

# Utilisation des CHRs pour générer des cas de test fonctionnel pour la Machine Virtuelle Java Card

S.-D. Gouraud et A. Gotlieb

IRISA-CNRS-INRIA, UMR 6074  
Campus Universitaire de Beaulieu  
35042 RENNES Cedex  
{gouraud,gotlieb}@irisa.fr

## Résumé

Le test fonctionnel basé sur une spécification formelle consiste à dériver des cas de test à partir d'un modèle formel pour détecter des fautes dans une implémentation. Dans nos travaux, nous étudions l'utilisation des *Constraint Handling Rules* (CHR) pour automatiser la génération de cas de test fonctionnel basée sur un modèle formel. Notre étude de cas est un modèle de la Machine Virtuelle Java Card (JCVM) écrit dans un sous-ensemble du langage Coq. Dans cet article, nous définissons une traduction automatique de ce modèle sous forme de règles CHR dans le but de générer des cas de test pour la JCVM. Le point clé de notre approche réside dans l'utilisation des *deep guards* pour modéliser fidèlement la sémantique de notre modèle formel. Ensuite, nous proposons une approche globale pour la génération de cas de test fonctionnel basée sur les CHR qui peut être appliquée à d'autres modèles formels.

## 1 Introduction

Le test fonctionnel consiste à 1) sélectionner des données de test à partir d'un modèle (formel), 2) exécuter une implémentation en utilisant cet ensemble de tests, puis 3) vérifier les résultats à l'aide d'un oracle. En test fonctionnel, un oracle est une procédure capable dans certaines situations de prédire les résultats attendus d'un programme. Les modèles formels classiquement utilisés en test fonctionnel sont les spécifications algébriques [7], les machines B [6] ou encore les machines à états finis. De telles spécifications formelles sont exploitées par les outils de preuve ou par les vérificateurs de modèles pour prouver la correction de l'implémentation mais quelquefois, il est souhaitable de les utiliser pour générer des ensembles de tests.

Dans nos travaux, nous nous intéressons au test fonctionnel basé sur des spécifications JSL (*Jakarta Specification Language*). JSL [8] est un langage formel issu d'un sous-ensemble du langage Coq et qui est basé sur des règles de réécriture conditionnelles. Coq est un

assistant de preuves destiné à l'origine à prouver des théorèmes d'ordre supérieur[1]. Le langage JSL a été défini afin d'obtenir un langage de spécification facile à lire et indépendant de tout outil. Dans le cadre du projet RNTL CASTLES [20] dont l'objectif est de définir un environnement pour automatiser la certification de la plate-forme Java Card [21], JSL a été choisi comme base commune. Une spécification JSL de la Machine Virtuelle Java Card a été définie [3, 4] dans le but de vérifier des propriétés de sécurité de la plate-forme Java Card.

La Machine Virtuelle Java Card (JCVM) exécute toutes les instructions (ou les bytecode) supportées par Java Card (`new`, `push`, `pop`, `invokestatic`, `invokevirtual`, etc.). Notre objectif consiste à générer des ensembles de tests à partir des spécifications JSL de chaque instruction, à traduire ces tests abstraits en tests exécutables, à les exécuter sur l'implémentation puis à vérifier les résultats par rapport aux spécifications. Dans cet article, nous nous intéressons à la première et à la dernière tâches qui sont des problèmes difficiles en test de logiciel. La concrétisation et l'exécution des tests ne sont pas discutées ici.

Dans nos travaux, nous avons étudié l'utilisation des *Constraint Handling Rules* (CHR) pour générer des cas de test et des oracles à partir de la spécification JSL de la JCVM. Nous définissons des objectifs de test basés sur la couverture de règles de réécriture et la détection de non-conformité, et nous fournissons des techniques pour générer des cas de test qui répondent à ces objectifs. Notre idée est de bénéficier de la haute déclarativité des CHR pour exprimer dans un même cadre à la fois les objectifs de test et les règles de la spécification JSL. Puis, en utilisant la propagation traditionnelle des CHR et l'énumération, nous générons des tests et des oracles comme solutions du système de contraintes.

Cet article est organisé de la façon suivante : la section 2 introduit la syntaxe de JSL et son modèle d'exé-

cution ; la section 3 est consacrée à la syntaxe et aux sémantiques des CHRs ; la section 4 introduit les règles de traduction qui permettent de passer de règles JSL à des règles CHRs ; dans la section 5 nous présentons deux techniques pour générer des cas de test fonctionnel selon des objectifs de test et nous discutons des applications au test de la JCVM ; les travaux connexes sont présentés en section 6, puis la section 7 termine l'article par nos perspectives de recherche.

## 2 Le Langage de Spécification Jakarta

Le Langage de Spécification Jakarta (JSL)[8] est un petit langage qui permet de décrire la machine virtuelle dans un style mathématique neutre. C'est un langage de premier ordre avec un système de type polymorphe où les fonctions sont décrites par des règles de réécriture conditionnelles.

### 2.1 La syntaxe

Les expressions du langage JSL sont des termes du premier ordre avec égalité ( $=$ ) construits à partir de variables et de symboles de constante. Un symbole de constante est soit un symbole de constructeur introduit par les définitions de type soit un symbole de fonction introduit par les définitions de fonction.

Soient  $\mathcal{C}$  un ensemble de symboles de constructeur,  $\mathcal{F}$  un ensemble de symboles de fonction et  $\mathcal{V}$  un ensemble de variables. L'ensemble des expressions JSL est l'ensemble  $\mathcal{E}$  de termes défini par  $\mathcal{E} ::= \mathcal{V} | \mathcal{E} == \mathcal{E} | \mathcal{C}\mathcal{E}^* | \mathcal{F}\mathcal{E}^*$ . Soit  $var$  une fonction définie sur  $\mathcal{E} \rightarrow \mathcal{V}^*$  qui retourne l'ensemble des variables d'une expression JSL.

Chaque symbole de fonction est défini par un ensemble de règles de réécriture conditionnelles. Ce formalisme peu commun en réécriture est en fait proche des langages fonctionnels avec filtrage et des assistants de preuves. La sémantique d'un symbole de fonction défini est décrite par un ensemble de règles de réécriture conditionnelles orientées :

$$l_1 \rightarrow r_1, \dots, l_n \rightarrow r_n \Rightarrow g \rightarrow d$$

où :

- $g = f v_1 \dots v_m$  avec  $\forall i, v_i \in \mathcal{V}$  et  $\forall i, j, v_i \neq v_j$
- $l_i$  est soit une fonction qui n'introduit pas de nouvelles variables soit une variable telle que  $1 \leq i \leq n$ ,  $var(l_i) \subseteq var(g) \cup var(r_1) \cup \dots \cup var(r_{i-1})$ .
- $r_i$  est une valeur appelée *motif* (*pattern*) et construite à partir de variables et de constructeurs. Un motif ne contient que des nouvelles variables, doit être linéaire<sup>1</sup> et tel que  $1 \leq i, j \leq n$  et  $i \neq j$ ,  $var(r_i) \cap var(g) = \emptyset$  et  $var(r_i) \cap var(r_j) = \emptyset$

<sup>1</sup>Toutes les variables doivent être différentes : le terme  $C v v$  où  $v \in \mathcal{V}$  et  $C \in \mathcal{C}$  n'est pas autorisé

- $d$  est une expression telle que  $var(d) \subseteq var(g) \cup var(r_1) \dots \cup var(r_n)$

Une règle signifie que si pour tout  $i$ , le terme  $l_i$  peut être réécrit en le motif  $r_i$ , alors l'expression  $g$  se réécrit en l'expression  $d$ . Par la suite, ces règles seront appelées des règles JSL.

L'exemple 1 donne une définition JSL de la fonction *plus* extraite de la spécification de la JCVM.

#### Exemple 1 (Les règles JSL pour définir *plus*)

*data nat* = 0 | *S nat*.

*function plus* :=

$$\begin{aligned} \langle plus\_r1 \rangle \quad n &\rightarrow 0 \\ &\Rightarrow (plus\ n\ m) \rightarrow m; \\ \langle plus\_r2 \rangle \quad n &\rightarrow (S\ p) \\ &\Rightarrow (plus\ n\ m) \rightarrow (S\ (plus\ p\ m)). \end{aligned}$$

Où  $n$ ,  $m$  et  $p$  sont des variables, 0 et  $S$  sont des symboles de constructeur et *plus* est un symbole de fonction. La construction  $\langle \dots \rangle$  permet de donner un nom à une règle : la première règle de *plus* est ainsi nommée  $\langle plus\_r1 \rangle$  et la seconde  $\langle plus\_r2 \rangle$ .

Des fonctions partielles et des fonctions indéterministes peuvent aussi être définies en JSL.

### 2.2 Le modèle d'exécution

Étant donné un terme  $e$ , nous rappelons que chaque sous-terme de  $e$  peut être identifié par une position  $p$  : le terme  $e|_p$  est le sous-terme de  $e$  qui se trouve à la position  $p$ . L'expression  $e[p \leftarrow d]$  signifie que dans le terme  $e$ , le sous-terme situé à la position  $p$  est remplacé par le terme  $d$ . Une substitution  $\theta$  peut être vue comme un renommage de variables.

Soit  $\mathcal{R}$  un ensemble de règles JSL. Une expression  $e$  est réécrite en une expression  $e'$  s'il existe une règle  $l_1 \rightarrow r_1, \dots, l_n \rightarrow r_n \Rightarrow g \rightarrow d$  dans  $\mathcal{R}$ , une position  $p$  dans  $e$  et une substitution  $\theta$  telles que :

- $e|_p = \theta g$  et  $e' = e[p \leftarrow \theta d]$
- pour  $1 \leq i \leq n$ ,  $\theta l_i \rightarrow^* \theta r_i$  où  $\rightarrow^*$  est la fermeture réflexive et transitive de  $\rightarrow$ .

#### Exemple 2

Étant donné la définition JSL de *plus*, le terme  $(plus\ 0\ (plus\ (S\ 0)\ 0))$  peut se réécrire en le motif  $(S\ 0)$  en trois dérivations différentes (ici, *confluentes*) :

$$\frac{(plus\ 0\ (plus\ (S\ 0)\ 0)) \rightarrow_{r1} (plus\ (S\ 0)\ 0)}{\rightarrow_{r2} (S\ (plus\ 0\ 0)) \rightarrow_{r1} (S\ 0)}$$

$$\frac{(plus\ 0\ (plus\ (S\ 0)\ 0)) \rightarrow_{r2} (plus\ 0\ (S\ (plus\ 0\ 0)))}{\rightarrow_{r1} (S\ (plus\ 0\ 0)) \rightarrow_{r1} (S\ 0)}$$

$$\frac{(plus\ 0\ (plus\ (S\ 0)\ 0)) \rightarrow_{r2} (plus\ 0\ (S\ (plus\ 0\ 0)))}{\rightarrow_{r1} (plus\ 0\ (S\ 0)) \rightarrow_{r1} (S\ 0)}$$

### 3 Les Constraint Handling Rules

Cette section est inspirée de l'article [12] de Thom Frühwirth, du livre [13] et du site Web dédié aux CHRs [22]. Le langage *Constraint Handling Rules* (CHR) est un langage à choix définitifs (*committed-choice*) qui se compose de règles gardées dont les multi-têtes se réécrivent en des contraintes plus simples jusqu'à ce qu'elles soient résolues. Ce langage étend toujours un langage hôte qui possèdent des capacités de résolution de contraintes. Des implémentations de CHRs sont disponibles en Eclipse Prolog, Sicstus Prolog, HAL, etc.

Les CHR sont basées sur deux principes : la **Simplification** qui consiste à remplacer des contraintes par de nouvelles contraintes logiquement équivalentes mais plus simples et la **Propagation** qui consiste à rajouter de nouvelles contraintes logiquement redondantes mais qui pourraient apporter de nouvelles simplifications.

Dans ce langage, une contrainte est un prédicat du premier ordre particulier. Ce prédicat est soit une contrainte prédéfinie (*built-in*) qui existe déjà dans le langage hôte soit une contrainte CHR (*user-defined*) qui est définie par l'utilisateur par un programme CHR (ensemble fini de règles CHR).

#### 3.1 La syntaxe

Il existe trois sortes de règles CHR :

- Les règles de simplification qui sont de la forme  
 $H \Leftarrow G \mid B$
- Les règles de propagation qui sont de la forme  
 $H \Rightarrow G \mid B$
- Les règles de simpagation qui sont de la forme  
 $H \setminus H' \Leftarrow G \mid B$

Plus précisément, les multi-têtes  $H$  et  $H'$  sont des conjonctions non-vides de contraintes CHR, la garde  $G$  est une conjonction de contraintes prédéfinies et le corps  $B$  est une conjonction non-vide de contraintes prédéfinies et de contraintes CHR.

Dans cette section, nous nous concentrons sur les règles de simplification et de propagation qui correspondent *explicitement* aux deux principes sur lesquelles reposent les CHRs.

#### Exemple 3 (Les règles CHR pour définir plus)

R1@ plus(A,B,R)  $\Leftarrow$  A=0 | R=B.

R2@ plus(A,B,R)  $\Leftarrow$  A=s(C)  
| plus(C,B,D), R=s(D).

C@ plus(A,B,R)  $\Rightarrow$  plus(B,A,R).

La construction ...@ permet de donner un nom à une règle CHR : la première règle de simplification de plus est ainsi nommée R1, la seconde R2 et la règle de propagation est nommée C.

#### 3.2 Les sémantiques

Étant donné une théorie sur les contraintes notée CT (avec *true*, *false* et une contrainte d'égalité =) qui détermine le sens des contraintes prédéfinies, l'interprétation déclarative d'un programme CHR est donnée par une conjonction de formules logiques universellement quantifiées. Il y a une formule par règle.

Si  $\bar{x}$  dénote les variables apparaissant dans la tête  $H$  et  $\bar{y}$  (resp.  $\bar{z}$ ) les variables apparaissant dans la garde (resp. corps) de la règle, alors :

- Si la garde est satisfaite, une règle CHR de simplification est une équivalence logique :  
 $\forall \bar{x}(\exists \bar{y}G \rightarrow (H \leftrightarrow \exists \bar{z}B))$
- Si la garde est satisfaite, une règle CHR de propagation est une implication :  
 $\forall \bar{x}(\exists \bar{y}G \rightarrow (H \rightarrow \exists \bar{z}B))$

La sémantique opérationnelle des programmes CHR est donnée par un système de transitions où chaque état est une conjonction de contraintes. L'état initial est le but à résoudre. Un état final est soit une conjonction consistante et dans ce cas c'est un succès, soit une conjonction inconsistante et dans ce cas c'est un échec. Il y a quatre transitions différentes : une transition pour résoudre les contraintes prédéfinies (Résoudre) et trois transitions pour appliquer chaque type de CHR (Simplifier, Propager et Simpagation).

#### Résoudre

Si  $C$  est une contrainte prédéfinie  
Et  $CT \models (C \wedge D) \leftrightarrow D'$   
Alors  $C, D \mapsto D'$

#### Simplifier

Si  $H \Leftarrow G \mid B$   
Et  $CT \models D \rightarrow \exists \bar{x}(H = H' \wedge G)$   
Alors  $H', D \mapsto D, H=H', B$

#### Propager

Si  $H \Rightarrow G \mid B$   
Et  $CT \models D \rightarrow \exists \bar{x}(H = H' \wedge G)$   
Alors  $H', D \mapsto H', D, H=H', B$

Les règles sont appliquées de manière équitable c'est-à-dire que toute règle qui est applicable à une chance d'être appliquée. De plus, la règle de propagation est appliquée au plus une fois sur une même contrainte afin d'éviter une non-terminaison triviale. Les programmes CHR peuvent être non-confluents et si les gardes des règles pour un même prédicat ne sont pas mutuellement exclusives alors ils peuvent être indéterministes.

#### Exemple 4 (Utilisation simple des CHRs)

Ces deux exemples montrent comment les CHRs fonctionnent.

$\mapsto_{R2}$  plus(s(0), s(0), R)  
 $\mapsto_{R1}$  plus(0, s(0), R1), R=s(R1)  
 $\mapsto_{Résoudre}$  R1=s(0), R=s(R1)  
R=s(s(0))

Ce second exemple exploite la règle de propagation : sans cette règle, la contrainte  $\text{plus}(M, s(0), s(s(0)))$  serait suspendue.

$$\begin{array}{l}
\mapsto_C \quad \text{plus}(M, s(0), s(s(0))) \\
\quad \text{plus}(M, s(0), s(s(0))), \\
\quad \text{plus}(s(0), M, s(s(0))) \\
\mapsto_{R2} \quad \text{plus}(M, s(0), s(s(0))), \\
\quad \text{plus}(0, M, s(0)) \\
\mapsto_{R1} \quad \text{plus}(M, s(0), s(s(0))), M=s(0) \\
\mapsto_{\text{Résoudre}} \quad \text{plus}(s(0), s(0), s(s(0))), M=s(0) \\
\mapsto_{R2} \quad \text{plus}(0, s(0), s(0)), M=s(0) \\
\mapsto_{R1} \quad s(0)=s(0), M=s(0) \\
\mapsto_{\text{Résoudre}} \quad M=s(0)
\end{array}$$

Le langage CHR a été conçu pour écrire des solveurs de contraintes efficaces et génériques [12]. L'exemple suivant montre un des principaux intérêts des CHRs :

#### Exemple 5

$$\begin{array}{l}
\text{plus}(M, 0, N) \\
\mapsto_C \quad \text{plus}(M, 0, N), \text{plus}(0, M, N) \\
\mapsto_{R1} \quad \text{plus}(M, 0, N), M=N \\
\mapsto_{\text{Résoudre}} \quad \text{plus}(M, 0, M), M=N
\end{array}$$

Ici, la relation  $M = N$  est déduite alors que ce ne serait pas le cas avec un solveur traditionnel sur les termes de Herbrand.

## 4 De JSL à CHR

La première étape de notre approche consiste à traduire automatiquement les spécifications JSL en CHR. Les règles de traduction sont données sous forme de jugements.

### 4.1 Les règles de traduction

Il y a trois sortes de règles de traduction : une règle pour les expressions, une pour les règles de réécriture dont le symbole est  $\rightarrow$  et une pour les règles JSL dont le symbole est  $\Rightarrow$ . Dans les jugements suivants, la variable  $r$  représentera toujours une variable fraîche.

Le jugement  $e \rightsquigarrow t \triangleleft \{C\}$  signifie que l'expression JSL  $e$  se traduit en un terme  $t$  sous les contraintes  $C$ . Le terme  $t$  peut être une variable Prolog, un atome ou une contrainte CHR.

$$\begin{array}{c}
\frac{}{v \rightsquigarrow v \triangleleft \{\text{true}\}} \quad \frac{}{c \rightsquigarrow c \triangleleft \{\text{true}\}} \\
\frac{e_1 \rightsquigarrow t_1 \triangleleft \{c_1\} \dots e_n \rightsquigarrow t_n \triangleleft \{c_n\}}{c e_1 \dots e_n \rightsquigarrow c(t_1, \dots, t_n) \triangleleft \{c_1, \dots, c_n\}} \\
\frac{e_1 \rightsquigarrow t_1 \triangleleft \{c_1\} \dots e_n \rightsquigarrow t_n \triangleleft \{c_n\}}{f e_1 \dots e_n \rightsquigarrow r \triangleleft \{c_1, \dots, c_n, f(t_1, \dots, t_n, r)\}}
\end{array}$$

Le jugement  $e \rightarrow p \rightsquigarrow C$  signifie que la règle qui permet de réécrire une expression JSL  $e$  en un motif

$p$  se traduit en un ensemble de contraintes  $C$ . Si l'expression  $e$  est une variable, l'ensemble de contraintes  $C$  est réduit à une contrainte d'égalité entre deux termes  $e=p$ . Si l'expression  $e$  est un appel de fonction, l'ensemble de contraintes  $C$  contient alors les contraintes qui portent sur les arguments de la fonction auxquelles s'ajoute la contrainte CHR définissant la fonction.

$$\frac{}{v \rightarrow p \rightsquigarrow v = p} \\
\frac{e_1 \rightsquigarrow t_1 \triangleleft \{c_1\} \dots e_n \rightsquigarrow t_n \triangleleft \{c_n\}}{f e_1 \dots e_n \rightarrow p \rightsquigarrow c_1, \dots, c_n, f(t_1, \dots, t_n, p)}$$

Le jugement  $l_1 \rightarrow r_1, \dots, l_n \rightarrow r_n \Rightarrow g \rightarrow d \rightsquigarrow g' \Leftrightarrow \text{guards} | \text{body}$  signifie que la règle JSL  $l_1 \rightarrow r_1, \dots, l_n \rightarrow r_n \Rightarrow g \rightarrow d$  se traduit par une règle CHR  $g' \Leftrightarrow \text{guards} | \text{body}$  où  $g'$  est la contrainte CHR associée à l'expression  $g$ ,  $\text{guards}$  est l'ensemble de contraintes correspondant à la traduction des règles de la forme  $l_i \rightarrow r_i$  et  $\text{body}$  est l'ensemble de contraintes qui correspondent à la traduction de l'expression  $d$ .

$$\frac{l_1 \rightarrow r_1 \rightsquigarrow g_1 \dots l_n \rightarrow r_n \rightsquigarrow g_n \quad e \rightsquigarrow t \triangleleft \{B\}}{l_1 \rightarrow r_1, \dots, l_n \rightarrow r_n \Rightarrow f v_1 \dots v_k \rightarrow e} \\
\rightsquigarrow f(v_1, \dots, v_k, r) \Leftrightarrow g_1, \dots, g_n | B, r = t.$$

Notons que l'indéterminisme, la confluence et la récurivité sont préservés par la traduction.

### 4.2 Les Deep Guards

Dans notre traduction, nous considérons que les gardes des CHR peuvent contenir à la fois des contraintes Prolog et des contraintes CHR. Cette approche, désignée sous le terme de *deep guards*, a beaucoup été étudiée par le passé. Par exemple, Smolka rappelle dans son article[17] que "les *deep guards* constitue le mécanisme central pour combiner les processus et la recherche (encapsulée) pour les problèmes de résolution". Les *deep guards* sont également utilisés dans plusieurs systèmes comme AKL, Oz [19] ou encore HAL[10]. Les *deep guards* sont basées sur la manière dont la satisfiabilité des gardes est testée quand elles sont composées non seulement de prédicats prédéfinis mais aussi d'appels de contraintes. Techniquement, le test d'implication d'une contrainte est appelée *ask constraint* alors que l'ajout d'une contrainte au magasin de contraintes est appelé *tell constraint*. Ces deux opérations sont clairement différentes. Par exemple, si le magasin de contraintes contient  $Z = f(X, Y), Y = a$  alors un *tell constraint*  $X = Y$  où  $=$  dénote l'unification de Prolog donnera un magasin de contraintes contenant  $Z = f(a, a)$  tandis que le *ask constraint* correspondant suspendra juste la contrainte jusqu'à ce qu'elle ou sa négation soit impliquée par le magasin de contraintes. Notons que chaque fois qu'une

clause CHR est réveillée, l'implication de ses gardes est testée. Pour une règle CHR donnée, si l'implication de sa garde est démontrée alors son corps est ajouté au magasin de contraintes.

L'approche actuelle<sup>2</sup> lorsque sont manipulées des *deep guards* contenant des buts Prolog (mais pas d'appels aux CHR) consiste à considérer les gardes comme des *tell constraints* et à vérifier à l'exécution qu'aucune variable contenue dans les gardes n'est modifiée. Cette approche est basée sur le fait que le seul moyen de contraindre les termes dans l'univers de Herbrand est l'unification. Notons que le *ask constraint* qui correspond à l'unification dans l'univers de Herbrand est bien connu : c'est le test d'égalité de termes. Par exemple, si  $X = Y$  est un *tell constraint* alors  $X == Y$  correspond à son *ask constraint*. Notons que cette approche n'est plus efficace lorsque de longs calculs sont considérés car les gardes sont testées à chaque fois qu'une clause CHR est réveillée. Une approche pour ce problème consiste à pré-calculer la garde en exécutant une seule fois le but Prolog puis à tester l'implication sur les arguments de la clause. Quand des buts Prolog sont impliqués dans les gardes, le test d'implication de la garde n'est plus décidable car des calculs non-terminant peuvent arriver. Cependant, dans notre cas, cela n'est pas possible car les gardes des CHRs sont obtenues par une traduction automatique de la spécification JSL de la JCVM [8] qui n'est composée que de règles terminantes.

Quand des appels aux CHRs sont présents dans les gardes, le problème est plus difficile car les gardes elles-mêmes peuvent rajouter des contraintes. Dans ce cas, considérer les gardes comme des *tell constraints* n'est plus correct et des déductions erronées peuvent être obtenues. Pour ce problème, notre approche consiste à suspendre le test d'implication de la garde jusqu'à ce qu'il devienne décidable. Plus précisément, le test d'implication de la garde est suspendu jusqu'à ce que toutes les variables de la gardes soient instanciées<sup>3</sup>. Au pire, cette instanciation arrive durant le processus d'énumération. Naturellement, cette approche amène peu de déductions dans le processus de propagation, mais nous pensons qu'elle reste acceptable pour la manipulation des *deep guards* qui contiennent des CHRs.

### 4.3 L'implémentation

Nous avons implémenté toutes ces règles de traduction dans la librairie `JSL2CHR.pl`. Étant donné un

<sup>2</sup>Cette approche est suivie dans beaucoup d'implémentations des CHRs comme dans Sicstus Prolog, Eclipse Prolog ou HAL[10]

<sup>3</sup>Naturellement cette solution est à rapprocher des techniques classiques de coroutinage en Prolog telles qu'implémentées par `freeze` ou `delay`

fichier contenant des définitions JSL, cette librairie construit un arbre de syntaxe abstraite en utilisant les DCG (*Definite Clause Grammar*) pour chaque définition JSL, puis produit automatiquement les règles CHR correspondantes.

Cette librairie a été utilisée pour traduire la spécification JSL de la JCVM qui est composée de 310 fonctions. 1537 règles CHR ont ainsi été automatiquement générées.

## 5 Génération de tests pour la JCVM

Dans le cadre du projet RNTL CASTLES, notre travail consiste à définir des techniques pour générer des cas de test fonctionnel pour une implémentation de la JCVM. Basée sur la traduction préservant la sémantique décrite en section 4, notre point de départ est un ensemble de règles CHR qui peuvent être considérées comme une spécification formelle. Cette section est consacrée à la manière dont nous générons les données de test et les oracles pour tester une implémentation de la JCVM par rapport à son modèle formel. Le test fonctionnel nécessitant des objectifs de test, nous en proposons deux qui sont à l'origine de deux techniques différentes.

Après avoir présentée brièvement la JCVM afin de pouvoir illustrer notre approche directement sur les règles qui nous intéressent, nous présentons les approches que nous proposons : la première est basée sur une couverture structurelle des règles JSL/CHR tandis que la seconde est basée sur des propriétés de sécurités que doit satisfaire la JCVM.

### 5.1 La JCVM

La spécification de la JCVM sur laquelle nous nous appuyons est automatiquement dérivée d'une formalisation en Coq précédemment réalisée dans un contexte de certification [5]. Dans cette formalisation, la JCVM fonctionne comme une machine à états avec une sémantique à petits pas : chaque instruction agit comme un transformateur d'état. Les entrées de la machine virtuelle sont un état initial et un programme. La sortie est un nouvel état issu de l'exécution du code associé au programme. Chaque état contient l'ensemble des éléments manipulés par un programme pendant son exécution : une pile pour la manipulation des valeurs, un tas pour le stockage des objets et un contexte d'exécution par méthode appelée. Les états sont formalisés par un enregistrement composé de trois champs : un tas (*he*) qui contient les objets créés pendant l'exécution, un tas statique (*sh*) qui contient les champs statiques des classes et une pile de contexte (*co*) qui contient les environnements pour exécuter les mé-

thodes. Les états sont étiquetés par Normal si l'exécution s'est bien passée et par Anormal si une exception (ou une erreur) est apparue. L'instruction *push* et la fonction *stack\_f* de l'exemple 6 sont directement extraits de la spécification JSL de la JCVM.

### Exemple 6 (Spécification JSL de push)

Étant donné un état *st* de la JCVM, la fonction *stack\_f* renvoie la pile de contexte *co* de cet état.

```
function stack_f :=
  (stack_f_r1) st → (Build_jcvm_state sh he co)
    ⇒ (stack_f st) → co.
```

Étant donné un type primaire *t*, une valeur *x* et un état *st* de la JCVM, si la pile de contexte de l'état *st* est vide alors la fonction **push** termine sur une erreur sinon la pile de contexte de l'état *st* est mis à jour à l'aide de la fonction *update\_frame* : en particulier, la valeur *x* de type *t* est rajoutée à la pile d'opérandes du premier contexte *h* à l'aide de la fonction *res*.

```
function push :=
  (push_r1) (stack_f st) → Nil
    ⇒ (push t x st) → (abortCode State_error st);
  (push_r2) (stack_f st) → (Cons h lf)
    ⇒ (push t x st) → (update_frame(res t x h) st).
```

L'exemple 7 donne les règles CHR produites automatiquement par notre librairie à partir des spécifications JSL données dans l'exemple 6.

### Exemple 7 (Règles CHR pour push)

```
stack_f_r1 @ stack_f(St,R) ⇔
  St=build_jcvm_state(Sh,He,Co)
  | R=Co.
push_r1 @ push(T,X,St,R) ⇔
  stack_f(St,nil)
  | abortCode(state_error(St),Ra), R=Ra.
push_r2 @ push(T,X,St,R) ⇔
  stack_f(St,cons(H,Lf))
  | res(T,X,H,Res), update_frame(Res,St,Ru),
  R=Ru.
```

## 5.2 Une règle, un test

Notre première approche s'inspire des techniques classiques de test fonctionnel qui consistent à générer des cas de test dans le but de tester chaque règle de la spécification. Ces techniques reposent sur deux hypothèses : la correction de la spécification et l'hypothèse d'uniformité [7]. Classiquement, la spécification est considérée comme une référence et est donc supposée correcte. L'hypothèse d'uniformité exprime le fait que si une règle a un comportement correct pour un cas de test alors ce comportement reste correct pour tous les cas de tests qui vont activer cette règle. Naturellement, cette hypothèse est très forte et rien ne peut

empêcher qu'elle soit violée mais c'est une hypothèse généralement faite dans le domaine du test fonctionnel. Rappelons que le test recherche des erreurs dans l'implémentation mais n'essaie pas de prouver l'absence d'erreurs.

### 5.2.1 Aperçu de l'approche

Pour chaque spécification JSL définissant une fonction, notre objectif est de trouver un cas de test (c'est-à-dire une substitution des variables d'entrée) qui permette d'activer chaque règle de cette spécification. Pour cela, un processus de recherche sur tous les termes possibles doit être effectué. Considérons, par exemple, le problème de couvrir toutes les règles JSL qui définissent l'instruction *push*. Pour activer la première règle *push\_r1*, la pile de contexte de l'état *st* doit pouvoir se réécrire en *Nil* (i.e. être vide) alors que pour activer la seconde règle *push\_r2*, la pile de contexte de l'état *st* doit pouvoir se réécrire en *Cons h lt* (i.e. contenir au moins un contexte). Notons qu'aucune contrainte n'est donnée sur les entrées *t* et *x* et dans ce cas, un processus de génération aléatoire est utilisé pour instancier ces variables. Pour assurer une couverture complète de l'instruction *push*, nous devons générer des triplets (*t, x, st*) tels que :

- *st* permette la réécriture suivante  
*stack\_f st* → *Nil*
- *st* permette la réécriture suivante  
*stack\_f st* → *Cons h lt*

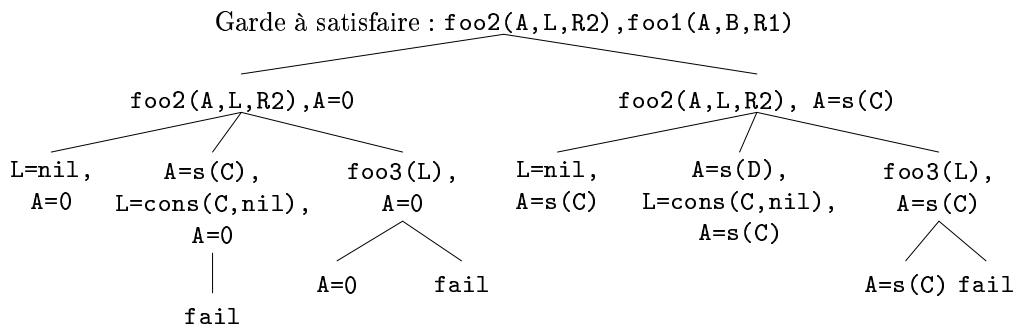
Notre processus de génération doit pouvoir générer, par exemple, les triplets solutions suivants<sup>4</sup> :

```
(Boolean, POS(XI(XO(XH))),
  Build_jcvm_state(Nil, Nil, Nil))
et
(Byte, NEG(XH), Build_jcvm_state(Nil, Nil,
  Cons(Build_frame(Nil, Nil, S(S(0)),
  Build_Package(0, S(0), Nil), True, S(0), Nil)))
```

Pour générer automatiquement ces entrées, nous utilisons les règles CHR correspondantes aux règles JSL et les techniques de résolution de contraintes (énumération et retour-arrière) afin d'unifier les variables d'entrées aux termes adéquats. Couvrir une règle JSL consiste alors à trouver des entrées qui satisfont la garde de la règle CHR associée.

Comme cela se fait habituellement en programmation par contraintes, les contraintes jouent ici un rôle actif puisque les relations sont exploitées avant la phase d'énumération (approche *tester puis générer*).

<sup>4</sup>*Build\_jcvm\_state, Build\_frame, Build\_Package, XI, X0, XH, POS, Byte, NEG, Boolean* et *True* sont des constructeurs définis dans la spécification JSL de la JCVM.

FIG. 1 – Arbre de recherche pour couvrir  $\text{foo2}(A, L, R2), \text{foo1}(A, B, R1)$ 

Notons que cette solution diffère des techniques classiques de test fonctionnel qui commencent par instancier les variables puis vérifient que les gardes sont satisfaites (approche *générer puis tester*).

### 5.2.2 Exploitation des contraintes

Cependant, exploiter les contraintes avant de lancer le processus d'énumération est plus complexe qu'il n'y paraît car dans notre cas, les *deep guards* sont acceptées. Quand une règle dont le garde contient à la fois des contraintes Prolog et des contraintes CHR est sélectionnée, la satisfaction des gardes sous-jacentes (i.e. les gardes des règles définissant les contraintes CHR présentes dans la garde) doit également être considérée. En effet, si l'on considère la règle CHR  $H \Leftrightarrow G \mid B$  où  $G$  est de la forme  $p_1, \dots, p_n$ , satisfaire la garde  $G$  consiste à satisfaire chaque contrainte CHR  $p_i$  c'est-à-dire à trouver une instantiation des variables telle que la contrainte se simplifie soit en *true* soit en une conjonction d'égalités cohérentes. L'appel au processus de simplification nécessite qu'au moins une des gardes des règles CHR définissant la contrainte  $p_i$  soit satisfaite. Nous formulons l'objectif de test pour chaque règle CHR par la formule suivante :

$$\bigwedge_i \left( \bigvee_j \text{garde}(p_i, j) \right)$$

où  $p_i$  est une contrainte CHR apparaissant dans  $G$  et  $\text{garde}(p_i, j)$  est la garde de la  $j$ ème règle définissant la contrainte CHR  $p_i$ . Toute solution de cette formule est une donnée de test possible qui permet d'activer la règle CHR considérée.

Pour trouver une solution à cette conjonction de contraintes, nous considérons l'arbre de recherche suivant :

- chaque noeud est un état où la recherche de solution peut arriver : c'est une conjonction de contraintes Prolog et de contraintes CHR ;

- chaque feuille est une conjonction de contraintes sans aucune contrainte CHR.

Étant donné un noeud de cet arbre et une contrainte CHR apparaissant dans ce noeud, chaque branche correspond au remplacement de la contrainte CHR par la garde d'une des règles CHR définissant cette contrainte.

Notons que cet arbre peut avoir des branches infinies, mais comme nous ne recherchons qu'une seule solution et non toutes les solutions (exhaustivité), certaines branches peuvent être coupées. De plus, il y a autant d'arbres de recherche que d'ordonnancement des contraintes à examiner et des règles qui les définissent. Dans notre cas, comme nous recherchons une seule solution, l'efficacité de la recherche peut être améliorée en utilisant des heuristiques qui permettent de guider le choix des branches à explorer dans l'arbre de recherche. Comme première heuristique, nous avons choisi de traiter les contraintes CHR d'un noeud de l'arbre de recherche dans l'ordre suivant :

1. Choisir la contrainte CHR définie par le moins de règles CHR ;
2. Si deux contraintes ont le même nombre de règles, utiliser l'ordre lexicographique.

Avec cette heuristique, nous choisirons, par exemple, d'abord de remplacer `stack_f`, puis `plus`, puis `push`. Si  $<$  est le symbole associé à cet ordre, nous avons alors `stack_f < plus < push`.

Dans l'arbre de recherche, si  $p_i < p_j$  alors  $p_i$  ne doit pas apparaître dans un sous-arbre dont la racine est une conjonction contenant la contrainte  $p_j$  mais pas  $p_i$ <sup>5</sup>. Si un noeud est une conjonction sans solution ou contenant une contrainte CHR qui ne peut pas être réduite, alors ce noeud est une feuille étiquetée `fail`. Une solution est une feuille qui n'est pas étiquetée `fail`.

<sup>5</sup>Ceci afin d'éviter les branches infinies issues des définitions récursives ou mutuellement récursives

**Exemple 8** *Considérons les règles CHR suivantes :*

```
foo(A,B,L,R) <=> foo2(A,L,R2),foo1(A,B,R1) | ...
foo1(A,B,R) <=> A=0 | ...
foo1(A,B,R) <=> A=s(C) | ...
foo2(A,L,R) <=> L=nil | ...
foo2(A,L,R) <=> A=s(B),L=cons(B,nil) | ...
foo2(A,L,R) <=> foo3(L,R1) | ...
foo3(L,R) <=> L=nil | ...
foo3(L,R) <=> L=cons(A,L2),foo1(A,A,R) | ...
```

*Les feuilles valides de l'arbre de la figure 1 correspondent aux différents cas de test qui permettent de couvrir la règle qui définit foo.*

### 5.2.3 Le processus d'énumération

Le processus d'énumération consiste à essayer d'instancier chaque variable libre d'un terme. Ce processus est basé sur des heuristiques déterministes ou randomisées [15]. Dans les méthodes de test de logiciel, la sélection randomisée est souvent utilisée car elle permet d'améliorer le pouvoir de détection des erreurs. Les heuristiques randomisées reposent sur des choix aléatoires basés sur une distribution probabiliste. L'approche la plus simple consiste à générer aléatoirement et uniformément des termes. De nombreux travaux ont été menés sur la génération uniforme de termes et sont liés à la génération aléatoire de structures combinatoires [11]. Dans nos travaux précédents [9], nous avons défini des techniques et utilisé un outil pour générer aléatoirement des structures combinatoires dans le contexte du test statistique. Nous pensons baser le processus d'énumération sur ces travaux.

Dans cette première approche, nous avons montré comment utiliser les CHR et les techniques de résolution de contraintes pour obtenir des cas de tests qui permettent de couvrir chaque règle d'une définition JSL d'une instruction de la JCVM. La règle à tester et l'objectif de test sont modélisés par des contraintes.

### 5.3 Test de conformité

Notre deuxième approche repose sur la notion de test de conformité qui consiste à tester si une implémentation satisfait une propriété de sécurité donnée. Les cas de test sont définis à partir d'un objectif de test qui est la propriété elle-même et doivent permettre de rejeter toute implémentation ne satisfaisant pas la propriété. Dans ce type d'approches, la spécification et la propriété sont supposées correctes. Dans la suite, nous supposons que la propriété est définie par des règles JSL.

L'idée est de générer des cas de test qui permettent de couvrir les règles JSL de la propriété en exploitant les spécifications JSL des fonctions/instructions sur lesquelles porte la propriété. Nous avons basé notre

technique sur un procédé de *projection* qui permet de croiser la propriété avec les spécifications qu'elle vise. Le procédé de projection consiste à remplacer les règles CHR définissant la propriété *prop* par des règles augmentées jusqu'à obtenir une décomposition de l'objectif de test initial en plusieurs objectifs de test plus précis.

L'opérateur principal du procédé de projection est une fonction *proj* qui étant donné deux règles CHR  $r_1$  et  $r_2$ , et une contrainte CHR  $p$  définie par la règle  $r_2$ , augmente la règle  $r_1$  en rajoutant la garde de  $r_2$  à la garde de  $r_1$  et en remplaçant la contrainte  $p$  dans le corps de  $r_1$  par le corps de  $r_2$  :

Si  $r_1$  est de la forme  $H \Leftarrow G' \mid p, B'$   
 Et  $r_2$  is  $q \Leftarrow G \mid B$   
 Et  $\exists \theta$  telle que  $\theta q = p$   
 Alors  $proj(r_1, r_2, p)$  est  $H \Leftarrow G', \theta G \mid \theta B, B'$

Étant donné une propriété *prop*, soient  $\mathcal{F}(prop)$  un ensemble de contraintes CHR visées par la propriété *prop*,  $\mathcal{P}(r, prop) \subseteq \mathcal{F}(prop)$  un ensemble de contraintes CHR apparaissant dans le corps de  $r$  et  $\mathcal{R}(p)$  l'ensemble des règles CHR définissant  $p$ . Le procédé de projection est définie par l'algorithme suivant :

Étant donné une règle  $r$  définissant la propriété *prop*

```
E = {r}.
Pour tout p dans P(r,prop)
  NewE = ∅
  Pour tout r_e dans E
    Pour tout r_p dans R(p)
      NewE = NewE ∪ {proj(r_e, r_p, p)}
E = NewE
```

Notons que toutes les règles dont les gardes qui sont (trivialement) insatisfiables sont retirées. Les gardes de toutes les règles apparaissant dans  $E$  correspondent aux cas de test permettant d'activer la règle  $r$ . Ce procédé est appliqué à chaque règle définissant la propriété *prop*.

Prenons par exemple, le cas de la propriété d'associativité de la fonction *plus*.

```
asso_r1 @ asso(A,B,C,R) <=>
  plus(B,C,R1), plus(A,R1,R2),
  plus(A,B,R3), plus(R3,C,R4),
  eq(R2,R4,R).
```

```
plus_r1 @ plus(A,B,R) <=> A=0 | R=B.
plus_r2 @ plus(A,B,R) <=> A=s(C)
  | plus(C,B,R1), R=R1(C).
```

```
eq_r1 @ eq(N,M,R) <=> N=0, M=0 | R=true.
eq_r2 @ eq(N,M,R) <=> N=0, M=s(P) | R=false.
eq_r3 @ eq(N,M,R) <=> N=s(P), M=0 | R=false.
eq_r4 @ eq(N,M,R) <=> N=s(P), M=s(Q)
  | eq(P,Q,R).
```

La sélection des cas de test à partir de la propriété d'associativité de la fonction `plus` consiste à instancier le triplet  $(A, B, C)$ . Après l'application de l'algorithme, la règle initiale `asso_r1` est projetée en quatre nouvelles règles :

```
asso(A, B, C, R) <=> B=0, A=0
| R1=C, R1=R2, R3=B, plus(R3, C, R4), eq(R2, R4, R) .
asso(A, B, C, R) <=> B=0, A=s(D)
| R1=C, R2=s(R5), R3=s(R6), plus(D, R1, R5),
  plus(D, B, R6), plus(R3, C, R4), eq(R2, R4, R) .
asso(A, B, C, R) <=> B=s(D), A=0
| R1=s(R5), R1=R2, B=R3,
  plus(D, C, R5), plus(R3, C, R4), eq(R2, R4, R) .
asso(A, B, C, R) <=> B=s(D), A=s(E)
| R1=s(R5), R2=s(R6), R3=s(R7),
  plus(D, C, R5), plus(E, R1, R6), plus(E, B, R7),
  plus(R3, C, R4), eq(R2, R4, R) .
```

La contrainte  $plus(R3, C, R4) \in \mathcal{P}(asso\_r1, asso)$  n'a pas été exploitée car aucune nouvelle contrainte sur  $A$ ,  $B$  ou  $C$  ne pouvait être ajoutée même de manière indirecte.

Une fois l'objectif de test initial découpé en plusieurs objectifs de test plus précis, il ne reste plus qu'à générer une donnée de test pour chacune des nouvelles règles en utilisant par exemple, l'approche présentée dans la section 5.2.

Bien que, nous n'ayons pas encore appliqué cette approche à la spécification JSL de la JCVM, nous présentons ici le type de propriétés de sécurité auquel nous allons devoir nous attaquer. La propriété suivante nommée FDP\_RIP.1 est extraite du document de Sun [2] qui définit les exigences de sécurité pour la plate-forme Java Card.

#### Propriété : Protection des informations résiduelles

*La plate-forme doit s'assurer que toute information contenue précédemment dans une ressource-mémoire est rendue indisponible lors de l'allocation de la ressource-mémoire aux objets suivants : instances de classe et tableaux. Lors de la création, les champs des objets et les éléments de tableaux sont initialisés avec des valeurs par défaut.*

Généralement, les propriétés sont données en langue naturelle, et la première étape non-triviale consiste à traduire ces propriétés dans un langage formel.

## 6 Travaux connexes

Bernot et al.[7] sont les pionniers dans l'utilisation de la programmation logique pour construire des ensembles de test à partir de spécifications formelles. Partant d'une spécification algébrique, les ensembles de test sont sélectionnés en utilisant la programmation logique équationnelle (clauses de Horn). Plus tard, la

programmation logique par contraintes a été exploitée par Gotlieb et al.[14] pour générer des ensembles de test pour le test structurel de programmes C. Étant donné le code d'un programme, un programme logique avec contraintes sémantiquement équivalent était construit et utilisé pour trouver des données de test qui couvrent un critère de test donné. Legeard et al.[6] ont proposé une méthode pour le test fonctionnel aux limites à partir de spécifications formelles décrites en  $B$  et en  $Z$ . Cette approche est basée sur des techniques de résolution de contraintes ensemblistes (CLP( $S$ )) et a été appliquée au test du mécanisme de transaction de Java Card.

Cependant, aucun de ces travaux n'a exploité les CHR pour la génération des cas de test, à l'exception de Lötzbeyer et Pretschner [16, 18]. Dans leurs travaux, ils proposent une technique de test qui utilise un solveur CHR. Les modèles sont des automates à états finis qui décrivent le comportement du système sous test et les cas de tests sont des séquences d'entrées/sorties. Les CHR sont utilisées pour définir de nouveaux solveurs de contraintes permettant la génération de données de type complexe.

Notre approche se distingue par la traduction systématique de la spécification formelle en CHR en utilisant les *deep guards*. De plus, elle n'impose aucune restriction sur la forme des gardes et semble être plus déclarative dans la génération des cas de test.

## 7 Conclusion

Dans cet article, nous avons proposé des techniques qui utilisent les CHR pour générer des cas de test fonctionnel pour une implémentation de la JCVM. La spécification JSL de la JCVM est utilisé comme modèle formel et la génération de tests a été orientée afin de satisfaire deux objectifs de test : la couverture structurelle de chaque règle JSL et la détection de la non-conformité par rapport à une propriété de sécurité.

Ce travail nous paraît prometteur : la traduction d'une spécification JSL en des contraintes CHR équivalentes est possible et permet une génération automatique des cas de tests. Il reste cependant à écrire une preuve formelle de la correction de la traduction et à générer un ensemble de tests complet pour la JCVM.

Au niveau du test, nous avons besoin de formaliser le processus de projection afin de décrire un nouveau critère de test basé sur des modèles formels. Nous avons également besoin d'étudier comment améliorer l'implémentation des *deep guards*.

À plus long terme, nous pensons pouvoir étendre facilement cette approche à d'autres modèles formels. Les CHR apparaissent donc comme étant un outil efficace pour générer des cas de test.

## 8 Remerciements

Nous tenons à remercier E. Coquery avec qui nous avons eu des discussions fructueuses sur les CHR, G. Dufay et G. Barthe qui nous ont fourni toutes les informations dont nous avons besoin sur les JSL, ainsi que les relecteurs pour leurs précieux conseils. Ces travaux sont financés par le projet RNTL CASTLES.

## Références

- [1] The Coq proof assistant. <http://coq.inria.fr/>.
- [2] Java Card System Protection Profile Collection (version 1.0b), 2003. <http://java.sun.com/products/javacard/pp.html>.
- [3] G. Barthe, G. Dufay, M. Huisman, and S. Sousa. Jakarta : a toolset for reasoning about JavaCard. In *Proceedings of E-smart 2001*, volume 2140 of *LNCS*, pages 2–18. In I. Attali and T. Jensen Eds, Springer-Verlag, 2001.
- [4] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, and S. Melo de Sousa. A Formal Executable Semantics of the JavaCard Platform. In *Proceedings of ESOP'01*, volume 2028 of *LNCS*, pages 302–319. D. Sands Eds, Springer-Verlag, 2001.
- [5] G. Barthe, G. Dufay, L. Jakubiec, B. Serpette, S. Melo de Sousa, and S-W. Yu. Formalization of the JavaCard Virtual Machine in Coq. In *Proceedings of FTfJP'00 (ECOOP Workshop on Formal Techniques for Java Programs)*, pages 50–56. S. Drossopoulou and al, Eds, 2000.
- [6] E. Bernard, B. Legeard, X. Luck, and F. Peureux. Generation of test sequences from formal specifications : GSM 11-11 standard case study. *International Journal of Software Practice and Experience*, 34(10) :915–948, 2004.
- [7] G. Bernot, M.-C. Gaudel, and B. Marre. Software testing based on formal specifications : a theory and a tool. *Software Engineering Journal*, 6(6) :387–405, 1991.
- [8] Simão Melo de Sousa. *Outils et techniques pour la vérification formelle de la plate-forme JavaCard*. PhD thesis, Université de Nice, février 2003.
- [9] A. Denise, M.-C. Gaudel, and S.-D. Gouraud. A Generic Method for Statistical Testing. In *Fifteenth IEEE International Symposium on Software Reliability Engineering*, pages 25–34, 2004.
- [10] G.J. Duck, P.J. Stuckey, M. Garcia de la Banda, and C. Holzbaur. Extending arbitrary solvers with constraint handling rules. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP03)*, page 2003, 79-90.
- [11] Ph. Flajolet, P. Zimmermann, and B. Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theoretical Computer Science*, 132 :1–35, 1994.
- [12] T. Frühwirth. Theory and Practice of Constraint Handling Rules. *Logic Programming*, 37(1-3), october 1998. Special Issue on Constraint Logic Programming, In P. Stuckey and K. Marriott Eds.
- [13] T. Frühwirth and S. Abdennadher. *Essentials of Constraint Programming*. Cognitive Technologies. Springer Verlag, 2003. ISBN 3-540-67623-6.
- [14] A. Gotlieb, B. Botella, and M. Rueher. A CLP Framework for Computing Structural Test Data. In *Constraints Stream, First International Conference on Computational Logic*, number 1891 in *LNAI*, pages 399–413. Springer-Verlag, 2000.
- [15] S.-D. Gouraud, A. Denise, M.-C. Gaudel, and B. Marre. A New Way of Automating Statistical Testing Methods. In *Sixteenth IEEE International Conference on Automated Software Engineering (ASE)*, pages 5–12, 2001.
- [16] H. Lötzbeyer and A. Pretschner. AutoFocus on Constraint Logic Programming. In *Proceedings of (Constraint) Logic Programming and Software Engineering (LPSE'2000)*, july 2000.
- [17] Andreas Podelski and Gert Smolka. Situated Simplification. In *Proceedings of the 1st Conference on Principles and Practice of Constraint Programming*, volume 976 of *LNCS*. Springer-Verlag, 1995.
- [18] A. Pretschner and H. Lötzbeyer. Model Based Testing with Constraint Logic Programming : First Results and Challenges. In *Proceedings 2nd ICSE Intl. Workshop on Automated Program Analysis, Testing and Verification*, 2001.
- [19] C. Schulte. Programming deep concurrent constraint combinators. In Enrico Pontelli and Vitor Santos Costa, editors, *Second International Workshop on Practical Aspects of Declarative Languages*, volume 1753 of *LNCS*, pages 215–229. Springer-Verlag, 2000.
- [20] INRIA Sophia-Antipolis, IRISA, AQL, and Oberthur. <http://www-sop.inria.fr/everest/projects/castles/>.
- [21] JavaCard Technology. <http://java.sun.com/products/javacard>.
- [22] CHR website. <http://www.cs.kuleuven.ac.be/~dtai/projects/CHR/>.