Using CHRs to generate functional test cases for the Java Card Virtual Machine*

Sandrine-Dominique Gouraud and Arnaud Gotlieb

IRISA/CNRS UMR 6074,
Campus Universitaire de Beaulieu,
35042 Rennes Cedex, FRANCE
Phone: +33 (0)2 99 84 75 76 – Fax: +33 (0) 2 99 84 71 71
gouraud@lri.fr, gotlieb@irisa.fr

Abstract. Automated functional testing consists in deriving test cases from the specification model of a program to detect faults within an implementation. In our work, we investigate using Constraint Handling Rules (CHRs) to automate the test cases generation process of functional testing. Our case study is a formal model of the Java Card Virtual Machine (JCVM) written in a sub-language of the Coq proof assistant. In this paper we define an automated translation from this formal model into CHRs and propose to generate test cases for each bytecode definition of the JCVM. The originality of our approach resides in the use of CHRs to faithfully model the formally specified operational semantics of the JCVM. The approach has been implemented in Eclipse Prolog and a full set of test cases have been generated for testing the JCVM.

Keywords: CHR, Software testing, Java Card Virtual Machine.

1 Introduction

The increasing complexity of computer programs ensures that automated software testing will continue to play a prevalent role in software validation. In this context, automated functional testing consists in 1) generating test cases from a specification model, 2) executing an implementation using the generated test cases and then 3) checking the computed results with the help of an oracle. In automated functional testing, oracles are generated from the model to provide the expected results. Several models have been used to generate test cases: algebraic specifications [1], B machineries [2] or finite state machines [3], just to name a few.
In our work, we investigate using Constraint Handling Rules (CHRs) to automate the test cases and oracles generation process of functional testing. Our specification model is written in a sub-language of Coq:

* This work is supported by the Réseau National des Technologies Logicielles as part of the CASTLES project (www-sop.inria.fr/everest/projects/castles/). This project aims at defining a certification environment for the JavaCard platform. The project involves two academic partners: the Everest and Lande teams of INRIA and two industrial partners: Oberthur Card Systems and Alliance Qualit Logicielle.

the Jakarta Specification Language (JSL) [4]. Coq is the INRIA's proof assistant [5] based on the calculus of inductive constructions that allows to mechanically prove high-order theorems. Recently, Coq and JSL were used to derive certified Byte Code Verifiers by abstraction from the specification of a Java Card Virtual Machine [4, 6]. The Java Card Virtual Machine (JCVM) carries out all the instructions (or bytecodes) supported by Java Card (new, push, pop, invokestatic, invokevirtual, etc.). In this paper, we present how to generate test cases and oracles for each JSL byte code specification. Our idea is to benefit from the high declarativity of CHRs to express the test purpose as well as the JSL specification rules into a single framework. Then, by using traditional CHR propagation and labelling, we generate test cases and oracles as solutions of the underlying constraint system. The approach has been implemented with the CHR library of Eclipse Prolog [7] and a full set of test cases have been generated for testing the JCVM.
This paper is organised as follows: Section 2 introduces JSL and its execution model; Section 3 recalls some background on CHRs; Section 4 introduces the translation rules used to convert a formal specification written in JSL into CHRs; Section 5 presents our algorithm to generate functional test cases and oracles for testing an implementation of the JCVM; Section 6 describes some related works, and finally Section 7 concludes the paper with some research perspectives.

2 The Jakarta Specification Language

The Jakarta Specification Language (JSL), as introduced in [8], is a first order language with a polymorphic type system. JSL functions are formally defined with conditional rewriting rules.

2.1 Syntax

JSL expressions are first order terms with equality (==), built from term variables and from constant symbols. A constant symbol is either a constructor symbol introduced by data types definitions or a function symbol introduced by function definitions.
Let $\mathcal{C}$ be a set of constructor symbols, $\mathcal{F}$ be a set of function symbols and $\mathcal{V}$ be a set of term variables. The JSL expressions set is the term set $\mathcal{E}$ defined by: $\mathcal{E} ::= \mathcal{V} | \mathcal{E} == \mathcal{E} | \mathcal{C}\mathcal{E}^* | \mathcal{F}\mathcal{E}^*$. Let var be the function defined on $\mathcal{E} \rightarrow \mathcal{V}^*$ which returns the set of variables of a JSL expression.
Each function symbol is defined by a set of conditional rewriting rules. This unusual format for rewriting is close to functional language with pattern-matching and proof assistant. These (oriented) conditional rewriting rules are of the form $l_1 \rightarrow r_1, \ldots, l_n \rightarrow r_n \Rightarrow g \rightarrow d$ where:

- $g = f v_1 \ldots v_m$ where $\forall i, v_i \in \mathcal{V}$ and $\forall i, j, v_i \neq v_j$
- l_i is either a variable or a function which does not introduce new variables: for $1 \leq i \leq n$, $var(l_i) \subseteq var(g) \cup var(r_1) \cup \ldots \cup var(r_{i-1})$
- r_i should be a value called *pattern* (built from variables and constructors), should contain only fresh variables and should be linear[1]:

[1] All the variables are required to be distinct

for $1 \leq i, j \leq n$ and $i \neq j$, $var(r_i) \cap var(g) = \emptyset$ and $var(r_i) \cap var(r_j) = \emptyset$
- d is an expression and $var(d) \subseteq var(g) \cup var(r_1) \ldots \cup var(r_n)$

The rule means if for all i, l_i can be rewritten into r_i then g is rewritten into d. Thereafter, these rules are called JSL rules. JSL allows the definition of partial or non-deterministic functions.

Example 1 (JSL def. of **plus** *extracted from the JCVM formal model).*
$data\, nat \;=\; 0 \mid S\, nat.$
$function$ **plus** :=
$\langle plus\_r1 \rangle \quad n \;\rightarrow\; 0 \qquad \Rightarrow\; (plus\, n\, m) \;\rightarrow\; m;$
$\langle plus\_r2 \rangle \quad n \;\rightarrow\; (S\, p) \Rightarrow\; (plus\, n\, m) \;\rightarrow\; (S\; (plus\, p\, m)).$

2.2 Execution model of JSL

Let $e_{|p}$ denote the subterm of e at position p then expression $e[p \leftarrow d]$ denotes the term e where $e_{|p}$ is replaced by term d.
Let $\mathcal{R}$ be a set of rewriting rules, then an expression e is rewritten into e' if there exists a rule $l_1 \rightarrow r_1, \ldots, l_n \rightarrow r_n \Rightarrow g \rightarrow d$ in $\mathcal{R}$, a position p and a substitution θ such as:
- $e_{|p} = \theta g$ and $e' = e[p \leftarrow \theta d]$
- $\{\theta l_i \rightarrow^* \theta r_i\}_{\forall 1 \leq i \leq n}$ where $\rightarrow^*$ is the transitive cloture of $\rightarrow$

Note that nothing prevents JSL specifications to be non-terminating or non-confluent. However, the formal model of the JCVM we are using as a case study has been proved terminating and confluent within the Coq proof assistant [4, 6].

Example 2 (Rewriting of $(plus\ 0\ (plus(S\ 0)\ 0))$*).*
$(plus\ 0\ (plus\ (S\ 0)\ 0)) \rightarrow_{r1} (plus\ (S\ 0)\ 0) \rightarrow_{r2} (S\ (plus\ 0\ 0)) \rightarrow_{r1} (S\ 0)$

3 Background on Constraint Handling Rules

This section is inspired of Thom Frühwirth's survey and book [9, 10]. The Constraint Handling Rules (CHRs) language is a committed-choice language, which consists of multi-headed guarded rules that rewrite constraints into simpler ones until they are solved. This language extends a host language with constraint solving capabilities. Implementations of CHRs are available in Eclipse Prolog [7], Sicstus Prolog, HAL [11], etc.

3.1 Syntax

The CHR language is based on **simplification** where constraints are replaced by simpler ones while logical equivalence is preserved and **propagation** where new constraints which are logically redundant are added to cause further simplification. A constraint is either a built-in (predefined) first-order predicate or a CHR (user-defined) constraint defined by a finite set of CHR rules. Simplification rules are of the form `H <=> G | B` and propagation rules are of the form `H ==> G | B` where H denotes a possibly multi-head CHR constraint, the guard G is a conjunction of constraints and the body B is a conjunction of built-in and CHR constraints.

Each time a CHR constraint is woken, its guard must either succeed or fail. If the guard succeeds, one commits to it and then the body is executed. Constraints in the guards are usually restricted to be built-in constraints. When other constraints are used in the guards (called *deep guards*), special attention must be paid to the way guards are evaluated. Section 4.2 discusses the use of *deep guards* in our framework.

Example 3 (CHRs that can be used to define the `plus` *constraint).*

```
R1 @ plus(A,B,R) <=> A=0    | R=B.
R2 @ plus(A,B,R) <=> A=s(C) | plus(C,B,D), R=s(D).
C  @ plus(A,B,R) ==> plus(B,A,R).
```

The construction ...`@` gives names to CHRs.

3.2 Semantics

Given a constraint theory (CT) (with true, false and an equality constraint =) which determines the meaning of built-in constraints, the declarative interpretation of a CHR program is given by a conjunction of universally quantified logical formula. There is a formula for each rule.
If $\bar{x}$ denotes the variables occurring in the head H and $\bar{y}$ (resp. $\bar{z}$) the variables occurring in the guard (resp. body) of the rule, then

- a simplification CHR is interpreted as $\forall\bar{x}(\exists\bar{y}G \rightarrow (H \leftrightarrow \exists\bar{z}B))$
- a propagation CHR is interpreted as $\forall\bar{x}(\exists\bar{y}G \rightarrow (H \rightarrow \exists\bar{z}B))$

The operational semantics of CHR programs is given by a transition system where a state $< G, C >$ consists of two components: the goal store G and the constraint store C. An initial state is of the form $< G, true >$. A final state $< G, C >$ is successful when no transition is applicable whereas it is failed when $C = false$ (the constraint store is contradictory).

Solve If C is a built-in constraint and $CT \models (C \wedge D) \leftrightarrow D'$
Then $< C \wedge G, D > \mapsto < G, D' >$
Simplify If $F <=> D|H$ and $CT \models \forall(C \rightarrow \exists\bar{x}(F = E \wedge D)$
Then $< E \wedge G, C > \mapsto < H \wedge G, (F = E) \wedge D \wedge C >$
Propagate If $F => D|H$ and $CT \models \forall(C \rightarrow \exists\bar{x}(F = E \wedge D)$
Then $< E \wedge G, C > \mapsto < E \wedge H \wedge G, (F = E) \wedge D \wedge C >$

Rules are applied fairly (every rule that is applicable is applied eventually). Propagation rule is applied at most once on the same constraints in order to avoid trivial non-termination. However, CHR programs can be non-confluent and non-terminating.

Example 4 (Several examples of the CHR solving process).

	`plus(s(0),s(0),R)`
$\mapsto_{\textbf{Simplify\_R2}}$	`plus(0,s(0),R1), R=s(R1)`
$\mapsto_{\textbf{Simplify\_R1}}$	`R1=s(0), R=s(R1)`
$\mapsto_{\textbf{Solve}}$	`R=s(s(0))`

The following example exploits the propagation rule of **plus**. Without this rule, the term `plus(M,s(0),s(s(0)))` would be delayed.

$$
\begin{array}{ll}
 & \texttt{plus(M,s(0),s(s(0)))} \\
\mapsto_{\textbf{Propagate\_C}} & \texttt{plus(M,s(0),s(s(0))),}\ \underline{\texttt{plus(s(0),M,s(s(0)))}} \\
\mapsto_{\textbf{Simplify\_R2}} & \texttt{plus(M,s(0),s(s(0))),}\ \underline{\texttt{plus(0,M,s(0))}} \\
\mapsto_{\textbf{Simplify\_R1}} & \texttt{plus(M,s(0),s(s(0))), M=s(0)} \\
\mapsto_{\textbf{Solve}} & \underline{\texttt{plus(s(0),s(0),s(s(0)))}}\texttt{, M=s(0)} \\
\mapsto_{\textbf{Simplify\_R2}} & \underline{\texttt{plus(0,s(0),s(0))}}\texttt{, M=s(0)} \\
\mapsto_{\textbf{Simplify\_R1}} & \texttt{s(0)=s(0), M=s(0)} \\
\mapsto_{\textbf{Solve}} & \texttt{M=s(0)}
\end{array}
$$

The following example shows the deduction of a relation ($M = N$):

$$
\begin{array}{ll}
 & \texttt{plus(M,0,N)} \\
\mapsto_{\textbf{Propagate\_C}} & \texttt{plus(M,0,N),}\ \underline{\texttt{plus(0,M,N)}} \\
\mapsto_{\textbf{Simplify\_R1}} & \texttt{plus(M,0,N), M=N} \\
\mapsto_{\textbf{Solve}} & \texttt{plus(M,0,M), M=N}
\end{array}
$$

4 JSL to CHR translation method

Our approach is based on the syntactical translation of JSL specifications into CHRs. The translation method is described under the form of judgements.

4.1 Translation method

There are three kinds of judgements: judgements for JSL expressions, judgements for JSL rewriting rules (main operator $\rightarrow$) and judgements for JSL functions (main operator $\Rightarrow$).
The judgement $e \rightsquigarrow \mathtt{t} \lhd \{\mathtt{C}\}$ states that JSL expression e is translated into term $\mathtt{t}$ under the conjunction of constraints $\mathtt{C}$.

$$
\frac{variable(v)}{v \rightsquigarrow \mathtt{v} \lhd \{\mathtt{true}\}} \qquad\qquad \frac{constant(c)}{c \rightsquigarrow \mathtt{c} \lhd \{\mathtt{true}\}}
$$

$$
\frac{e_1 \rightsquigarrow \mathtt{t_1} \lhd \{\mathtt{c_1}\} \ \ \ldots \ \ e_n \rightsquigarrow \mathtt{t_n} \lhd \{\mathtt{c_n}\}}{c\, e_1 \ldots e_n \rightsquigarrow \mathtt{c}(\mathtt{t_1}, \ldots, \mathtt{t_n}) \lhd \{\mathtt{c_1}, \ldots, \mathtt{c_n}\}}
$$

$$
\frac{e_1 \rightsquigarrow \mathtt{t_1} \lhd \{\mathtt{c_1}\} \ \ \ldots \ \ e_n \rightsquigarrow \mathtt{t_n} \lhd \{\mathtt{c_n}\}}{f\, e_1 \ldots e_n \rightsquigarrow \mathtt{r} \lhd \{\mathtt{c_1}, \ldots, \mathtt{c_n}, \mathtt{f}(\mathtt{t_1}, \ldots, \mathtt{t_n}, \mathtt{r})\}}
$$

The judgement $(e \rightarrow p) \rightsquigarrow \{\mathtt{C}\}$ states that the JSL rewriting rule $e \rightarrow p$ is translated into the conjunction of constraints $\{\mathtt{C}\}$.

$$
\frac{}{(v \rightarrow p) \rightsquigarrow \{\mathtt{v} = \mathtt{p}\}}
$$

$$
\frac{e_1 \rightsquigarrow \mathtt{t_1} \lhd \{\mathtt{c_1}\} \ \ \ldots \ \ e_n \rightsquigarrow \mathtt{t_n} \lhd \{\mathtt{c_n}\} \quad p \rightsquigarrow \mathtt{p} \lhd \{\mathtt{true}\}}{(f\, e_1 \ldots e_n \rightarrow p) \rightsquigarrow \{\mathtt{c_1}, \ldots, \mathtt{c_n}, \mathtt{f}(\mathtt{t_1}, \ldots, \mathtt{t_n}, \mathtt{p})\}}
$$

The judgement $(l_1 \rightarrow r_1, \ldots, l_n \rightarrow r_n \Rightarrow g \rightarrow d) \rightsquigarrow \mathtt{g}' \Leftrightarrow \mathtt{guard}|\mathtt{body}$ states that the JSL function rule $l_1 \rightarrow r_1, \ldots, l_n \rightarrow r_n \Rightarrow g \rightarrow d$ is translated into the CHR $\mathtt{g}' \Leftrightarrow \mathtt{guard}|\mathtt{body}$ where $\mathtt{g}'$ is a CHR constraint associated to the expression g, $\mathtt{guard}$ is the conjunction of constraints corresponding to the translation of the rules $l_i \rightarrow r_i$, and `body` is a conjunction of constraints corresponding to the translation of the expression d.

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

Note that non-determinism, confluence and termination are preserved by the translation as the operational semantics of CHRs extends the execution model of JSL functions.

4.2 Deep guards

In the translation method, we considered that CHR guards could be built over prolog goals and CHR calls. This approach, which is referred to as deep guards, has received much attention by the past. See [9, 12] for a detailed presentation of deep guards. Smolka recalls in [13] that "deep guards constitute the central mechanism to combine processes and (encapsulated) search for problem-solving". Deep guards are used in several systems such as AKL, Eclipse Prolog [7, 9], Oz [12] or HAL [11]. Deep guards rely on how guard entailment is tested in conditional constraints and CHRs. Technically, a guard entailment test is called an "ask constraint" whereas a constraint added to the constraint store is called a "tell constraint" and both operations are clearly distinct. For example, if the constraint store contains $X = p(Z), Y = p(a)$ then a tell constraint $X = Y$ where $=$ denotes Prolog unification, will result in the store $X = p(a), Y = p(a), Z = a$ whereas the corresponding ask constraint will leave the store unchanged and will suspend until the constraint $Z = a$ would be entailed or disentailed.
The current approach to deal with deep guards that contain Prolog goals (but not CHR calls) consists in considering guards as tell constraints and checking at runtime that no guard variable is modified. This approach is based on the fact that the only way of constraining terms in the Herbrand Universe is unification (=) and that the corresponding ask constraint of unification is well-known: this is the "equality of terms" test (==). For example, if $X = Y$ is a tell constraint then $X == Y$ corresponds to its ask constraint. However, when Prolog goals are involved into the guards, the guard entailment test is no more decidable as non-terminating computations can arise. Note that CHR programs are not guaranteed to terminate (consider for example $p <=> true|p$). Even when non-terminating computations are avoided this approach can be very inefficient as possible long term computations in guards are executed every time a CHR constraint is woken. An approach for this problem consists in pre–computing the guard by executing the Prolog goal only once, and then testing entailment on the guard variables.
When CHRs are involved into the guards, the problem is more difficult as guards can set up constraints. In that case, considering guards as tell constraints is no longer correct as wrong deductions can be made. Our approach for this problem consists in suspending the guard entailment

test until it could be decided. More precisely, the guard entailment test is delayed until all the guard variables become instantiated[2]. At worst, this instantiation arises during the labelling process. Of course, this approach leads to fewer deductions at propagation time but it remains manageable when we have to deal with deep guards containing CHR calls.

4.3 Implementation of the translation method

We implemented the translation method into a library called `JSL2CHR.pl`. Given a file containing JSL definitions, the library builds an abstract syntax tree by using a Definite Clause Grammar of JSL, and then automatically produces equivalent CHR rules. The library was used on the JSL specifications of the JCVM, which is composed of 310 functions. As a result, 1537 CHRs were generated.

5 Tests generation for the JCVM

This section is devoted to the presentation of both the JCVM specification model and the test cases and oracle generation method. The experimental results we obtained by generating test cases for the JCVM are presented in Section 5.3.

5.1 The Java Card Virtual Machine

Unlike other smart cards, a Java Card includes a Java Virtual Machine implemented in its read-only memory part. The structure of a Java Card platform is given in Fig.1. It consists of several components, such as a runtime environment, an implementation of the Java Virtual Machine, the open and global platform applications, a set of packages implementing the standard SUN's Java Card API and a set of proprietary APIs. A Java Card program is called an applet and communicates with a card reader through APDU[3] buffers.
All the components of a Java Card platform must be thoroughly tested before the Card would be released. But, in this paper, we concentrate only on the JCVM functional testing process. In the formal model given in [14], the JCVM is a state machine described by a small-step semantics: each bytecode is formalised as a state transformer.

States modelling Each state contains all the elements manipulated by a program during its execution: values, objects and an execution environment for each called method. States are formalised as a record consisting of a heap (he) which contains the objects created during execution, a static heap (sh) which contains static fields of classes and a stack of frames (fr) which contain the execution environments of methods. States are tagged "Abnormal" if an exception (or an error) is raised, "Normal" otherwise.

[2] This solution is close to the traditional techniques of coroutining in Prolog as implemented by `freeze` or `delay` built-in predicates.

[3] Application Protocol Data Unit is an ISO-normalised communication format between the card and the off-card applications.

Bytecodes modelling The JCVM contains 185 distinct bytecodes which can be classified into the following classes[15]: arithmetic operations (`sadd`, `idiv`, `sshr`, ...), type verifications on objects (`instanceof`, ...), (conditional) branching (`ifcmp`, `goto`, ...), method calls (`invokestatic`, `invokevirtual`, ...), operations on local variables (`iload`, `sstore`, ...), operations on objects (`getfield`, `newarray`, ...), operations on operands stack (`ipush`, `pop`, ...) and flow modifiers (`sreturn`, `throw`, ...).
Most of the bytecodes have a similar execution scheme: to decompose the current state, to get components of the state, to perform tests in order to detect execution errors then to build the next state. In the JSL formal model of the JCVM, several bytecodes are specified with the similar JSL functions. They only distinguish by their type which is embodied in the JSL function definition as a parameter. As a result, the model contains only 45 distinct JSL functions associated to the bytecodes. Remaining functions are auxiliary functions that perform various computations. Some JSL functions calls other functions in their rewriting rules; this process is modelled by using deep guards in CHR, preserving so the operational semantics of the JCVM.

Example of a JSL bytecode specification As an example, consider the JSL specification of bytecode *push*: given a primary type t, a value x and a JCVM state st, *push* updates the operand stack of the first execution method environment in st by adding the value x of type t:
$function$ **push** :=
$\langle push\_r1\rangle\ (stack\_f\ st) \rightarrow Nil$
$\quad \Rightarrow (push\ t\ x\ st) \rightarrow (abortCode\ State\_error\ st);$
$\langle push\_r2\rangle\ (stack\_f\ st) \rightarrow (Cons\ h\ lf)$
$\quad \Rightarrow (push\ t\ x\ st) \rightarrow (update\_frame(result\_push\ t\ x\ h)\ st).$

push uses the auxiliary function **stack_f** that returns the stack of frames (environments for executing methods) of a given state.
$function$ **stack_f** :=
$\langle stack\_f\_r1\rangle\ st \rightarrow (Jcvm\_state\ sh\ he\ fr) \Rightarrow (stack\_f\ st) \rightarrow fr.$

Example of CHR generated for a bytecode The following CHRs were produced by the library `JSL2CHR.pl`:

```
stack_f_r1 @ stack_f(St,R) <=> St=jcvm_state(Sh,He,Fr)
    | R=Fr.
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))
    | result_push(T,X,H,Res), update_frame(Res,St,Ru), R=Ru.
```

In this example, the JSL function **stack_f** was translated into a CHR although it is only an accessor. As a consequence we get a deep guard in the definition of CHR `push`. This could be easily optimised by identifying the accessors into the JSL specification with the help of the user. However, we would like the approach to remain fully automated hence we did not realized this improvement and maintained the deep guards.

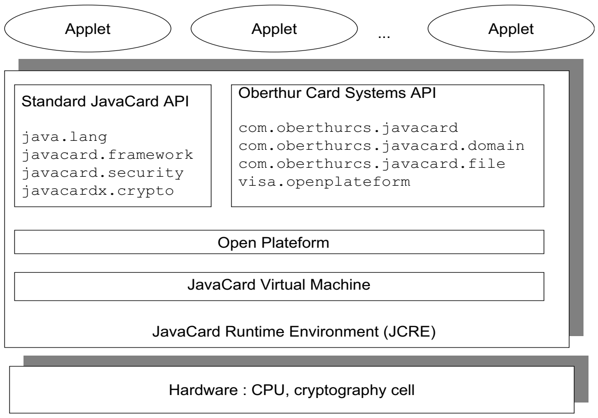

Fig. 1. A Java Card platform

5.2 Test cases and oracles generation method

Our approach is inspired of classical functional testing where test cases are generated according to some coverage criteria. We proposed to generate test cases that ensure each CHR would be covered at least once during the selection. We call this criterion *All_rules*. Note that this approach is based on two usual assumptions, namely the correctness of the formal specification and the uniformity hypothesis[1]. The uniformity hypothesis says that if a rule provides a correct answer for a single test case then it will provide correct answers for all the test cases that activate the rule. Of course, this assumption is strong and nothing can prevent it to be violated but recall that testing can only detect faults within an implementation and cannot prove the correctness of the implementation (as stated by Prof. E. Dijkstra).

Abstract test cases In the JSL formal model of the JCVM, a test case consists of a fully instantiated state of the VM and the valuation of several input parameters. However, it happens that several values of the state or several parameter values remain useless when testing a selected bytecode. To deal with these situations, the notion of abstract test case is used. In our case, an abstract test case represents a class of test cases that activate a given JSL function or equivalently a given CHR. The process which consists to instantiate an abstract test case to actually test an implementation is called concretization [2] and can be delayed until the test-execution time. For each CHR automatically generated, the goal is to find a minimal substitution of the variables (an abstract test case) that activate it. Covering a CHR consists in finding input values such as its guard would be satisfied. Hence, a constrained search process over the guards and the possible substitutions is performed. Before going to more details into this process, consider the CHRs of bytecode *push*. To activate *push_r1*, the states stack St must be empty whereas to activate *push_r2*, St must be rewritten into *cons* H Lf (i.e. to posses at least one

frame). Note that H, Lf, T and X are not constrained and do not require to be instantiated in the abstract test case. However, a randomised labelling can be used and to generate the two following concrete test cases, written under the form of JSL expressions[4]:
$(Bool, POS(XI(XO(XH))), Jcvm\_state(Nil, Nil, Nil))$ and
$(Byte, NEG(XH), Jcvm\_state(Nil, Nil, Cons(Frame(Nil, Nil, S(S(0)), Package(0, S(0), Nil), True, S(0)), Nil)))$.

A constrained search process over the guards As usual in constraint programming, we would like to see the constraints playing an active role by exploiting the relations before labelling (test-and-generate approach). Note that this contrasts with classical functional testing techniques that usually instantiate first the variables and then check if they satisfy the requirements (generate-and-test approach).
Consider a CHR $\mathtt{r} : \mathtt{H} \Leftrightarrow \mathtt{G}|\mathtt{B}$ where $\mathtt{G} = p_1, \ldots, p_n$. Satisfying the guard G requires to satisfy at least one guard of the CHRs that define each predicate $\mathtt{p_i}$ of G, i.e. finding a valuation such as $\mathtt{p_i}$ is simplified either in `true` or in a consistent conjunction of equalities. When $\mathtt{p_i}$ himself is a CHR call (deep guards), then its guard and body are also required to be consistent with the rest of the constraints. According to the *All_rules* testing criterion, the constraint store takes the following form:

$$\bigwedge_i \left(\bigvee_j (guard(\mathtt{p_i}, j) \wedge body(\mathtt{p_i}, j)) \right)$$

where $guard(\mathtt{p_i}, j)$ (resp. $body(\mathtt{p_i}, j)$) denotes the guard (resp. body) of the jth rule defining $\mathtt{p_i}$. Any solution of this constraint store can be interpreted as a test case that activates the CHR under test. Finding a solution to this constraint store leads to explore a possibly infinite search tree, as recursive or mutually recursive CHR are allowed. However, a simple occur-check test permits to avoid such problems. In this work, we followed a heuristic which consists to select first the guard with the easier guard to satisfy. A guard was considered easier to satisfy than another when it contains a smaller number of deep guards. The idea behind this heuristic is to avoid the complex case during the generation. This approach is debatable as these complex cases may contain the more subtle faults. See section 5.3 for a discussion on possible improvements. Note that the constraint store consistency is checked before going into a next branch, hence constraints allows pruning the search tree before making a choice. Note also that the test case generation process requires only to find a single solution and not all solutions, hence a breath-first search could be performed to avoid infinite derivations.

Oracles generation As the CHR specification of the JCVM is executable and the formal model is supposed to be correct, oracles can be generated just by interpreting the CHR program with generated test cases. For example, the following request gives us the oracle for the test

[4] *Jcvm_state*, *Frame*, *Package*, *XI*, *X*0, *XH*, *POS*, *Byte*, *NEG*, *Bool* and *True* are JSL constructor symbols given in the JCVM formal model.

case generated for *push_r*1:

```
?- push(bool,pOS(xI(xO(xH))),jcvm_state(nil,nil,nil)),R).
R=abnormal(jCVMError(eCode(state_error)),jcvm_state(nil,nil,nil))
```

Providentially, oracles can also be derived for abstract test cases. For example, oracle for abstract test case of *push_r*1 is computed by the following request: `?- push(T,X,jcvm_state(Sh,He,nil)),R).`

```
R=abnormal(jCVMError(eCode(state_error)),jcvm_state(Sh,He,nil))
```

When delayed goals are present, a labelling process must be launched to avoid suspension. For example, the following request obtained by using the generated abstract test cases for *push_r*2:

```
?- push(T,X,jcvm_state(Sh,He,cons(H,Lf))),R).
T=T, X=X, Sh=Sh, He=He, H=H, Lf=Lf, R=R
Delayed goals: push(T,X,jcvm_state(Sh,He,cons(H,Lf)),R)
```

requires R to be unified to $cons(\_X, \_S)$ to wake up the suspended goal. The labelling process can be based on deterministic or randomised[16] labelling strategies. In software testing approaches, random selection is usually preferred as it improves the flaws detection capacity. The simplest approach consists in generating terms based on a uniform distribution. Lot of works have been carried out to address the problem of uniform generation of terms and are related to the random generation of combinatorial structures [17]. In a previous work [18], we proposed a uniform random test cases generation technique based on combinatorial structures designs.

5.3 Experimental results

As previously said, the library `JSL2CHR.pl` generated 1537 CHRs that specify 45 JCVM bytecodes. The library generates a CHR program that is compiled by using the *ech* library of Eclipse Prolog [7]. We present the experimental results we obtained by generating abstract test cases for covering all the 443 CHRs associated to the bytecodes of the JCVM. These results were obtained on an Intel Pentium M at 2GHz with 1GB of RAM under Linux Redhat 2.6 The full process of generation of the abstract test cases for the 45 bytecodes (443 test cases) took 3.4s of CPU time and 47 Mbytes as the global stack size, 0.3 Mbytes as the local stack size and 2.6 Mbytes as the trail stack size. The detailed results for each bytecode are given in Tab.1, ordered by increasing number of abstract test cases (second column). Tab.1 contains the stack sizes as well as the CPU time (excluding time spent in garbage collection and system calls) required for the generation.

Analysis and discussion The approach ensures the coverage of each rule of the JSL bytecodes in a very short period of CPU time. The global and trail stacks remain stable whereas the local stack size increases with the number of test cases. A possible explanation is that some CHRs exit non-deterministically and allocation of variables cannot be undone in this case. We implemented a heuristic which consists to favour the CHRs that contain the smallest number of deep guards. This heuristic behaves well as shown by the short CPU time required for the bytecodes

Name	#tc	global stack (bytes)	local stack (bytes)	trail stack (bytes)	runtime (ms)
aload	1	33976048	148	1307064	0
arraylength	1	33849512	148	1299504	0
astore	1	33945864	148	1306660	0
invokestatic	1	33849512	148	1299504	0
nop	1	33945864	148	1306660	0
aconstnull	2	33854760	424	1300336	0
goto	2	33951112	424	1307492	0
jsr	2	33951112	424	1307492	0
push	2	33951112	424	1307492	0
conv	3	34055304	1076	1315924	10
dup	3	33972696	992	1310252	0
getfield	3	33876344	992	1303096	10
getfield_this	3	33876344	992	1303096	0
neg	3	34055304	1076	1315924	11
new	3	33971904	1020	1309932	11
pop	3	33972152	992	1310156	0
pop2	3	34074480	1076	1317828	0
putfield	3	33885840	1076	1303944	0
putfield_this	3	33876344	992	1303096	0
dup2	4	34122280	1884	1322772	10
swap	4	34029448	1884	1315948	11
ifnull	5	34023088	2216	1315392	10
ifnonnull	5	33926736	2216	1308236	10
icmp	6	34409440	3480	1343428	50
if_acmp_cond	6	34012528	3624	1316892	20
const	7	33968512	1512	1309716	0
invokespecial	7	34027448	4020	1317168	20
if_cond	8	34047496	3772	1319316	10
ret	8	34059064	3940	1320780	11
invokevirtual	9	34432272	7632	1349576	60
arith	11	33948080	836	1306660	0
athrow	11	34596760	7648	1364512	90
invokeinterface	11	35007240	12104	1394104	120
newarray	13	34073536	7604	1325612	20
return	13	34889544	11448	1386532	91
if_scmp_cond	14	34349752	11004	1351220	49
inc	18	34305392	9568	1344628	29
lookupswitch	18	34117768	9548	1332008	29
tableswitch	18	34117768	9548	1332008	31
load	19	34263232	11672	1343060	30
store	25	34752536	20108	1390876	81
checkcast	30	35053520	21384	1419380	280
getstatic	33	34408808	20652	1360596	60
putstatic	34	34944800	28660	1416196	120
instanceof	62	36468800	46964	1555588	580

Table 1. Memory and CPU runtime measures for each bytecode

that are specified with a lot of CHRs (`instanceof` is specified with 62 CHRs and only 0.6s of CPU time is required to generate the 62 abstract test cases). However, most of the time, this heuristic leads to generate test cases that put the JCVM into an abnormal state. In fact, in the JSL specification of the JCVM the abnormal states can often be reached by corrupting an input parameter. As a consequence, they are easy to reach. Although this heuristic is suitable to reach our test purpose (covering *All_rules*) and corresponds to some specific testing criterion such as *Test_all_corrupting_input* , it is debatable because it does not represent the general behaviour. Other approaches, which could lead to better test cases, need to be studied and evaluated. For example, selecting first the guard that contains the greatest number of deep guards could lead to build test cases that activate interesting parts of the specification. Finally, in these experiments, we only generated abstract test cases and did not evaluate the time required in the concretization step. Although, this step does not introduce research problems, considering it would allow to get a more accurate picture of test case and oracle generation with CHR. Thus, we could evaluate the efficiency of our approach and compare it to existing techniques.

6 Related Work

Bernot and al. [1] pioneered the use of Logic Programming to construct a test set from a formal specification. Starting from an algebraic specification, the test cases were selected using Horn clauses Logic. More recently, Gotlieb and al. [19] proposed to generate test sets for structural testing of C programs by using Constraint Logic Programming over finite domains. Given the source code of a program, a semantically-equivalent constraint logic program was built and questioned to find test data that cover a selected testing criterion. Legeard and al.[2] proposed a method for functional boundary testing from B and Z formal specifications based on set constraint solving techniques (CLP($\mathcal{S}$)). They applied the approach to the transaction mechanism of Java Card that was formally specified in B. Test cases were only derived to activate the boundary states of the specification of the transaction mechanism. Only Lötzbeyer and Pretschner [20, 21] proposed a software testing technique that uses CHR constraint solving. In this work, models are finite state automata describing the behaviour of the system under test and test cases are composed of sequence of input/output events. CHR is used to define new constraint solvers and permits to generate complex data types. Our work distinguishes by the systematic translation of formal specifications into CHRs. Our approach does not restrict the form of guards in CHR and appears so as more declarative to generate test cases.

7 Conclusion

In this paper, we have proposed to use the CHRs to generate functional test cases for a JCVM implementation. A JSL formal specification of the JCVM has been automatically translated into a CHR program and a test cases and oracles generation process has been proposed. The method permits to generate 443 test cases to test the 45 bytecodes formally specified. This result shows that the proposed approach scales up to a real-world example.

However, as discussed previously, other approaches need to be explored and evaluated. In particular, the coverage criterion *All_rules* initially selected appears as being too restrictive and other testing criteria could be advantageously used. Moreover, the test concretization step need to be studied in order to compare the efficiency of our approach against existing methods.

Finally, the key point of the approach resides in the use of deep guards, although their treatment needs to be evaluated both from the analytic and the experimental points of view.

8 Acknowledgements

We wish to acknowledge E. Coquery for fruitful discussions on CHR, and G. Dufay and G. Barthe who gave us the JSL formal model of the JCVM.

References

1. Bernot, G., Gaudel, M.C., Marre, B.: Software testing based on formal specifications : a theory and a tool. Software Engineering Journal **6** (1991) 387–405
2. Bernard, E., Legeard, B., Luck, X., Peureux, F.: Generation of test sequences from formal specifications: GSM 11-11 standard case study. International Journal of Software Practice and Experience **34** (2004) 915–948
3. Grieskamp, W., Gurevich, Y., Schulte, W., Veanes, M.: Generating finite state machines from abstract state machines. In: ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis, New York, NY, USA, ACM Press (2002) 112–122
4. Barthe, G., Dufay, G., Huisman, M., Sousa, S.: Jakarta: a toolset for reasoning about JavaCard. In: Proceedings of E-smart 2001. Volume 2140 of LNCS., In I. Attali and T. Jensen Eds, Springer-Verlag (2001) 2–18
5. INRIA: The Coq proof assistant (1999) `http://coq.inria.fr/`.
6. Barthe, G., Dufay, G., Jakubiec, L., Serpette, B., de Sousa, S.M.: A Formal Executable Semantics of the JavaCard Platform. In: Proceedings of ESOP'01. Volume 2028 of LNCS., D. Sands Eds, Springer-Verlag (2001) 302–319

7. Brisset, P., Sakkout, H., Fruhwirth, T., Gervet, C., Harvey, e.a.: ECLiPSe Constraint Library Manual. International Computers Limited and Imperial College London, UK. (2005) Release 5.8.
8. de Sousa, S.M.: Outils et techniques pour la vérification formelle de la plate-forme JavaCard. PhD thesis, Université de Nice (2003)
9. Frühwirth, T.: Theory and Practice of Constraint Handling Rules. Logic Programming **37** (1998) Special Issue on Constraint Logic Programming, In P. Stuckey and K. Marriott Eds.
10. Frühwirth, T., Abdennadher, S.: Essentials of Constraint Programming. Cognitive Technologies. Springer Verlag (2003) ISBN 3-540-67623-6.
11. Duck, G., Stuckey, P., de la Banda, M.G., Holzbaur, C.: Extending arbitrary solvers with constraint handling rules. In: Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming (PPDP03). (79-90) 2003
12. Schulte, C.: Programming deep concurrent constraint combinators. In Pontelli, E., Costa, V.S., eds.: Second International Workshop on Practical Aspects of Declarative Languages. Volume 1753 of LNCS., Springer-Verlag (2000) 215–229
13. Podelski, A., Smolka, G.: Situated Simplification. In Montanari, U., ed.: Proceedings of the 1st Conference on Principles and Practice of Constraint Programming. Volume 976 of LNCS., Springer-Verlag (1995) 328–344
14. Barthe, G., Dufay, G., Jakubiec, L., Serpette, B., de Sousa, S.M., Yu, S.W.: Formalization of the JavaCard Virtual Machine in Coq. In: Proceedings of FTfJP'00 (ECOOP Workshop on Formal Techniques for Java Programs), S. Drossopoulou and al, Eds (2000) 50–56
15. Dufay, G.: Vérification formelle de la plate-forme Java Card. PhD thesis, Université de Nice-Sophia Antipolis (2003)
16. Gouraud, S.D., Denise, A., Gaudel, M.C., Marre, B.: A New Way of Automating Statistical Testing Methods. In: Sixteenth IEEE International Conference on Automated Software Engineering (ASE). (2001) 5–12
17. Flajolet, P., Zimmermann, P., Van Cutsem, B.: A calculus for the random generation of labelled combinatorial structures. Theoretical Computer Science **132** (1994) 1–35
18. Denise, A., Gaudel, M.C., Gouraud, S.D.: A Generic Method for Statistical Testing. In: Fifteenth IEEE International Symposium on Software Reliability Engineering. (2004) 25–34
19. Gotlieb, A., Botella, B., Rueher, M.: A CLP Framework for Computing Structural Test Data. In: Constraints Stream, First International Conference on Computational Logic. Number 1891 in LNAI, Springer-Verlag (2000) 399–413
20. Pretschner, A., Lötzbeyer, H.: 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)
21. Lötzbeyer, H., Pretschner, A.: AutoFocus on Constraint Logic Programming. In: Proceedings of (Constraint) Logic Programming and Software Engineering (LPSE'2000). (2000)