),\rho) & = & Ch(LR,0,\rho) \\
Ch([(c_i,\sigma_i),r_{i+1},\ldots,r_n],i,\rho) & = & Ch(r_i,i,\rho).Ch([r_{i+1},\ldots,r_n],i+1,\rho') \\
 & & \text{où } \rho' = \rho \text{ si } \rho(c_i) = 0 \text{ et } \rho' = \rho(\sigma) \text{ si } \rho(c_i) = 1 \\
Ch([],i,\rho) & = & \varepsilon \\
Ch((c_i,\sigma_i),i,\rho) & = & \begin{cases} -i & \text{si} \quad \rho(c_i) = 0 \\ +i & \text{si} \quad \rho(c_i) = 1 \end{cases}
\end{array}$$

Remarque 1 *Soient une spécification S et une valuation initiale ρ. L'analyse des deux sémantiques permet de remarquer que si on calcule le chemin $Ch(S,\rho)$ alors dans le dernier appel récursif $Ch([],i,\rho')$ on a $\rho' = S(\rho)$.*

4.4 Formule logique et spécification

Dans cette section nous allons associer une formule logique à toute spécification et ensuite nous montrerons le lien qui existe entre exécution de la spécification et satisfaisabilité de la formule construite. Dans un premier temps nous allons décrire la construction de la formule avec le même formalisme que celui utilisé pour la sémantique.
Si $S = (pre, CT, LR)$ est une spécification et w un chemin d'exécution pour S alors la formule $F(S,w)$ est définie par : .

$$\begin{array}{lcl}
F((pre,CT,LR),w) & = & F(LR,w,id) \\
F([(c,\sigma),r_1,\ldots,r_n],s.k.w,\varphi) & = & \begin{cases} \neg\varphi(c) \wedge F([r_1,\ldots,r_n],w,\varphi) & \text{si} \quad s = - \\ \varphi(c) \wedge F([r_1,\ldots,r_n],w,\varphi(\sigma)) & \text{si} \quad s = + \end{cases} \\
F([],\varepsilon,\varphi) & = & \top
\end{array}$$

où id la substitution identité de $\mathcal{T}(\Sigma,X)$, $s \in \{-,+\}$ et $k \in \mathcal{N}$. Il existe un lien étroit entre la substitution φ qui est perpétué dans $F(S,w)$ et la valuation ρ calculée dans la sémantique de S, nous donnons ci-dessous leurs évolutions respectives afin de mettre en évidence ce lien. Supposons que w soit le chemin d'exécution de S pour la valuation initiale ρ dans ce cas

$$\begin{array}{lcl}
Evol(S,w,\rho) & = & Evol(LR,w,\rho,id) \\
Evol([(c,\sigma),r_1,\ldots,r_n],s.k.w,\rho,\varphi) & = & Evol([r_1,\ldots,r_n],w,\rho',\varphi') \\
 & & \text{où } \rho' = \rho \text{ et } \varphi' = \varphi \quad \text{si} \quad s = - \quad \text{et} \\
 & & \quad \rho' = \rho(\sigma) \text{ et } \varphi' = \varphi(\sigma) \quad \text{si} \quad s = + \\
Evol([],\varepsilon,\rho,\varphi) & = & (\rho,\varphi)
\end{array}$$

ou $k \in \mathcal{N}$. On peut facilement voir que $Evol$ a été obtenu en nettoyant la sémantique de S et la fonctions F.

Lemme 1 *Soient S est une spécification et ρ une valuation initiale. Si $w = Ch(S,\rho)$ alors $Evol(S,w,\rho) = (\rho',\varphi')$ est tel que $\rho' = \rho(\varphi')$ $[Dom(\rho')]$ et $Dom(\varphi') \subseteq Dom(\rho')$*

Preuve: La preuve se fait par récurrence sur le nombre de règles n de la spécification avec la notation $\rho' = \rho_n$ et $\varphi' = \varphi_n$. Si $n = 0$ alors on a $\rho_0 = \rho(\varphi_0) = \rho(id)$ $[Dom(\rho_0)]$ et $Dom(\varphi_0) \subseteq Dom(\rho_0)$. Si $n > 0$ alors avec l'hypothèse $\rho_{n-1} = \rho(\varphi_{n-1})$ $[Dom(\rho_{n-1})]$ et $Dom(\varphi_{n-1}) \subseteq Dom(\rho_{n-1})$ on a deux cas. Si $s = +$ alors $\rho_n = \rho_{n-1}(\sigma)$ et $\varphi_n = \varphi_{n-1}(\sigma)$ et en utilisant l'hypothèse on a donc

$$\rho_n = \rho_{n-1}(\sigma) =_{hyp} \rho(\varphi_{n-1}(\sigma)) = \rho(\varphi_n) \quad [Dom(\rho_{n-1}) \cup Dom(\sigma) = Dom(\rho_n)]$$

et $Dom(\varphi_n) \subseteq Dom(\rho_n)$. Si $s = -$ alors la preuve est triviale puisque $\rho_n = \rho_{n-1}$ et $\varphi_n = \varphi_{n-1}$. □

Le lemme suivant énonce le lien qui existe entre solution de la formule et exécution de la spécification.

Lemme 2 *Soient une spécification* $S = (pre, CT, [(c_0, \sigma_0), \ldots, (c_n, \sigma_n)])$ *et un chemin d'exécution* $w = s_0 0, s_1 1 \ldots s_n n$ *pour* S. ρ *est une solution de* $F(S, w)$ *si et seulement si* $Ch(S, \rho) = w$.

Preuve: Dans la preuve φ_i et ρ_i font référence aux valeurs courantes de φ et ρ lorsque (c_i, σ_i) est traité dans $F(S, w)$ et $Ch(S, \rho)$.

Si ρ est solution de $F(S, w)$ alors pour tout $i \in [0..n]$ on a $\rho(\varphi_i(c_i)) = 1$ si $s_i = +$ et $\rho(\varphi_i(c_i)) = 0$ si $s_i = -$. Si $s_i = +$ alors par le lemme précédent on a $\rho_i = \rho(\varphi_i)$ $[Dom(\rho_i)]$ donc $\rho_i(c_i) = 1$ et $Ch((c_i, \sigma_i), i, \rho) = +$. Si $s_i = -$ alors le raisonnement est le même.

La preuve de la réciproque est similaire: si $Ch(S, \rho) = w$ alors pour tout $i \in [0..n]$ on a $\rho_i(c_i) = 1$ si $s_i = +$ et $\rho_i(c_i) = 0$ si $s_i = -$. Comme on a $\rho_i = \rho\sigma_1\sigma_2 \ldots \sigma_{i-1}$ et $\varphi_i = \sigma_1\sigma_2 \ldots \sigma_{i-1}$ on en déduit $\rho(\varphi_i(c_i)) = 1$ si $s_i = +$ et $\rho(\varphi_i(\neg c_i)) = 1$ si $s_i = -$. La formule étant une conjonction de termes de cette forme on en déduit que ρ est solution de $F(S, w)$ □

Le lemme suivant permet d'obtenir plus d'information sur les exécutions d'une spécification S.

Lemme 3 *Soient* S *une spécification et* w *un chemin d'exécution pour* S. *Si* ρ *est solution de* $F(S, w)$ *alors* $S(\rho) = \rho(\varphi')$ *où* φ' *est la substitution courante lors de la dernière étape de la construction de* $F(S, w)$ *c'est à dire* $F([], \varepsilon, \varphi')$

Preuve: C'est une conséquence directe du lemme 1 et de la remarque 1. □

Remarque 2 *Ce lemme est important car il permet de déterminer à partir d'une solution* ρ *de* $F(S, w)$ *la valuation finale pour la valuation initiale* ρ. *En fait c'est la valeur des variables à n'importe quel moment de l'exécution qui peut être déterminée, en effet si on s'intéresse par exemple à la valeur des variables après le déclenchement de la règle numéro i il suffit de considérer la spécification tronquée à partir de la règle i.*
Il est possible de compléter $F(S, w)$ *de façon à ce que la valeur d'une variable d'intérêt soit directement retournée dans la solution de la formule. Si on s'intéresse par exemple à la valeur de* x *après le déclenchement de la règle i alors il suffit de considérer la formule* $F(S, w) \wedge x' = \varphi_i(x)$ *ou* x' *est une nouvelle variable.*
Supposons que l'on ait ajouté à $F(S, w)$ *un ensemble de variables d'intérêts* X' *et que la formule obtenue soit* $F'(S, w)$, *si on ajoute à* $F'(S, w)$ *une formule logique quelconque* $C(X')$ *à variables dans* X' *alors* $F(S, w)' \wedge C(X')$ *aura une solution si et seulement si il existe une valuation initiale* ρ *telle* $Ch(S, \rho) = w$ *et* $C(X')$ *soit vérifié durant l'exécution. Il est ainsi possible de vérifier que la valeur finale d'une variable ne dépasse pas une certaine borne, etc.*

4.5 Vérification

Dans ce qui suit nous allons mettre à profit les résultats obtenu plus haut pour effectuer les deux tests qui nous intéressent.

4.5.1 Test d'ambiguïté

Si $S = (pre, CT, LR)$ est une spécification alors l'ambiguïté de la variable de sortie x se teste comme suit :

1. Soit R l'ensemble des règles d'exception qui assignent la variable x dans leur conclusion. On construit l'ensemble W de tous les chemins d'exécution qui codent le déclenchement d'au moins deux règles de R.

2. On construit la formule
$$F = \bigvee_{w \in W} (F(S, w) \land pre \land CT)$$
Notons qu'en ajoutant les contraintes de type de cette manière, ces dernières ne sont prises en compte que pour la valuation initiale.

3. On teste la satisfaisabilité de F et par le lemme 3 on a: il y a ambiguïté sur x si et seulement si F est satisfaisable. En d'autres termes cela signifie qu'il existe au moins un chemin $w \in W$ tel que $F(S, w)$ soit satisfaisable donc il existe une valuation initiale ρ telle qu'au moins deux règles de R se déclenchent.

Le test de non ambiguïté consiste a répéter ce processus pour toutes les variables de sortie.

4.6 Test de non définition

Si S est une spécification alors la non définition de la variable x se teste comme suit :

1. Soit R l'ensemble des règles qui assignent la variable x dans leur conclusion. Si R contient une règle non conditionnelle alors x est défini. Si ce n'est pas le cas on construit l'ensemble W de tous les chemins d'exécution qui codent le non déclenchement de toutes les règles de R.

2. On construit la formule
$$F = \bigvee_{w \in W} (F(S, w) \land pre \land CT)$$
Notons qu'en ajoutant les contraintes de type de cette manière, ces dernières ne sont prises en compte que pour la valuation initiale.

3. On teste la satisfaisabilité de F et par le lemme 3 on a: x est non défini si et seulement si F est satisfaisable. En d'autres termes cela signifie qu'il existe au moins un chemin $w \in W$ tel que $F(S, w)$ soit satisfaisable donc il existe une valuation initiale ρ telle qu'aucune règle de R ne se déclenche.

Le test de non définition consiste a répéter ce processus pour toutes les variables de sortie.

4.7 Table de dépendances

On peut remarquer que pour chacun des deux tests, le but est de déterminer si un ensemble de règles se déclenchent où ne se déclenchent pas pour toute valuation initiale.
Considérons dans un premier temps le cas d'une seule règle. Il est clair que le déclenchement d'une règle donnée ne dépend pas des règles qui la suivent dans la spécification. D'autre par il est possible que le déclenchement de cette règle soit indépendant de certaines règles qui la précède dans la spécification. Afin de considérer les formules logiques les plus courtes possible, des tables de dépendances sont construites.
Soit une spécification contenant n règles $[(c_1, \sigma_1), r_2, \ldots, r_n]$ et k variables $[x_1, x_2, \ldots, x_k]$. Deux tables de dépendances sont construites simultanément :

- La table de dépendances de valeurs TDV est un tableau $(k \times n)$ de booléens tel que $TDV(i,j) = 1$ si la valeur de la variable x_i dépend de la règle r_j et $TDV(i,j) = 0$ sinon.
- La table de dépendances de déclenchements TDD est un tableau $(n \times n)$ de booléens tel que $TDD(i,j) = 1$ si le déclenchement de la règle r_i dépend de la règle r_j et $TDD(i,j) = 0$ sinon.

Ces tableaux initialisés à 0 sont construits simultanément. La notation à un argument pour les tableaux référence les lignes et la notation à zéro argument référence tout le tableau.

TDV = 0
TDD = 0
Pour $i = 1$ a n **faire**
  TDV' =TDV
  Pour $x_j \in Dom(\sigma_i)$ **faire**
    $$TDV(j) = TDV'(j) \bigvee_{x_l \in Var(c_i) \cup Var(\sigma_i(x_j))} TDV'(l)$$
    $TDV(j,i) = 1$
  fin
  $$TDD(j) = TDD'(j) \bigvee_{x_l \in Var(c_i)} TDV'(l)$$
fin

Ou en d'autres termes,

- Si la variable x_j est assignée par t dans la règle i alors la dépendance de valeur de v_j après la règle i est égale à la dépendance de valeur de v_j avant la règle i plus la règle i plus les dépendances de valeurs des variables qui apparaissent soit dans t soit dans la condition de la règle i. Si la variable x_j n'est pas assignée pas la règle i alors sa dépendance de valeur est inchangée.
- La dépendance de déclenchement de la règle i est égale à l'ensemble des dépendances de valeurs de toutes les variables qui apparaissent dans la condition de la règle i.

Ces tables sont abondamment utilisées dans tout le programme afin de minimiser le travail effectué. Remarquons finalement que la dépendance de déclenchement d'un ensemble de règles est égale à l'union des dépendances de déclenchement des règles de l'ensemble.

4.8 Test de satisfaisabilité des contraintes réelles linéaires.

L'algorithme implanté teste si une conjonction d'équations $LX = c$, d'inéquations larges $LX \leq c$, d'inéquations strictes $LX < c$ et de diséquations $LX \neq c$ admet une solution. Le problème est donné sous la forme canonique d'une matrice A $n \times m$ et d'un vecteur colonne C à m composantes. La matrice est divisée logiquement en quatre parties contenant respectivement les coefficients des équations, des inéquations larges, des inéquations strictes et des diséquations.

$$A \left\{ \begin{array}{|c|} \hline A_1 \\ \hline A_2 \\ \hline A_3 \\ \hline A_4 \\ \hline \end{array} \begin{array}{c} = \\ \leq \\ < \\ \neq \end{array} \begin{array}{|c|} \hline C_1 \\ \hline C_2 \\ \hline C_3 \\ \hline C_4 \\ \hline \end{array} \right\} C$$

qu'on notera sous la forme

$$\left[\begin{array}{c|c} A_1 & C_1 \\ A_2 & C_2 \\ A_3 & C_3 \\ A_4 & C_4 \end{array} \right]$$

Le test consiste essentiellement à déterminer l'enveloppe affine de l'ensemble des solutions des contraintes positives puis à tester que cette enveloppe n'est pas contenue dans un des hyperplans exclus par les contraintes négatives.

4.8.1 Réduction du problème par les égalités

On commence par réduire toutes les contraintes par les égalité au moyen d'un pivot sur A_1. Pour chaque ligne de A_1, on choisit un pivot non nul puis on annule tous les coefficients en dessous du pivot dans A. Le résultat est de la forme

$$\left[\begin{array}{c|c} A'_1 & C'_1 \\ A'_2 & C'_2 \\ A'_3 & C'_3 \\ A'_4 & C'_4 \end{array} \right]$$

A ce stade, soit A'_1 contient une ligne nulle avec un coefficient non nul sur la ligne correspondante de C'_1, auquel cas le problème n'admet pas de solution, soit on peut oublier $[\ A'_1 \mid C'_1\ ]$. En effet, à partir de toute solution de

$$\left[\begin{array}{c|c} A'_2 & C'_2 \\ A'_3 & C'_3 \\ A'_4 & C'_4 \end{array} \right]$$

on pourra calculer une solution du problème initial.

4.8.2 Recherche d'une solution aux inéquations

Dans cette phase de l'algorithme, on considère toutes les inéquations comme des inéquations larges. On se ramène à une conjonction d'équations à satisfaire sur les réels positifs. Pour cela, on introduit des variables d'écart, c'est à dire que l'inéquation $LX \leq c$ est transformée en $LX + y = c,\ y \geq 0$.

En pratique, cela revient à considérer la matrice $[A'_{23}\ I \mid C'_{23}]$ où A'_{23} (resp. C'_{23}) est la matrice formée des lignes de A'_2 et A'_3 (resp. C'_2 et C'_3). Par un pivot sur les lignes de A'_{23}, on réduit cette matrice à

$$\left[\begin{array}{cc|c} B & R & C'' \\ 0 & S & D \end{array} \right]$$

où le rang de B est égal à son nombre de ligne et D est un vecteur colonne positif. Le problème est maintenant de déterminer si le problème

$$P : \begin{cases} BX + RY = C'' \\ SY = D \\ Y \geq 0 \end{cases}$$

admet une solution, ce qui revient à déterminer si

$$Q : \begin{cases} SY = D \\ Y \geq 0 \end{cases}$$

admet une solution. En effet, à partir de toute solution de Q, on pourra calculer une solution de P.
Une solution de Q sera obtenue par l'algorithme du simplexe appliqué au problème d'optimisation

$$\begin{array}{l} \min \sum \lambda_i \\ SY + \Lambda = D \\ Y, \Lambda \geq 0 \end{array}$$

où les λ_i sont les variables *artificielles* et Λ est le vecteur des λ_i. On a immédiatement la solution de base admissible $(Y, \Lambda) = (0, D)$ car D est positif. Le tableau initial du simplexe est donc

$\sum_i D_i$	$\sum_i S_i^1 \quad \cdots \quad \sum_i S_i^n$
D	$S^1 \quad \cdots \quad S^n$

où S^j désigne la $j^{\text{ème}}$ colonne de S.
Le tableau retourné par l'algorithme du simplexe est de la forme

α	?
D'	T

Si α est non nul, le problème Q (et donc P) n'admet pas de solution. Sinon après élimination des variables artificielles, le tableau est de la forme

	0	0 (Y_N)
Y_B	D'_B	T'

où Y_B est constitué des variables de base et Y_N des variables hors base. $(Y_B, Y_N) = (D'_B, 0)$ est alors une solution de Q.

4.8.3 Recherche des équations implicites

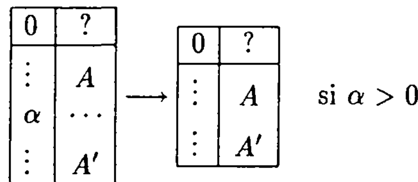
Le problème consiste maintenant à déterminer parmi les inéquations de P lesquelles sont en fait des équations. Ces inéquations sont appelées *équations implicites* et définissent l'enveloppe affine de l'ensemble des solutions de P.

Si le problème P est de la forme $P' \wedge LX \leq c$, $LX \leq c$ est une équation implicite de P si et seulement si $P \Longrightarrow LX = c$, ou encore $P' \Longrightarrow LX \geq c$.
Déterminer si $Lx \leq c$ est une équation implicite revient à déterminer si $\max\{y \mid SY = D,\ Y \geq 0\} = 0$ où y est la variable d'écart introduite dans l'inéquation $LX \leq c$. On ne va donc travailler que sur les variables d'écart.
Lorsqu'on a déterminé qu'une inéquation implicite est une équation implicite, on peut remplacer la variable d'écart correspondante par 0 dans $SY = D$ (ce qui revient à supprimer une colonne de S) et *oublier* l'inéquation. De même, lorsqu'on a déterminé qu'une inéquation n'est pas une équation implicite, on peut l'*oublier* grâce au théorème suivant.

Théorème 1 *Si $L_1X \leq c_1$ est une équation implicite de $P = (AX \leq B \wedge L_1X \leq c_1 \wedge L_2X \leq c_2)$ et $L_2X \leq c_2$ n'est pas une équation implicite de P, alors $L_1X \leq c_1$ est une équation implicite de $AX \leq B \wedge L_1X \leq c_1$.*

En partant de la solution de P trouvée précédemment, on peut pour certaines inéquations déterminer immédiatement si elle sont ou non des équations implicites.

- Si une composante de la solution de P est non nulle, la variable d'écart correspondante n'est donc pas nulle et l'inéquation associée n'est pas une équation implicite. On peut supprimer la ligne correspondante dans le tableau du simplexe.



  En itérant ce processus, on finit avec un tableau de la forme

0	?
0	?

- Si le tableau contient une ligne positive, c'est à dire est de la forme

		$y_{i_1} \cdots y_{i_N}$
	0	
	$\vdots$	
y_i	0	L
	$\vdots$	

  $L \geq 0$

  en posant $L = (L_1, \ldots, L_N)$ avec $L_j \geq 0$, l'équation correspondant à cette ligne est

$$y_i + L_1 y_{i_1} + \cdots L_N y_{i_N} = 0$$

  Comme les y_k sont positifs, y_i est nul ainsi que chaque y_{i_j} pour lequel $L_j > 0$. Les inéquations correspondantes sont des équations implicites. Les colonnes correspondantes ainsi que la ligne de y_i peuvent alors être supprimées du tableau.

- Si la colonne j du tableau est négative Le système peut alors être écrit sous la forme

$$Y_B + A_1 Y_{N_1} + C y_{i_j} + A_2 Y_{N_2} = 0 \quad \text{avec } C \leq 0$$

et admet en particulier la famille de solutions positives $(Y_B, Y_{N_1}, y_{i_j}, Y_{N_2}) = \lambda(-C, 0, 1, 0)$, $(\lambda \geq 0)$ où y_{i_j} n'est pas borné. L'inéquation correspondante n'est donc pas une équation implicite. Pour supprimer la variable y_{i_j} (c'est à dire l'inéquation correspondante), il faut encore la ramener dans la base en pivotant (on peut prendre un pivot négatif) ou que la colonne soit entièrement nulle (éventuellement vide au cas où il ne reste aucune ligne dans le tableau).

Preuve du théorème 1

La preuve du théorème fait appel au lemme suivant

Lemme 4 *Si $AX \leq B$ admet une solution et $AX \leq B \Longrightarrow CX \leq D$, alors il existe $\lambda \geq 0$ tel que $\lambda A = C$ et $\lambda B \leq D$.*

On prouve le théorème par l'absurde en supposant que $L_1X \leq c_1$ n'est pas une équation implicite de $AX \leq B \wedge L_1X \leq c_1$. Les hypothèses sont alors:

(H0) $AX \leq B \wedge L_2X \leq c_2 \Longrightarrow L_1X \geq c_1$

(H1) $AX_1 \leq B \wedge L_1X_1 \leq c_1 \wedge L_2X_1 < c_2$

(H2) $AX_2 \leq B \wedge L_1X_2 < c_1$

De l'hypothèse (H1), on déduit que $AX \leq B \wedge L_2X \leq c_2$ admet une solution, ce qui nous permet d'utiliser le lemme.
De l'hypothèse (H0) et du lemme, on déduit

$$\lambda A + \lambda_1 L_2 = -L_1 \quad \text{et} \quad \lambda B + \lambda_1 c_2 \leq -c_1 \quad \text{avec } \lambda, \lambda_1 \geq 0 \tag{1}$$

On a alors

$$\begin{array}{rcll} -c_1 & \leq & -L_1X_1 & \text{par (H1)} \\ & = & \lambda AX_1 + \lambda_1 L_2 X_1 & \text{par (1)} \\ & \leq & \lambda B + \lambda_1 L_2 X_1 & \text{par (H1) car } \lambda \geq 0 \\ & \leq & \lambda B + \lambda_1 c_2 & \text{par (H1) car } \lambda_1 \geq 0 \\ & \leq & -c_1 & \text{par (1)} \end{array}$$

Les inégalités sont donc des égalités dont on déduit $\lambda_1 L_2 X_1 = \lambda_1 c_2$ c'est à dire $\lambda_1 = 0$ ou $L_2X_1 = c_2$. Comme $L_2X_1 = c_2$ contredit (H1), on a $\lambda_1 = 0$ et donc

$$\lambda A = -L_1 \quad \text{et} \quad \lambda B \leq -c_1 \tag{2}$$

On obtient alors

$$\begin{array}{rcll} -c_1 & < & -L_1X_2 & \text{par (H2)} \\ & = & \lambda AX_2 & \text{par (2)} \\ & \leq & \lambda B & \text{par (H2) car } \lambda \geq 0 \\ & \leq & -c_1 & \text{par (2)} \end{array}$$

Ce qui est impossible. □

5 Exemples d'exécution

Dans cette section nous introduisons les informations nécessaires pour la maintenance et l'amélioration du programme. Pour ce faire, nous allons dérouler complètement sur un exemple les deux tests qui sont implantés. Ces exemples serviront de support pour la description de l'implantation du programme qui sera présentée dans la section suivante.
Afin d'être le plus proche possible de la réalité l'exemple utilisé ci-après a effectivement été traité par le programme. La plupart des tables et des formules qui sont données par la suite sont à une remise en forme près celles qui sont affichées par le programme lorsque les différentes options sont utilisées au lancement du programme. Il est fortement conseillé au lecteur de cette section d'essayer les exemples, de les modifier et d'afficher les différentes informations. La spécification de l'option -h au lancement du programme provoque l'affichage des options disponibles :

```
Utilisation: versem -[123vst]

-1 : affichage resultat test court
-2 : affichage resultat test normal
-3 : affichage resultat test long
-v : verifie la specification
-s : affiche structure interne specification
-t : affiche table interne specification
-h : affiche l'aide


La specification est lue
sur l'entree standard.
```

5.1 Exemple pour le test de variable ambiguë

Dans cette partie nous allons dérouler sur un exemple le test de non ambiguïté des variables. Avant de commencer rappelons qu'une variable de sortie v est ambiguë s'il existe des valeurs pour les variables d'entrée telles que la variable v soit assignée par au moins deux règles d'exception lors de l'exécution de la spécification.
Rappelons qu'un chemin d'exécution est un mot $s_1k_1, k_2k_2, \ldots, s_nk_n$ ou $s_i \in \{+,-\}$ et k_i est un numéro de règle. Un chemin d'exécution code le comportement de la spécification, par exemple +1−2+3 signifie les règles 1 et 3 se sont déclenchées et la règle 2 ne s'est pas déclenchée.
Considérons la spécification de la figure 4 où les numéros de règles sont en commentaire. Comme l'ambiguïté d'une variable de sortie correspond a l'assignation de cette dernière par deux règles distinctes, nous allons tester pour chaque variable s'il existe une exécution qui satisfait cette condition.

```
/* Petit exemple de demonstration */

module: exemple
types:            positif = reel [0 <= @]
entrees:          x, y : positif
                  z    :< a, b, c >
sorties:          v, w : positif
preconditions:    x < y + 2;
bases:
/* 0 */           v = 2;
exceptions:
/* 1 */           si x < 10            alors z = a et y = x - 1 et w = x;
/* 2 */           si y > 10 et z != c  alors w = x + v;
/* 3 */           si z = c             alors w = y;
```

FIG. 4 - *Spécification exemple*

Avant de détailler le déroulement du test pour cet exemple nous allons donner les points essentiels de ce dernier en donnant pour chaque point les étapes correspondantes.

- Construction de la structure interne (étapes 1 à 2).
- Choix d'une variable de sortie par exemple w (étape 4).
- Choix d'une paire de règles qui assignent la variable choisie par exemple la paire $(2, 3)$ pour w (étape 6).
- Choix d'une chemin d'exécution tel que les deux règles choisies soient déclenchées, par exemple +0+1+2+3 pour $(2, 3)$. Nous verrons plus bas qu'il n'est pas nécessaire de considérer la règle 0. (étape 8).
- Tester s'il existe une exécution qui satisfait ce chemin, si c'est le cas pour +0+1+2+3 alors il y a ambiguïté sur la variable w par les règles $(2, 3)$ sur le chemin +0+1+2+3. (étapes 9 à 10).

ETAPE 1

L'analyse lexicale et syntaxique de la spécification est effectuée. Durant cette phase la table des types, la table des symboles et la structure interne de la spécification sont construites. Pour cet exemple, la structure de la table des types est (option -s au lancement) :

```
TABLE DES TYPES:

Type: 2              : < enumere > contenant:
                         nom: c no: 2 element type: <enumere>
                         nom: b no: 1 element type: <enumere>
                         nom: a no: 0 element type: <enumere>

Type: -1 positif     : <contraint>
                         base_real <reel> 0 <= @
```

Cette table se lit comme suit : le type numéro 2 est anonyme (pas de nom après le numéro), c'est un type énuméré qui contient les symbole c, b et a. Le type numéro -1 a pour nom "positif", c'est un type contraint sur les réels tel que p est un élément de "positif" si $0 \leq p$. Notons au passage que les types contraints ont toujours des numéros négatifs et que la numérotation des autres types commence à 2 car les numéros 0 et 1 sont réservés aux types de base booléen et réel. La table des symboles a quant à elle la structure suivante (option -s au lancement) :

```
TABLE DES SYMBOLES:
nom: w    no: 1    var sortie    type: positif    <contraint>
nom: v    no: 0    var sortie    type: positif    <contraint>
nom: z    no: 4    var entree    type:            < enumere >
nom: c    no: 2    element       type:            < enumere >
nom: b    no: 1    element       type:            < enumere >
nom: a    no: 0    element       type:            < enumere >
nom: y    no: 3    var entree    type: positif    <contraint>
nom: x    no: 2    var entree    type: positif    <contraint>
```

qui se lit : le symbole w a le numéro 1, c'est une variable de sortie du type "positif" qui est un type contraint, etc. Les symboles sont numérotés de 0 à n pour chaque type, en réalité toute l'information de type est accessible depuis le descripteur de symbole puisque ce dernier contient un accès au descripteur de type.
De façon simplifiée, les préconditions sont stockées dans une table de formules logiques et les règles dans une table de couples (formule logique, substitution), la formule représente la condition et la substitution la liste d'affectations. Ces tables peuvent également être affichées avec l'option -s.

ETAPE 2

Après l'initialisation de la structure interne, cette dernière est utilisée pour construire les différentes tables de dépendances. Comme nous le verrons à l'étape 8 ces tables servent essentiellement à déterminer les règles qui doivent être considérées pour un test donné. Les tables de dépendances construites peuvent être affichées avec l'option -t au lancement. Pour notre exemple la première table a la forme :

TABLE D'ASSIGNATION DES VARIABLES

<table>
<tr><th>No règle :</th><th>0</th><th>1</th><th>2</th><th>3</th></tr>
<tr><td>v</td><td>|</td><td>.</td><td>.</td><td>.</td></tr>
<tr><td>w</td><td>.</td><td>|</td><td>|</td><td>|</td></tr>
<tr><td>x</td><td>.</td><td>.</td><td>.</td><td>.</td></tr>
<tr><td>y</td><td>.</td><td>|</td><td>.</td><td>.</td></tr>
<tr><td>z</td><td>.</td><td>|</td><td>.</td><td>.</td></tr>
</table>

Cette table indique les dépendances d'assignations, c'est à dire quelle règle assigne quelle variable. Si x est un symbole de variable et si n est un numéro de règle alors la présence de | à l'intersection de la ligne et de la colonne correspondantes signifie : x est assigné par la règle n. La présence d'un point signifie au contraire que la règle n n'assigne pas la variable x. La seconde table décrit

les dépendances de valeurs c'est à dire de quelle règle dépend la valeur d'une variable :

TABLE DEPENDANCES VALEURS

No règle :	0	1	2	3
v	|	.	.	.
w	|	|	|	|
x	.	.	.	.
y	.	|	.	.
z	.	|	.	.

qui se lit par exemple pour la variable w : la valeur finale de la variable w dépend des règles 0, 1, 2 et 3. On remarque que par rapport à la table précédente w dépend en plus de la règle 0. C'est effectivement le cas car la variable v qui est assigné par la règle 0 intervient dans le calcul de la valeur assignée à w dans la conclusion de la règle numéro 2. Toute dépendance d'assignation sera forcément une dépendance de valeur car la valeur d'une variable dépend forcément des règles où elle est assignée. La dernière table qui est la plus importante indique les dépendances de déclenchements :

TABLE DEPENDANCES DECLENCHEMENTS

No règle :	0	1	2	3
0	.	.	.	.
1	.	.	.	.
2	.	|	.	.
3	.	|	.	.

Cette table se lit par exemple pour la règle numéro 2 : le déclenchement de la règle numéro 2 dépend de la règle numéro 1. La raison de cette dépendance est la présence dans la condition de la règle numéro 2 de la variable y qui est assignée dans la conclusion de la règle numéro 1.

ETAPE 3

A ce stade, toute la structure interne est construite et le test lui même va pouvoir débuter. Dans cette étape O est initialisé avec l'ensemble des variables de sortie de la spécification donc $O = \{v, w\}$.

ETAPE 4

Si O est non vide à cette étape alors une variable est choisie et retirée de O, si O est vide alors le test est terminé. Pour le déroulement de notre exemple, supposons que cela soit le premier passage dans cette étape et que la variable choisie soit w. Dans ces conditions on a $O = O \backslash \{w\} = \{v\}$.

ETAPE 5

L'ensemble P est initialisé avec toutes les paires de règles qui assignent la variable choisie. Pour notre exemple cela donne $P = \{(1,2),(1,3),(2,3)\}$.

ETAPE 6

Si P est vide alors on saute à l'étape 4, si ce n'est pas le cas une paire est choisie et retirée de P. Supposons pour le déroulement de notre exemple que cela soit le premier passage dans cette étape et que la paire choisie soit $(2,3)$. Dans ces conditions nous avons $P = P \backslash \{(2,3)\} = \{(1,2),(1,3)\}$.

ETAPE 7

L'ensemble C est initialisé avec tous les chemins d'exécution qui codent le déclenchement de la paire de règles choisie. Pour cette opération on ne considère que les règles dont dépendent les deux règles choisie. En effet si le déclenchement des deux règles choisie est indépendant d'une troisième règle alors cette règle n'est d'aucun intérêt pour le test courant. Pour le déroulement de notre exemple supposons que cela soit le premier passage dans cette étape et que le couple de règles choisi soit $(2,3)$. Dans ces conditions en utilisant la table de dépendances de déclenchements on obtient $P = \{+1+2+3, -1+2+3\}$.

ETAPE 8

Si C est vide alors on saute à l'étape 6, si ce n'est pas le cas un chemin est choisi et retiré de C. Supposons pour le déroulement de notre exemple que cela soit le premier passage dans cette étape et que le chemin choisi soit +1+2+3. Dans ces conditions nous avons $C = C\backslash\{+1+2+3\} = \{-1+2+3\}$.

ETAPE 9

Une formule logique est construite comme nous l'avons vu dans la section précédente. Si on suppose que le chemin choisi est +1+2+3 alors la formule obtenue est donnée ci-dessous. Cette formule a la propriété suivante : elle a une solution si et seulement si il existe une valuation initiale qui conduit au déclenchement des règles codées dans le chemin choisi. L'option -3 permet l'affichage des chemins choisis et des formules construites (afin de faciliter la lecture, les règles non conditionnelles sont indiquées par un point à la place de + avec cette option). L'affichage avec cette option est le suivant :

```
test non ambiguite var w par regle 2 3
cas            : +1+2+3
formule        : &(x - 10 < 0,  x - 11 > 0,  a! = c,  a = c, x - y + 2 < 0,  -x <= 0,  -y <= 0)
```

où la formule est notée sous forme aplatie et ou & est le "et" logique. On reconnaît successivement dans cette formule

- la condition de la règle 1 : $x - 10 < 0$,
- une partie de la condition de la règle 2 où y a été remplacé par son assignation dans la règle 1 ($y = x - 1$) : $x - 11 > 0$,
- une partie de la condition de la règle 2 où z a été remplacé par son assignation dans la règle 1 ($z = a$) : $a! = c$,
- la condition de la règle 3 où z a été remplacé par son assignation dans la règle 1 ($z = a$) : $a = c$,
- la précondition : $x - y + 2 < 0$,
- les contraintes de type : $-x <= 0$ et $-y <= 0$.

ETAPE 10

Dans cette étape le test de satisfaisabilité de la formule construite est effectué. Pour notre exemple la formule est clairement insatisfaisable à cause de $a = c$. Comme la formule n'a pas de solution il n'existe pas de valuation initiale qui conduit au chemin choisi donc il n'y a pas ambiguïté sur la variable w pour le chemin +1+2+3. Ceci est signalé pas le message "`pas d'ambiguite`" à la suite de l'affichage précédent si l'option -3 est spécifiée. Après cette étape on continue à l'étape 8.

5.2 Exemple pour le test de variable indéfinie

Le principe de ce test est essentiellement le même que celui de variable ambiguë. Avant de commencer rappelons qu'une variable de sortie v est indéfinie s'il existe des valeurs pour les variables d'entrée telles que la variable v est indéfinie après l'exécution de la spécification. Nous n'allons pas complètement développer ce test qui est très proche du précédent mais nous donnerons cependant les étapes principales et les variations par rapport au test précédent.

- Construction de la structure interne
- Choix d'une variable de sortie par exemple w
- Détermination de l'ensemble R de toute les règles qui assignent w, pour notre exemple nous avons $R = \{1, 2, 3\}$.
- Choix d'une chemin d'exécution tel qu'aucune des règles de R ne soit déclenchée par exemple +0−1−2−3. Comme dans le test précédent la règle 0 n'est pas nécessaire donc l'unique chemin à considérer est −1−2−3.
- Construction de la formule correspondante et test de la satisfaisabilité de cette dernière. Si la formule est satisfaisable alors il existe des valeurs initiales qui conduisent au non déclenchement des règles 1, 2 et 3 donc w est une variable indéfinie. Si ce n'est pas le cas alors pour toute exécution de la spécification, w a une valeur finale qui est définie.

La formule qui est construite et le verdict dans notre cas est donné ci-dessous (option -3 au lancement du programme).

```
teste si w defini en sortie
cas        : −1−2−3
formule    : &(!(x − 10 < 0), !&(y − 10 > 0, z! = c), !(z = c), x − y + 2 < 0,
               −x <= 0, −y <= 0)
variable definie
```

Dans cette formule on reconnaît :

- la négation de la condition de la règle 1 : $!(x - 10 < 0)$,
- la négation de la condition de la règle 2 : $!\&(y - 10 > 0, z! = c)$,
- la négation de la condition de la règle 3 : $!(z = c)$,
- la précondition : $x - y + 2 < 0$,
- les contraintes de type : $-x <= 0$ et $-y <= 0$.

6 Description de l'implantation

Nous décrivons maintenant la structure générale du programme. Les fichiers de définitions étant abondamment commentés il est fortement conseillé au lecteur d'avoir les fichiers de définitions (extension ".h") à disposition lors de la lecture de cette section. D'autre part la lecture de la section précédente pourra aider à comprendre comment les différentes parties du programme coopèrent entre elles.
La structure du programme est donnée dans la figure 5. Dans cette figure une flèche $a \rightarrow b$ signifie : on a besoin de b pour pouvoir déclarer a. Les noms qui apparaissent dans cette figure sont en général des noms de classe, les fichiers qui déclarent et définissent ces classes ont le même nom avec les extensions ".h" et ".cc". Cette règle est toujours vérifiée sauf pour Type et Symb qui sont regroupés dans un même fichier appelé SymbType et pour SymbTable et TypeTable qui sont regroupés dans SymbTypeTable. Toutes ces conventions et plusieurs autres relatives à la syntaxe et à la gestion d'erreurs sont abondamment commentées dans les fichiers sources. Si on revient à notre figure la structure se décrit comme suit :

- **Type** implante les types utilisés. On peut s'étonner de voire une flèche allant sur Formula depuis cette classe, la raison en est la suivante : pour pouvoir décrire les types contraints une référence à une formule logique est nécessaire pour coder la contrainte.

- **Symb** implante les différents symboles : variables et éléments des types énumérés. La raison de la flèche entre les types et les symboles est la suivante : un type énuméré contient des éléments qui sont des symboles.

- **TypeTable** implante la table des types. Cet objet permet essentiellement de déclarer de nouveaux types et d'avoir un accès aux types existants.

- **SymbTable** implante la table des symboles. Son rôle est essentiellement le même que la table des types mais pour les symboles.

- **Term** implante les termes linéaires sur les réels et les termes de types énumérés. Les méthodes de ces classes implantent les différentes opérations nécessaires pour les termes (addition, soustraction, assignation, etc.).

- **Pred** implante les différents prédicats utilisés sur les termes ($=$, $\leq$, $<$, etc.) ainsi que les opérations qui s'y rapportent (négation, évaluation, etc.).

- **Formula** implante les formules logiques et les opérations qui s'y rapportent (et, ou, non, descente des négations, mise sous forme disjonctive, etc).

- **Subst** implante les substitutions et les opérations qui s'y rapportent (application, composition, domaine, etc.).

- **Rule** implante les règles qui sont représentées par une formule logique pour leur condition et par une substitution pour leur conclusion. Les méthodes de cette classe servent essentiellement à la création et à l'accès des différentes composantes d'une règle.

- **Const** est un module similaire à Rule. Son but est de pouvoir représenter différents types de contraintes de façon standard pour des extensions futures. Pour le moment seules les préconditions sont stockées dans les objets de cette classe.

- **ConstrTable** et **RuleTable** sont des classes qui servent essentiellement à regrouper et classer les règles et les contraintes.

- **parser** n'est pas une classe, il s'agit de l'ensemble de fonctions dont le but est d'effectuer l'analyse lexicale et syntaxique de la spécification. Ces fonctions ont été engendrées automatiquement par les générateurs de programmes flex et bison (programme analogue à lex et yacc distribué par la Free Software Fondation) à partir des fichiers de descriptions parser.lex pour la passe lexicale et parser.y pour la passe syntaxique.

- **solver** est un ensemble de fonctions regroupées dans solver.cc. Le but de ces fonctions est de tester la satisfaisabilité d'une formule logique à l'aide des résolveurs spécifiques à chaque type de contraintes (enumcheck pour les contraintes de types énumérés, LpPresolve pour les contraintes d'intervalles sur les réels et lpcheck pour les contraintes linéaires générales sur les réels).

- **Specif** est la classe la plus importante, elle implante les spécifications. Parmi les méthodes de cette classe on a entre autre la lecture d'une spécification et les deux tests implantés.

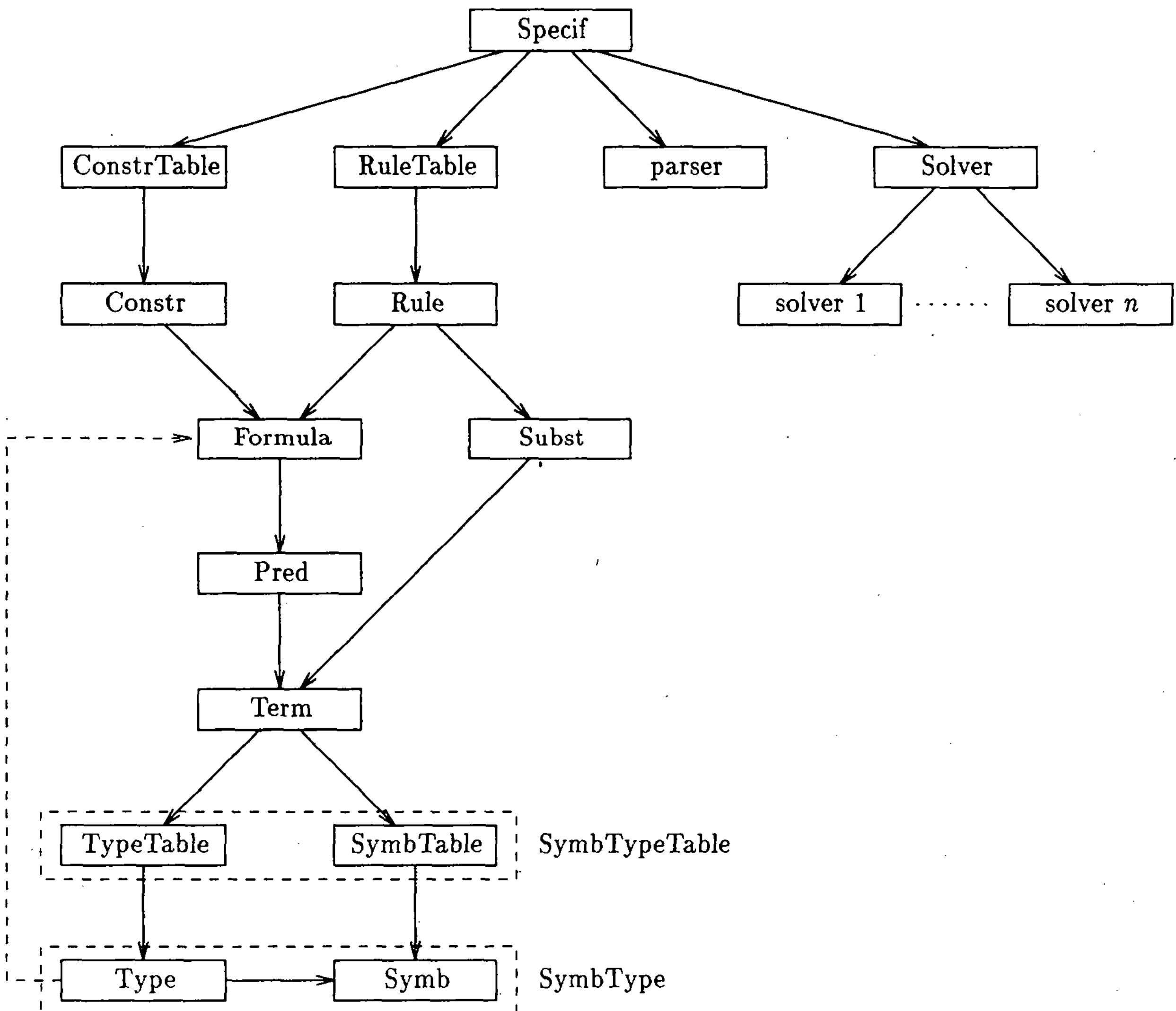


FIG. 5 - *Structure du programme*

Remarque 3 *Un effort particulier a été fait pour que le programme soit le plus souple possible. Les objets sont polymorphiques, c'est à dire par exemple qu'un prédicat manipule des termes sans connaître de quel type de terme il s'agit, ou qu'une formule manipule des prédicats sans connaître leurs types. Actuellement aucune table n'a donc de taille fixe, la taille des spécifications pouvant être traitée est donc (à l'efficacité près) uniquement limitée par la taille mémoire disponible. La spécification est un objet comme un autre et il sera donc par exemple possible de travailler avec plusieurs spécifications simultanément.*

7 Erreurs non traitées actuellement

Certaines erreurs ne sont pas détectés par le vérificateur, nous les donnons ci-dessous.

/* Petit exemple de demonstration */

module: exemple		
types:	$positif =$ **reel** $[0 <= @]$	
entrees:	$x, y : positif$	
	$z \quad :< a, b, c >$	
sorties:	$v, w : [0..10]$	
preconditions:	$v < y + 2;$	
bases:		
/* 0 */	$v = 2;$	
exceptions:		
/* 1 */	**si** $w < 10$	**alors** $z = a$ **et** $y = w - 1;$
/* 2 */	**si** $y > 10$ **et** $z \mathrel{!=} c$	**alors** $w = x + v;$
/* 3 */	**si** $z = c$	**alors** $w = 100;$

- La variable de sortie v apparaissant dans la précondition implique une erreur qui n'est pas détectée: v ne peut pas apparaître dans la précondition car c'est une variable de sortie donc sa valeur est a priori indéterminée lors de l'évaluation de la précondition.
- La variable de sortie w apparaissant dans la condition de la règle numéro 1 implique une erreur qui n'est pas détectée: w est indéterminé lors de l'évaluation de la condition.
- La variable de sortie w apparaissant dans de l'assignation de y dans la règle numéro 1 implique une erreur qui n'est pas détectée: w est indéterminé lors de l'évaluation de $w - 1$.
- L'assignation hors domaine de w dans la dernière règle implique une erreur qui n'est pas détectée.

Remarque 4 *Pour effectuer ce type de vérification, c'est le résultat du lemme 3 qui devra être utilisé.*

Un autre problème qui devra être réglé est celui du traitement propre des erreurs syntaxiques ce qui n'est pas le cas actuellement.

8 Conclusion

De façon générale l'intérêt de l'approche qui a été présentée peut être résumé comme suit : le langage de spécification défini est plus pauvre qu'un langage de programmation classique ainsi des propriétés qui sont indécidables en général peuvent être vérifiées automatiquement.
L'aspect le plus important du programme Versem est l'existence d'une base théorique. Cette aspect est très intéressant car outre la fiabilité du produit, il a permis de mettre en évidence certains cas particuliers oubliés lors de la spécification de Versem en langage naturel. D'un autre point de vue l'implantation en langage C++ s'est avérée être un bon choix. Ce choix a permis d'une part de faire un développement incrémental et ainsi de converger vers la version finale à partir de la maquette et d'autre part, les différents mécanisme de ce langage font de Versem un produit extensible en interdisant des modifications qui ne respectent pas la philosophie générale du programme.
Le programme sous sa version actuelle permet de contrôler les erreurs les plus fréquentes. Cependant certaines améliorations devrons être effectuées. En particulier on peut citer : la détection des erreurs non traitées et un traitement plus convivial des erreurs lors de l'analyse syntaxique. Certaines extensions de Versem sont envisageables, parmi ces dernières on peut citer :

- Ajouter des macros et des définitions de constantes.
- Ajouter le test de condition à n'importe qu'elle étape de la spécification.
- Ajouter le type entier qui est très fréquents dans un milieu industriel (nombre de paquets, ...).
- Ajouter (statiquement ou dynamiquement) des prédicats qui permettraient de définir un ensemble de propriétés a tester plus large.
- Ajouter la récursivité pour certains cas particuliers.

Il faut noter que l'utilisation d'un langage de programmation avec contraintes tel que CLP(R) [JL87] ou CHIP [DvHS+88] aurait également permis la réalisation d'un prototype. Les problèmes de complexité dus à la taille de l'espace de recherche nous ont amené à considérer des méthodes particulières de propagation de contraintes dont il n'est pas évident qu'elles auraient pu être décrites facilement dans ces langages. Par contre un environnement de description de simplification de contraintes tel qu'ELAN [KKV93] pourrait permettre de poursuivre dans cette direction. Mentionnons enfin qu'ELAN a été utilisé pour spécifier la syntaxe de Versem et la traduction des règles obtenues en Fortran.

A Description des fichiers

versem.cc	Programme principal
Specif.cc	Spécification
Specif.h	
ConstrTable.h	Table des contraintes
ConstrTable.o	
Constr.cc	Contraintes
Constr.h	
RuleTable.cc	Table des règles
RuleTable.h	
Rule.cc	Règles
Rule.h	
Formula.cc	Formules
Formula.h	
Subst.cc	Substitutions
Subst.h	
Pred.cc	Prédicats
Pred.h	
Term.cc	Termes
Term.h	
SymbTypeTable.cc	Table des types et des symboles
SymbTypeTable.h	
SymbType.cc	Descripteurs de type et de symboles
SymbType.h	
base_defs.h	Définitions de base
base_defs.cc	
Real.h	Définition des réels
parser.lex	Source flex analyseur lexical
parser_lex.cc	Source analyseur lexical engendré par flex
parser.y	Source bison Analyseur syntaxique
parser.cc	Source analyseur syntaxique engendré par bison
parser.h	
solver.cc	Interface avec les résolveurs
solver.h	
LpPresolve.cc	Résolveur contraintes type intervalle réel
LpPresolve.h	
lpcheck.cc	Résolveur contraintes linéaires sur réel
lpcheck.h	
enumcheck.cc	Résolveur contraintes type énuméré
enumcheck.h	
matrix.h	Matrice
Amatrix.cc	Matrice Arithmétique
Amatrix.h	

Makefile	Fichier de commande pour compiler versem
demo.spec	Spécification utilisée dans intro rapport
rapport.sep	Spécification utilisée dans exemple rapport
sollac.spec	Spécification donnée en exemple par Sollac
int.DLList.cc	Fichiers engendrés par genclass
int.DLList.h	(voir documentation lib-g++)
PtPred.List.cc	
PtPred.List.h	
PtPred.Set.cc	
PtPred.Set.h	
PtPred.SplayNode.cc	
PtPred.SplayNode.h	
PtPred.SplaySet.cc	
PtPred.SplaySet.h	
PtPred.defs.h	
PtSymb.PtTerm.Map.cc	
PtSymb.PtTerm.Map.h	
PtSymb.PtTerm.SplayMap.cc	
PtSymb.PtTerm.SplayMap.h	
PtSymb.Real.Map.cc	
PtSymb.Real.Map.h	
PtSymb.Real.SplayMap.cc	
PtSymb.Real.SplayMap.h	
PtSymb.SLList.cc	
PtSymb.SLList.h	
PtSymb.Set.cc	
PtSymb.Set.h	
PtSymb.SplayNode.cc	
PtSymb.SplayNode.h	
PtSymb.SplaySet.cc	
PtSymb.SplaySet.h	
PtSymb.defs.h	
PtTerm.defs.h	
PtType.SLList.cc	
PtType.SLList.h	
PtType.defs.h	
String.SLList.cc	
String.SLList.h	
String.SLStack.cc	
String.SLStack.h	
String.Stack.cc	
String.Stack.h	
String.defs.h	
Real.defs.h	
int.defs.h	

B Primitives des librairies g++ utilisé

Instance de la classe List pour le type PtPred (pointeur sur prédicat).

```
first()
length()
operator()(Pix)
operator=(PtPredList &)
next(Pix &)
append(PtPredList &,PtPredList &)
```

Instance de la classe SLLList pour le type PtSymb (pointeur sur symbole).

```
clear()
first()
ength()
operator()(Pix)
prepend(PtSymbSymb)
next(Pix &)
```

Instance de la classe SLLList pour le type PtType (pointeur sur symbole).

```
clear()
first()
operator()(Pix)
prepend(PtType)
next(Pix &)
```

Instance de la classe SLList pour la classe String.

```
append(String &)
first()
operator()(Pix)
next(Pix &)
```

Instance de la classe DLList pour les entiers.

```
append(int)
clear()
empty()
first()
length()
operator(Pix)
prepend(int)
prev(Pix &)
remove_front()
next(Pix &)
```

Instance de la classe SLStack pour la classe String.

del_top()
push(String &)
top()

Instance de la classe SplaySet pour PtPred (pointeur sur prédicat).

first()
operator()(Pix)
next(Pix &)

Instance de la classe SplaySet pour PtSymb (pointeur sur symbole).

clear()
first()
operator&=(PtSymbSplaySet &)
operator()(Pix)
operator—=(PtSymbSplaySet &)
next(Pix &)

Instance de la classe SplayMap pour (PtSymb,PtTerm)

clear()
contents(Pix)
del(PtSymb)
first()
key(Pix)
operator[](PtSymb)
seek(PtSymb)
next(Pix &)

Instance de la classe SplayMap pour(PtSymb,Real)

clear()
contents(Pix)
del(PtSymb)
first()
key(Pix)
operator[](PtSymb)
next(Pix &)

Classe BitSet

operator &=(BitSet)
operator -=(BitSet)
operator |=(BitSet)
operator ==(BitSet)

operator <=(BitSet)
clear()
set()
fist()
next()
last()
count()

Références

[DvHS+88] M. Dincbas, P. van Hentenryck, H. Simonis, A. Aggoun, T. Graf, and F. Berthier. The constraint logic programming language CHIP. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, volume 2, pages 693–702. Institute for New Generation Computer Technology, 1988.

[JL87] J. Jaffar and Jean-Louis Lassez. Constraint logic programming. In *Proceedings of the 14th Annual ACM Symposium on Principles Of Programming Languages, Munich (Germany)*, pages 111–119, 1987.

[KKV93] Claude Kirchner, Hélène Kirchner, and M. Vittek. Implementing computational systems with constraints. In Paris Kanellakis, Jean-Louis Lassez, and Vijay Saraswat, editors, *Proceedings of the first Workshop on Principles and Practice of Constraint Programming, Providence (R.I., USA)*, pages 166–175. Brown University, 1993.

Unité de Recherche INRIA Lorraine
Technopôle de Nancy-Brabois - Campus Scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 VILLERS LES NANCY Cedex (France)

Unité de Recherche INRIA Rennes IRISA, Campus Universitaire de Beaulieu 35042 RENNES Cedex (France)
Unité de Recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 GRENOBLE Cedex (France)
Unité de Recherche INRIA Rocquencourt Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)
Unité de Recherche INRIA Sophia Antipolis 2004, route des Lucioles - B.P. 93 - 06902 SOPHIA ANTIPOLIS Cedex (France)

EDITEUR
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 LE CHESNAY Cedex (France)

ISSN 0249 - 6399