The KRAKATOA Tool for Certification of JAVA/JAVACARD Programs annotated in JML

C. Marché, C. Paulin-Mohring and X. Urbain

PCRI — LRI, CNRS UMR 8623 — LogiCal Project, INRIA Futurs
Bât. 490, Université Paris-Sud, Centre d'Orsay
91405 Orsay Cedex, France

Abstract

We describe the basic structure of an environment for proving JAVA programs annotated with JML specifications. Our method is generic with respect to the API, and thus well suited for JAVACARD applets certification. It involves three distinct components: the WHY tool, which computes proof obligations for a core imperative language annotated with pre- and post-conditions, the COQ proof assistant for modeling the program semantics and conducting the development of proofs, and finally the KRAKATOA tool, a translator of our own, which reads the JAVA files and produces specifications for COQ and a representation of the semantics of the JAVA program into WHY's input language.

1 Introduction

The JAVA Modeling Language [1,2], JML for short, has been designed to specify the behavior of JAVA programs formally. It is based on JAVA expressions and consequently can be easily integrated in the development environment of JAVA programmers. In particular, it has a main interest in the context of JAVACARD [3,4] programs, short programs which require high level of confidence. An example is the specification in JML of the JAVACARD API [5]. One of the goals of the VERIFICARD European project is to design tools for formal verification of JML specifications, and we propose such a tool, called KRAKATOA.

The general goal we seek is verifying that a JML-annotated method of a JAVA or JAVACARD program meets its specifications. More precisely, we focus on verifying the soundness of implementations with reference to pre/post-conditions (given in the specification) and class invariants as well as exceptional behavior. This amounts to proving that class invariants, as well as post-conditions, hold at the end of a method execution, provided that invariants and pre-conditions were valid at the beginning.

Supported features. Verification takes place at source code level and there are few differences between the treatment of JAVA code and the treatment of JAVACARD code. We are dealing only with sequential programs without dynamic loading. Actually, JAVACARD limitations such as forbidding multi-dimensional arrays will not need any particular treatment, we simply make the assumption that input programs are accepted by the JAVA compiler and, in the case of JAVACARD programs, the CAP bytecode converter. We do not have any particular support for the JAVACARD firewall and transaction mechanisms, so we only handle what can be specified directly in JML in the `JCSystem` class of the JAVACARD API [6]. Thorough the article, we study JAVA programs but everything is valid for JAVACARD also.

We only support part of JML [2]. Roughly speaking, The JML constructs taken into account are: class *invariants*; *requires*, *assignable*, *ensures* and *signals* clauses for each method as well as *loop invariants* and *decreases* clauses for each while-loop or for-loop. Regarding the *decreases* clauses, that means we prove the termination of the loops. On the other hand, for recursive methods we do not handle the *measured_by* clauses, and prove only partial correctness, that is we do not prove their termination. In assertions, we support the specific constructs `\old`, `\result`, `\forall`, `\exists`, `\fresh`, `\not_modified`, and specific constructs for *assignable* clauses (see Section 5). Other constructs like `\max` or `\sum` are not yet supported, since they require particular care because these expressions are only defined when the variable is restricted to a finite set. We also support `model` fields which are interpreted as new fields in the class; but there is not yet support for the associated `represents` clause.

Related work. Several tools exist which manipulate JAVA programs annotated with JML specifications [7]. Their objectives can be different, they may aim at producing code with dynamic testing, or generating programs for unit testing of classes, or proving properties of programs. The ESC/JAVA system [8] tries to detect automatically simple errors such as dereferencing of the null pointer or out-of-bounds access to an array. It can be useful to discover bugs in programs but its approach is neither sound nor complete.

Other approaches like LOOP [9–11], JIVE [12], JACK [13] or our tool KRAKATOA are intended to generate for any JML specification of the program, verification conditions for these properties to hold. The generated conditions can be proved interactively with partial automation using proof strategies. These tools are based on different techniques. The JACK environment [13] was developed by the Gemplus company. From the code annotated with JML specifications, it generates a set of proof obligations using a calculus of weakest preconditions [14] and uses the B system [15] as a back-end for proof obligations. A large part of the effort concerns the user interface: the generated proof obligations are associated to the JAVA source code and presented using JAVA-like notations. The same applies to JIVE: the user interface permits to apply the Hoare logic rules by hand, and proof obligations are generated for PVS or HOL. We should point out that JIVE has a specification language which differs syntactically from JML. In LOOP, the semantics of JML-annotated JAVA programs is translated into functional PVS expressions which represent the denotational semantics of the program. Properties of these programs can be established using PVS tactics. Special theorems have been developed to simulate Hoare logic reasoning on JAVA programs, amongst which the computation of weakest pre-conditions [16].

Our approach. The originality of our work lies in our methodology: we proceed by translation of the JAVA program into the WHY input language [17,18]. The WHY tool [19] is a completely stand-alone tool. It is aimed at producing proof obligations for programs written in its own language especially designed for program certification. Moreover, it relies on an original method based on a functional interpretation using a static analysis of effects and monads, and a weakest pre-conditions calculus. The WHY input language is an ML-like minimal language, with very limited imperative features: references and exceptions. Hence, translating JAVA into WHY is a non-trivial task, we describe it in the remaining of this paper. As soon as this translation task has ended, the generation of proof obligations, including calculus of weakest preconditions, is performed by the WHY tool. The WHY tool has a multi-prover output [18]. Since we use its COQ output, the generated proof obligations are supposed to be proved interactively by the user, using the COQ proof assistant [20].

Overview. In Section 2, we explain our approach from the user's point of view. We use Dijkstra's classical Dutch National Flag example [21,22] to illustrate our method all along the paper. Then we go into details on how it works. In Section 3, we give an overview of the WHY tool and its input language, which is the target of our translation. In Section 4, we describe our model of JAVA, and how JAVA expressions and statements are translated into the WHY input language. In Section 5, we show how method calls and method definitions, including their JML specifications, are translated. Section 6 is devoted to translation of *pure* methods in the sense of JML. In Section 7, we consider statements generating abrupt termination: `break`, `continue`, and exceptions. All of them are translated into the exception constructs of WHY, allowing a uniform handling of proof obligations generation and weakest preconditions rules. We conclude in Section 8 with a discussion on unsupported features and future work.

2 General Architecture of the Approach

The Dutch National Flag problem is the following: given an array of colored (blue/white/red) elements, rearrange them so that blue elements occur first, then white ones, then red ones. To keep things short, we consider we have an array of such colors. In JAVA, we may code these colors by:

```
public class Flag {

public static final int BLUE = 1, WHITE = 2, RED = 3;

//@ public normal_behavior
//@   ensures \result <==> (i == BLUE || i == WHITE || i == RED);
public static /*@ pure @*/ boolean isColor(int i);
```

We do not give any body to `isColor`. In fact, we are going to prove our algorithm without using any knowledge of it, but with knowledge of its specification only. Moreover, we annotate `isColor` as *pure*, meaning that it has no side-effect, and as a consequence that it may be used in other specifications.

Our algorithm works on an array of colors, so we introduce an instance variable for this array, together with an invariant to specify that it is made of valid colors:

```
public int t[];
//@ public invariant  t != null &&
//@    (\forall int k; 0 <= k && k < t.length; isColor(t[k]));
```

Then we want to specify a method `flag` supposed to solve the problem. For this purpose, it is simpler to introduce a predicate `isMonochrome(i,j,c)` which specifies that some part `t[i..j-1]` of our array is of the same color `c`:

```
/*@ public normal_behavior
  @  requires 0 <= i && i <= j && j < t.length;
  @  ensures \result <==> (\forall int k; i <= k && k < j; t[k] == c);
  @*/
public /*@ pure @*/ boolean isMonochrome(int i, int j, int c);
```

The specification of our main method is then:

```
/*@ public normal_behavior
  @  assignable t[*];
  @  ensures
  @   (\exists int b,r; isMonochrome(0,b,BLUE) &&
  @    isMonochrome(b,r,WHITE) && isMonochrome(r,t.length,RED));
  @*/
public void flag()
```

Remark that this specification is uncomplete: we should also specify that the numbers of occurrences of each color are the same at the beginning and at the end, but we did not add this to keep things short.

To provide a body to the `flag` method, it is nice to have a method `swap` for permuting two elements of array `t`. Here is its specification:

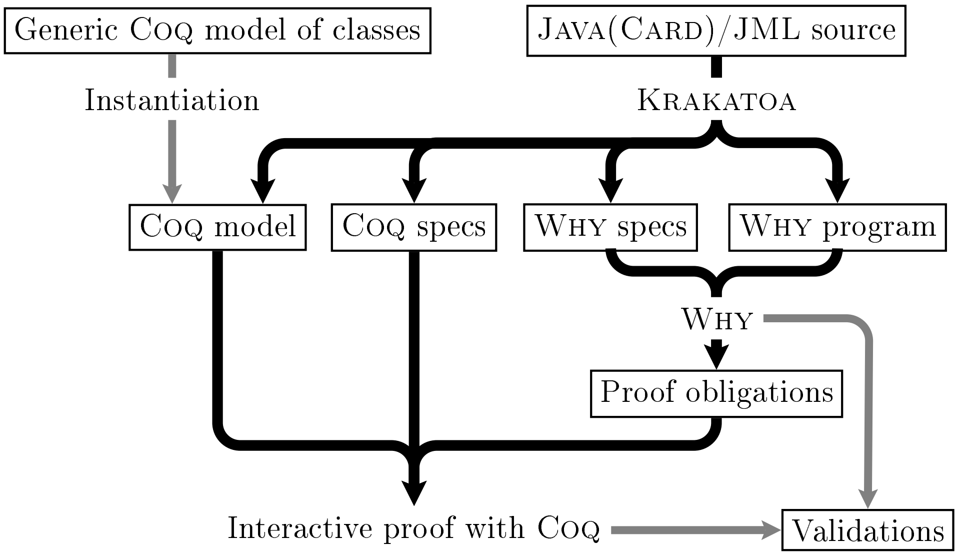

Fig. 1. Our methodology

```
/*@ public normal_behavior
  @  requires   0 <= i && i < t.length && 0 <= j && j < t.length;
  @  assignable t[i],t[j];
  @  ensures    t[i] == \old(t[j]) && t[j] == \old(t[i]);
  @*/
public void swap(int i, int j);
```

and here is the code we consider for `flag`:

```
public void flag() {
 int b = 0, i = 0, r = t.length;
 /*@ loop_invariant
   @  (\forall int k; 0 <= k && k < t.length; isColor(t[k])) &&
   @  0 <= b && b <= i && i <= r && r <= t.length &&
   @  isMonochrome(0,b,BLUE) && isMonochrome(b,i,WHITE) &&
   @  isMonochrome(r,t.length,RED);
   @ decreases r - i;
   @*/
 while (i < r) { switch (t[i]) {
  case BLUE:  swap(b, i); b++; i++; break;
  case WHITE: i++; break;
  case RED:   r--; swap(r, i); break;
  }}}
```

Our goal is to prove that method `flag` meets its JML specification.

The Krakatoa approach for performing such a verification is schematized on Figure 1. The generic Coq model of classes is a model of Java objects and memory heap, which is independent of any particular program, but parameterized by the sets of class and field names. In this model, we define in particular the type of JAVA values (primitive ones and references, i.e. objects and arrays), and the type of the memory heap. Given a JAVA program, and a particular method m in it, we generate several input files for WHY and COQ.

In a first step, KRAKATOA applies the so-called JML desugaring [23] to put each method specification in a simple form, with exactly one `requires`, one `assignable`, one `ensures` and one `signals` clause (i.e. clause specifying the behavior in case of an exception thrown). Class invariants are also put together to form a single assertion. In a second step, KRAKATOA performs a static analysis of *effects* which tells for each method whether: it reads and/or writes the memory heap, it reads and/or writes x for each static variable x of the program, and whether it may throw an exception (without distinction between types of exceptions, see Section 7) or not. These effects are approximated enough to be computable statically by a quite simple algorithm (although not completely trivial since it amounts to computing a fix-point). We shall point out that `NullPointerException` and `ArrayIndexOutOfBoundsException` are treated differently, as explained below.

The analysis of effects tells us that `isColor` does not read nor write the heap, `isMonochrome` reads the heap but does not write it, and `swap` and `flag` both read and write the heap. None of them accesses a static variable, since no such variable exists, and finally none can throw an exception. There is an important design decision to note here: KRAKATOA does not allow `NullPointerException` nor `ArrayIndexOutOfBounds` to be thrown. Instead, proof obligations which ensure that every access will be on a non-null value, and moreover that every array access will be inside the array's bounds will be generated. We explain more precisely how it is done in Section 4.

The third step is to generate WHY specification files for the program (one file for each class). This includes declarations of global variables (in the example it is only the memory heap, but in general there will be also one variable for each static field), declaration for each method of a WHY *parameter* function (i.e. without body) whose type is a translation of its specification, and declaration of a WHY predicate for each pure method. This process is detailed in Section 5. On the Dutch Flag example we get declarations of predicates `Flag_isColor`, `Flag_isMonochrome` and `Flag_invariant`, as well as an parameter function `Flag_swap_parameter`.

The fourth step is to generate COQ files containing definitions of predicates and functions corresponding to pure methods and class invariants. For the Dutch Flag example, we generate COQ definitions for predicates `Flag_isColor`, `Flag_isMonochrome` and `Flag_invariant`, automatically translated from their respective JML specifications. Details on how JML specifications are translated are given in Section 6.

The fifth step is to generate a WHY file for each method we want to verify. This file contains an annotated WHY program translated from its JML specification and JAVA code automatically. For the Dutch Flag example, we get a WHY function `Flag_flag` whose body is the actual translation of the body of method `flag`.

The work of the KRAKATOA tool ends at this point. The next step is to run the WHY tool on all generated WHY files, resulting, for each method m in a class C, in a COQ file containing the proof obligations for that method.

For the Dutch Flag example, ten proof obligations are generated. These have to be proved by the user with the COQ proof assistant. So as to help the user in his or her task, we provide some tactics which deal with operations defined in the COQ model. In particular, some tactics automatically apply lemmas regarding accesses to updated objects. On the Dutch Flag example, 121 lines of COQ tactics are needed to prove the proof obligations, the most difficult one (49 lines) corresponds to the preservation of the loop invariant. The total development amounts to 2300 lines of COQ source, but only the 121 lines of tactics have to be written by hand.

As soon as the proof obligations have been completed, we have proved that the initial method meets its specification. To provide a visible certificate of soundness of the approach, the WHY tool also provides a *validation* for each method. This validation is a functional translation of the original method m, but with an enriched type which reflects the JML specification of m. Using the program-as-proof paradigm of type theory, one may view a program with input x, output y, precondition P and post-condition Q as a constructive proof of the formula $\forall x, P(x) \Rightarrow \exists y, Q(x, y)$. In our case, the validation for a method m is a COQ function whose type has the following shape:

$$
\begin{array}{l}
\forall\langle\text{input memory heap}\rangle,\ \forall\langle\text{parameters of } m\rangle, \\
\quad \langle\text{preconditions on heap and parameters}\rangle \Rightarrow \\
\quad\quad \exists\langle\text{output memory heap}\rangle, \texttt{result}, \\
\quad\quad\quad \langle\text{post-conditions on input and output heaps, parameters and } \texttt{result}\rangle
\end{array}
$$

where pre-conditions include the `requires` of m, the class invariants and well-typedness of the parameters; and post-conditions include the `ensures` and/or `signals` clause, again class invariants, and a relation between input and output heaps corresponding to the `assignable` clause. Thus, a correct type-checking of the COQ validation gives a certificate of soundness to method m. Further note that some of the proof obligations are proved internally by WHY with full automation, and the validation provides a certification of their soundness also. Both the WHY translation of the JAVA program and the validation can be executed in order to test the adequation of the translation with reference to subtleties in JAVACARD semantics.

On the Dutch flag example, the Coq compilation (i.e. type-checking) of all files (i.e. the model files, the proof obligations file with proofs completed by the user, and the validations) needs 65s on a machine with a 1GHz processor.

3 Presentation of the Why tool

This tool is aimed at producing proof obligations for several provers [18], including Coq. The Why core language is a typed functional language à la ML, with additional imperative features and ML-like exception constructs. We present here the abstract syntax for the part of the language we need. The full concrete syntax can be found in the Why manual [19].

3.1 The Why *core language*

There are basic types: `int`, `bool`, and `unit` with constant `void`. As usual in functional languages, there is no syntactic distinction between statements and expressions, since statements can be seen as expressions of type `unit`.

Expressions are made of:

- constants of basic types
- primitive operations (unary and binary) applied to expressions
- variables
- local bindings: `let` $var = e_1$ `in` e_2
- conditionals: `if` e_1 `then` e_2 `else` e_3
- applications, denoted $(e\ e_1\ \ldots\ e_2)$ (Curried form)
- anonymous function definitions, denoted `fun` $(x_1 : t_1)\ \cdots\ (x_n : t_n) \rightarrow e$

The language of Why is mainly functional; it provides nevertheless some imperative features, namely

- local reference binding: `let` $var\ =$ `ref` e_1 `in` e_2
- reference access: `!`var
- assignment: var`:=`e
- blocks: `begin` $e_1; ...; e_n$ `end`
- *while* loops: `while` e_1 `do` e_2 `done`

The Why tool has an internal exception mechanism which allows:

- raising of an exception carrying a value: `raise` (Exc e) where exception Exc carrying a value of type τ must be declared by `exception` Exc `of` τ
- catching of exception: `try` e `with` $(E_1\ v_1)$ `->` e_1 `|` $(E_2\ v_2)$ `->` e_2 `|` $\cdots$ `end`

3.2 Annotations in Why

In Why, any expression e can be annotated, using the Hoare triple notation $\{pre\}\ e\ \{post\}$, or more generally, when this expression may raise exceptions,

$$\{pre\}\ e\ \{normalpost \mid E_1 \Rightarrow exc\_post_1 \mid \cdots \mid E_n \Rightarrow exc\_post_n\ \}$$

Additionally, *while* loops must be annotated as

`while` e `do` { `invariant` *inv* `variant` *var* } e `done`

in order to specify a loop invariant, as well as a *variant* which is supposed to decrease at each loop iteration. Unlike JML, annotations *pre*, *post* and *inv* are not boolean expressions of WHY, but formulas of first-order logic. Their variables are the program variables, and predicate and function symbols are assumed to be defined in the target proof assistant and declared using the `logic` keyword. Moreover, in *post*, one may use special keyword `result` to denote the value of the expression, and v`@` to denote the value of v before evaluation of the expression.

3.3 WHY *parameter declarations*

In a WHY program, any called sub-program must be either defined earlier, or declared as a *parameter* (which corresponds more or less to a abstract method). A parameter declaration has the form

`parameter` *prog_id* : *type with effects*

The effects tell which of the references this program may access or modify, and whether it may throw an exception or not. The general form of a type with effects is:

$$\tau_1 \texttt{->} \cdots \tau_n \texttt{->} \{pre\}\ \tau_{result}\ \texttt{reads}\ r_1,\ldots\ \texttt{writes}\ w_1,\ldots\ \texttt{raises}\ E_1,\ldots$$
$$\{normal\ post \mid E_1 \Rightarrow exc\_post_1 \mid \ldots\}$$

where the r_i are the references possibly accessed, w_i the references possibly modified, and E_i the exceptions possibly raised. Keyword `result` may appear in normal, resp. exceptional, post-condition to refer to the returned value, resp. the value carried by the thrown exception.

4 The WHY model of objects, and translation of JAVA statements

We now go into details of our approach. We first describe our model of JAVA, and how JAVA expressions and statements are translated into the WHY input language.

Values. The structure of the JAVA objects is defined in WHY via abstract data types. The type of field and class names are respectively `fieldId` and `classId`. We introduce the abstract type `javaType` for JAVA types with the expected constructors :

```
IntType : javaType
```

```
BoolType :  javaType
ArrayType : javaType -> javaType
ClassType : classId -> javaType
```

We define a type name `value` in WHY to denote JAVA values, primitive ones (integers, booleans,...) and references (objects and arrays). The references are represented as addresses referring to a global heap of abstract type `store`. There is a type `cell` which is either an integer for access in an array or a field name for accessing an object. We introduce also a primitive function which returns the length of an array. This is given by the WHY signature:

```
Null : value
Field : fieldId -> cell
Index : int -> cell
arraylength : value -> int
instanceof : value -> javaType -> bool
```

Heap accesses and updates. We have to specify the abstract operations allowed on the store: mainly accesses and updates regarding fields or array cells.

```
access : store -> value -> cell -> value
update : store -> value -> cell -> value -> store
```

The `access` and `update` operations are valid only when some extra conditions are satisfied. The cell should be meaningful for the value: if the value is an object, the cell must a field of its class, or if the value is an array, the cell must be a non-negative integer less than the length of the array. In case of an update the stored value should be allocated in the heap (or a primitive value) and of appropriate type with reference to the cell which is updated.

In order to achieve this, we introduce two predicates in WHY which express under which conditions `access` or `update` can be performed.

```
is_valid_cell : value, cell -> prop
is_valid_update : store, value, cell, value -> prop
```

We use these functions to translate any access or update of the heap. Firstly, the global memory is defined in WHY as a modifiable variable of type `store`:

```
parameter heap: store ref
```

Then, any field access $e.f$ is translated into the Hoare triple

let tmp $= \widetilde{e}$ in
$\{$ `is_valid_cell(tmp, Field(f))` $\}$ `(access !heap tmp (Field f))` $\{ \}$

where $\widetilde{e}$ is the translation of e. Note that `heap` must be explicitly dereferenced with `!` since it is a modifiable WHY variable. The `tmp` local variable is only necessary when there are possible side-effects in expression. Analogously, array access e`[`e_1`]` is translated into

$$\texttt{let tmp} = \widetilde{e} \texttt{ in let tmpi} = \widetilde{e_1} \texttt{ in}$$
$$\{ \texttt{is\_valid\_cell}(\texttt{tmp}, \texttt{Index}(\texttt{tmpi})) \} \ (\texttt{access !heap tmp} \ (\texttt{Index tmpi})) \ \{ \}$$

Thus, in both field and array accesses, the `access` function is protected with a pre-condition which ensures that the access is valid. Null pointer dereferencing and out-of-bounds accesses are ruled out at this very point, as discussed in Section 2. See Section 8 for a discussion on how we could authorize them.

Field update e`.`f `=` e_1 is translated into

$$\texttt{let tmp} = \widetilde{e} \texttt{ in let tmpv} = \widetilde{e_1} \texttt{ in}$$
$$\{ \texttt{is\_valid\_update}(\texttt{heap}, \texttt{tmp}, \texttt{Field}(\texttt{f}), \texttt{tmpv})) \}$$
$$\texttt{heap:=}(\texttt{update !heap tmp} \ (\texttt{Field f}) \ \texttt{tmpv}) \ \ \{ \}$$

and array update e`[`e_1`]` `=` e_2 is translated into

$$\texttt{let tmp} = \widetilde{e} \texttt{ in let tmpi} = \widetilde{e_1} \texttt{ in let tmpv} = \widetilde{e_2} \texttt{ in}$$
$$\{ \texttt{is\_valid\_update}(\texttt{heap}, \texttt{tmp}, \texttt{Index}(\texttt{tmpi}), \texttt{tmpv})) \}$$
$$\texttt{heap:=}(\texttt{update !heap tmp} \ (\texttt{Index tmpi}) \ \texttt{tmpv}) \ \ \{ \}$$

Interpretation in COQ. The WHY abstract data types which introduced must be realized in COQ. Hence we are lead to design a COQ *model* of JAVA. Indeed, this model is generic in the sense it is defined as a COQ functor which is parameterized by the set of classes and fields names, and It is instantiated automatically by KRAKATOA for each JAVA program. The parameter of this functor parameter must contain:

- an enumerated type `classId` of class names, with relation `extends`
- an enumerated type `fieldId` of field names
- additional basic functions and properties such as the function which gives the type of each field, the properties that `Object` class id always exists and has no field.

The generic model defines first a type `javaType` of JAVA types which are booleans, integers, classes or arrays. then the type `value` of JAVA values: primitive (booleans, integers), `null` and references (objects and arrays). A reference is built from an address in the heap and a `kind` which is either the name of a class for an object or a type and a length for an array.

Then the store is defined as a partial function from pairs of kinds and addresses to maps from cells to values. It is furthermore constrained to have certain well-formedness conditions. We say that a value is *alive* in a heap if it is a primitive value, or a null pointer or reference which is allocated in the heap. A heap will be said *well-formed* if for each value allocated in the heap and each cell valid for this value, then the value in the cell is alive and has the appropriate type. Basic operations on the store (`update`, and `store_new_ref` defined below for object creation) are built, in such a way that they preserve the well-formedness of stores. Several lemmas are then proved in the model, such as lemmas which state what do you get when you access a address in a store which just has been updated, the result depending whether you access the adress just updated or another one. These lemmas are of course needed for proving the obligations generated afterwards.

5 Method calls, handling class invariants and assignable clauses

5.1 Assignable clauses

To handle assignable clauses, we define an abstract type `mod_loc` corresponding to assignable sets of locations (a location is defined as a value and a cell). It is interpreted in Coq as a higher-order type (relation between addresses and cells). There is a Why `logic` declaration for each JML construction of assignable clause:

`nothing_loc : -> mod_loc`	(for `\nothing` in JML)
`everything_loc : -> mod_loc`	(for `\everything`)
`inter_loc : mod_loc,mod_loc -> mod_loc`	(for m_1, m_2)
`access_loc : value, cell -> mod_loc`	(for $v\,.\,f$ or $v[n]$)
`array_sub_loc : value, int, int -> mod_loc`	(for $v[i..j]$)
`allcells_loc : value -> mod_loc`	(for `\all_fields`(v))
`reachable_loc : store,value -> mod_loc`	(for `\reachable`(v))

In the Why declarations appear also the following relations:

```
well_formed : store, value, javaType  -> prop
modifiable : store, store, mod_loc -> prop
```

`well_formed`(h, v, t) means that v is a value alive (as defined in previous section) in heap h which has type t. `modifiable`(h_1, h_2, m) means that any value alive in h_1 is also alive in h_2 and that any location which does not belong to m (assignable locations) and is allocated in h_1 is left unchanged in h_2. Note that here we proceed as there were never garbage collection: indeed, we may assume that since the semantics of Java does not depend on whether there is garbage collection or not.

5.2 Method calls

Method call $e\,.\,m(e_1, \ldots, e_n)$ is translated into

let tmpv $= \widetilde{e}$ in let tmpv1 $= \widetilde{e_1}$ in ... let tmpv$n = \widetilde{e_n}$ in
(C_m_parameter tmpv tmpv1 $\cdots$ tmpv$n$)

Notice this does not take dynamic method call into account. See Section 8 for further comments on this issue.

5.3 Object creation and constructor calls

Object creation involves both an allocation on the heap, and a call to a constructor. For allocation, we introduce three new WHY parameters:

```
new_obj : store -> classId -> value
new_array : store -> int -> javaType -> value
store_new_ref : store -> value -> store
```

`new_obj` and `new_array` return a reference of the wanted type, not already allocated ; and `store_new_ref` adds this new reference in the given store, initializing its fields by default values, and returns the new store obtained.

Object creation `new` $C(e_1, \ldots, e_n)$ is then translated into

let krak_new = (new_obj !heap C) in
 heap := (store_new_ref !heap krak_new);
 (C_Ci_parameter krak_new $\widetilde{e_1} \cdots \widetilde{e_n}$);
 krak_new

where C_Ci_parameter is the user defined constructor, the index i is an optional number given to handle constructor overloading. Array creation is translated in a similar way.

5.4 Method and constructor declarations

For each method or constructor in a class C annotated with JML specification:

/*@ *requires* r;
  @ *assignable* a;
  @ *ensures* e;
  @*/
$\tau\ m(x_1 : \tau_1, \ldots, x_n : \tau_n)$;

a WHY parameter is generated:

parameter C_m_parameter :
 this:value -> $x_1 : \widetilde{\tau_1}$ -> $\cdots x_n : \widetilde{\tau_n}$ ->
 { $\widetilde{r}$ and this <> Null
   and well_formed(heap, this, C)
   and C_invariant(heap, this)

$\quad$ and well_formed(heap, x_1, τ_1)
$\quad$ and $\widetilde{\tau_1}$_invariant(heap, x_1) $\ldots$ and $\widetilde{\tau_n}$_invariant(heap, x_n) }
$\widetilde{\tau}$ reads *read* writes *write*
{ $\widetilde{e}$ and C_invariant(heap, this)
$\quad$ and $\widetilde{\tau_1}$_invariant(heap, x_1) $\ldots$ and $\widetilde{\tau_n}$_invariant(heap, x_n)
$\quad$ modifiable(heap@, heap, $\widetilde{a}$)}

where $\widetilde{\tau}$ = `value` for reference types, `int` for integer types, and `bool` for booleans. We should point out that on such case of primitive types, no `well_formed` nor invariant is generated. For *read* and *write* we use the effects computed in the second step, as described in Section 2.

On the Dutch Flag example Krakatoa generates the following Why specifications[1]:

```
logic Flag_invariant : store, value -> prop
parameter Flag_swap_parameter : this:value -> i:int -> j:int ->
    {     0 <= i and i < arraylength(access(heap, this, t))
      and 0 <= j and j < arraylength(access(heap, this, t))
      and this <> Null
      and well_formed(heap, this, Flag)
      and Flag_invariant(heap, this) }
    unit reads heap writes heap
    {     access(heap, access(heap, this, t), i) =
          access(heap@, access(heap@, this, t), j)
      and access(heap, access(heap, this, t),j) =
          access(heap@, access(heap@, this, t), i)
      and Flag_invariant(heap, this)
      and modifiable(heap@, heap,
          inter_loc(access_loc(access(heap@, this, t), i),
          access_loc(access(heap@, this, t), j))) }
```

and the following Coq definition which interprets the `Flag_invariant` predicate:

```
Definition Flag_invariant := [heap : store] [this : value]
  (~(access heap this t)=Null)
  /\ ((k:Z)((0<=k/\k<(arraylength (access heap this t))) ->
      (Flag_isColor (int_val (access heap (access heap this t) k))))).
```

5.5 Translating method or constructor bodies

For translation of a method (or a constructor) with body, we translate its JML specification the same way the specification of a method without body is translated. Its body is translated as shown in Section 4. For instance, method

[1] In order to improve readability, we have omitted constructors like `Field`, `Index` or `ClassType` which can easily be inferred from the context.

`flag` is translated into an annotated WHY program which is roughly 60 lines long. It is available, together with other example programs, on the KRAKATOA Web page [24].

There is an important issue regarding *assignable* clauses and loops: in addition to the loop invariant given by the user, it is usually mandatory to constrain the set of assignable locations inside the body of the loop. For this purpose, if the method is supposed to only assign a set m of locations, we add in the loop-invariant the clause `modifiable`(`heap@init`, `heap`, m) which means that the current heap differs from the original heap before calling the method only on the set m of locations.

6 Handling pure methods

Pure methods of JML, like `isColor` and `isMonochrome` in the Dutch Flag example, need an additional treatment, since we need to be able to use them in WHY annotations as well as in COQ proofs. Regarding their use in WHY annotations, we need to generate in the WHY specification file a declaration for them, i.e. what their types are; regarding COQ, in addition to their types, we need to generate a definition for them, from their JML specification. There is a difficulty here because in both WHY and COQ, like in classical first-order logic, there is a clear distinction between function symbols and predicate symbols. A pure method whose return type is not `boolean` will be naturally translated into a function. But when the return type is `boolean`, it is natural to translate it into a predicate. In practice, this is possible only if its post-condition has the form `\result <==>` *expr* where `\result` does not occur in *expr*. These considerations are rather technical, but they are very important for the user: pure methods will appear frequently in proof obligations, and it is important that they are easy to manipulate in interactive proving.

On the Dutch Flag example, we get the following WHY declarations:

```
logic Flag_isColor : int -> prop
logic Flag_isMonochrome : store, value, int, int, int -> prop
```

and we get the following COQ definitions:

```
Definition Flag_isColor := [i : Z]
  (i=Flag_BLUE \/ i=Flag_WHITE \/ i=Flag_RED).
Definition Flag_isMonochrome := [heap : store] [this : value]
  [i : Z] [j : Z] [c : Z] (k:Z)((i<=k/\k<j) ->
    (int_val (access heap (access heap this t) k)) = c).
```

As discussed in [25], pure methods may also have preconditions and may throw exceptions. The semantics of calls to them, in an assertion, in a invalid case w.r.t. the precondition, or in an exceptional case, is to return a nonspecified value. This means that for our translation, we can safely forgot the precondition and the exceptional case.

7 Abrupt termination and exceptions

Abrupt termination occurs whenever a statement is left because of `break`, `continue`, `return`, or a thrown exception. Each of these cases is translated into WHY exceptions. JAVA exceptions are handled via a unique WHY exception declared as

```
exception Exception of value
```

carrying any JAVA value (the JAVA exception object). Exception throwing `throw` e is translated into `raise` (`Exception` $\widetilde{e}$) and a *try-catch* construct

try s
catch (Exc_1 v_1) s_1
$\vdots$
catch (Exc_n v_n) s_1

is translated into:

try $\widetilde{s}$
with (Exception tmp) ->
    if (instanceof tmp Exc_1) then $\widetilde{s_1}$
    $\vdots$
    else if (instanceof tmp Exc_n) then $\widetilde{s_n}$
    else raise (Exception tmp)

Signals clauses are desugared: multiple signals clauses are gathered into a single one [23], and an expression stating that an exception must be an instance of one of the declared exceptional types is added. This is easily translated using WHY exceptions constructs and no special treatment is needed.

Other cases of abrupt termination are translated using additional exceptions `Continue`, `Break`, `Return` and `Switch` we declare to WHY. The instruction `return` e; becomes `raise` (`Return` $\widetilde{e}$) and a method body *body* which may return abruptly will be translated into: `try` $\widetilde{body}$ `with` (`Return` r) $\rightarrow r$. The instruction `break` lab; becomes `raise` (`Break` lab), `continue` lab; becomes `raise` (`Continue` lab), and a labelled *while* loop is translated into:

try while $\widetilde{e}$ do
    try $\widetilde{s}$
    with (Continue x) $\rightarrow$ if $x \neq lab$ then raise (Continue x) end
  done
with (Break x) $\rightarrow$if $x \neq lab$ then raise (Break x) end

Similar translation schemes exist for *switch* and *try-catch-finally* statements.

For instance, proving the following method `withdraw` in a classical electronic purse application:

```
class Purse {
 int balance;
 //@ public invariant balance >= 0;

 /*@ public behavior
   @  requires s >= 0;
   @  assignable balance;
   @  ensures s <= \old(balance) && balance == \old(balance) - s;
   @  signals (NoCreditException) s > \old(balance) &&
   @   balance == \old(balance);
   @*/
 public void withdraw(int s) throws NoCreditException {
    if (balance >= s) { balance -= s; }
    else { throw new NoCreditException(); }
 }
```

leads to a WHY program which is 40 lines long, and for which five proof obligations are needed. It is important to notice that WHY's proof generation mechanism automatically split between normal case and exceptional cases: there is no need for any special exception handling when proving those obligations interactively. For example, the last proof obligation is the exceptional post-condition and invariant kept in case the exceptional branch of the `if` is executed.

8 Conclusion and further work

In this paper, we have presented the design choices of our tool KRAKATOA. In particular any method is interpreted as a terminating function, which does not raise dynamic errors `NullPointerException` or `ArrayIndexOutOfBounds`. In order to ensure that, extra properties have to be proved before accessing memory cells. Note that this design choice is not due to a limitation of our method based on translation into WHY: indeed we may translate a field access $e\,.\,f$ into

$$\texttt{let tmp} = \widetilde{e}\ \texttt{in if tmp} = \texttt{Null then raise NullPointerException}$$
$$\texttt{else}\ \{\ \texttt{is\_valid\_cell}(\texttt{tmp}, \texttt{Field(f)})\ \}\ (\texttt{access !heap tmp (Field f)})\ \{\ \}$$

but this would result in more complicated translated programs, thus probably more complicated proof obligations. We made the assumption the well-designed JAVACARD programs will not rely on `NullPointerException` to be thrown and catch afterwards.

Our methodology is based on the generic Why tool for program certification which provides a modular and readable interpretation of the Java program into an intermediate language using references and exceptions. The Why tool generates automatically both the proof obligations and a functional certified version of the program.

Our tool has been successfully applied on simple Java programs like the ones presented in this paper. We are currently verifying security properties (e.g. that particular exceptions cannot be raised) of a real JavaCard applet. This experiment shows that the tool supports larger programs and specifications easily, the main problem being to prove the proof obligations. This is still a time-consuming effort due to the very low level of automation that is currently provided in Coq. However the numerous proof obligations remain human readable, and that was one of our concerns when we designed the model.

8.1 Unsupported features

There are a few important features which are not yet supported by Krakatoa and which we plan to add in a very near future. As mentioned before, we do not yet prove termination of recursive methods, indeed there are some issues with the semantics of *measured_by* JML clauses, in particular with mutually recursive methods.

We do not yet consider possible overflow of integer operations. We plan to continue to map Java numerical values into Why integers, adding preconditions for each numerical type and assertions before arithmetic operations to prevent overflow. An alternative model used in LOOP could be used to model the semantic of Java operations precisely.

There is no difficulty for handling overloading of methods, because the static typing phase of Java is able to resolve such ambiguities, and method names can be made unique. On the other hand, we do not have explicit support for redefinition (or overriding, that is when a method is defined with the same name and the same parameter types as an existing one in a superclass, thus requiring dynamic method call). We could handle this case by taking for the specification of the method a combination of specifications of the redefined methods depending on the dynamic type of `this`. This is not yet implemented. A methodological point is whether we constrain the redefined methods to satisfy the specification of the method it overrides also, that is the so-called *subclassing contract* discussed in [26].

8.2 Alternative models of values

We developed also a functional semantics of Java where values were represented as a co-inductive type in Coq. Our aim was to translate each Java variable as a Why variable and avoid the use of a global heap. Of course the `update` operation is, in that case, only valid when no aliasing occurs between two different variables. The idea was to have sufficient syntactic conditions which ensure that no aliasing occurs. However, because we wanted to address quickly real JavaCard applications, and because the APDU buffer is usually aliased, we decided to move to a global memory model. We still believe that proofs will be made much easier if the model may associate different variables to different values and internalize the separation in the memory. In particular, we plan to experiment with a more local representation of the heap where there is a different store for each field [27,28]. Experimenting with this alternative representation requires only to modify the translation from Java programs to Why.

Acknowledgments We would like to thank Evelyne Contejean, Jean Duprat and Jean-Christophe Filliâtre for their participation in the design of the Krakatoa methodology, and the anonymous referees for their fruitful comments on the first version of this paper.

References

[1] G. T. Leavens, A. L. Baker, C. Ruby, Preliminary design of JML: A behavioral interface specification language for Java, Tech. Rep. 98-06i, Iowa State University (2000).

[2] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, JML Reference Manual, draft (Apr. 2003).

[3] Sun Microsystems, The JavaCard^{TM} application programming interface (API), `http://java.sun.com/products/javacard/`.

[4] Z. Chen, JavaCard^{TM} Technology for Smart Cards, The Java Series, Addison-Wesley, 2000.

[5] E. Poll, J. van den Berg, B. Jacobs, Formal specification of the JavaCard^{TM} API in JML: the APDU class, Computer Networks 36 (4) (2001) 407–421.

[6] H. Meijer, E. Poll, Towards a full formal specification of the java card, in: I. Attali, T. Jensen (Eds.), Smart Card Programming and Security, no. 2140 in Lecture Notes in Computer Science, Springer-Verlag, 2001, pp. 165–178.

[7] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, E. Poll, An overview of JML tools and applications, Tech. Rep. NIII-R0309, Dept. of Computer Science, University of Nijmegen (2003).

[8] D. L. Detlefs, K. R. M. Leino, G. Nelson, J. B. Saxe, Extended static checking, Tech. Rep. 159, Compaq Systems Research Center, see also `http://research.compaq.com/SRC/esc/` (Dec. 1998).

[9] B. Jacobs, Loop project, `http://www.cs.kun.nl/~bart/LOOP`.

[10] J. van den Berg, M. Huisman, B. Jacobs, E. Poll, A type-theoretic memory model for verification of sequential Java programs, in: D. Bert, C. Choppy,

P. Mosses (Eds.), Recent Trends in Algebraic Development Techniques, Vol. 1827 of Lecture Notes in Computer Science, Springer-Verlag, 2000, pp. 1–21.

[11] J. van den Berg, B. Jacobs, The LOOP compiler for Java and JML, in: T. Margaria, W. Yi (Eds.), Tools and Algorithms for the Construction and Analysis of Software, Vol. 2031 of Lecture Notes in Computer Science, Springer-Verlag, 2001, pp. 299–312.

[12] J. Meyer, P. Müller, A. Poetzsch-Heffter, The JIVE system, implementation description available from `http://www.informatik.fernuni-hagen.de/pi5/publications.html` (2000).

[13] L. Burdy, JACK: Java Applet Correctness Kit, Gemplus Developer Conference (2002).

[14] F. S. de Boer, A WP-calculus for OO, in: Foundations of Software Science and Computation Structures, Vol. 1578 of Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 135–149.

[15] J.-R. Abrial, The B-Book, assigning programs to meaning, Cambridge University Press, 1996.

[16] B. Jacobs, Weakest precondition reasoning for Java programs with JML annotations, available from `http://www.cs.kun.nl/~bart/` (2002).

[17] J.-C. Filliâtre, Verification of Non-Functional Programs using Interpretations in Type Theory, Journal of Functional Programming 13 (4) (2003) 709–745.

[18] J.-C. Filliâtre, Why: a multi-language multi-prover verification tool, Research Report 1366, LRI, Université Paris Sud, `http://www.lri.fr/~filliatr/ftp/publis/why-tool.ps.gz` (March 2003).

[19] J.-C. Filliâtre, The Why certification tool, `http://why.lri.fr/`.

[20] The Coq Development Team, The Coq Proof Assistant Reference Manual – Version V7.4, `http://coq.inria.fr` (Jan. 2003).

[21] E. W. Dijkstra, A discipline of programming, Series in Automatic Computation, Prentice Hall Int., 1976.

[22] J. R. Bitner, An asymptotically optimal algorithm for the Dutch national flag problem, SIAM Journal on Computing 11 (2) (1982) 243–262.

[23] A. D. Raghavan, G. T. Leavens, Desugaring JML method specifications, Tech. Rep. 00-03a, Iowa State University (2000).

[24] C. Marché, C. Paulin, X. Urbain, The KRAKATOA proof tool, `http://krakatoa.lri.fr/` (2002).

[25] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, D. R. Cok, How the design of JML accomodates both runtime assertion checking and formal verification, Tech. Report 03-04, Iowa State University (Mar. 2003).

[26] C. Ruby, G. T. Leavens, Safely creating correct subclasses without seeing superclass code, Tech. Report 00-05d, Iowa State University (Jul. 2000).

[27] R. Bornat, Proving pointer programs in Hoare logic, in: Mathematics of Program Construction, 2000, pp. 102–126.

[28] F. Mehta, T. Nipkow, Proving pointer programs in higher-order logic, in: F. Baader (Ed.), 19th Conference on Automated Deduction, Lecture Notes in Computer Science, Springer-Verlag, 2003.