Reasoning about Java Programs with Aliasing and Frame Conditions

Claude Marché, Christine Paulin-Mohring

Reasoning about Java Programs with Aliasing and Frame Conditions

Claude Marché and Christine Paulin-Mohring

PCRI: LRI (Université Paris-Sud & CNRS UMR 8623), INRIA Futurs
Bât. 490, Université Paris-Sud
91405 Orsay Cedex, France

Abstract. Several tools exist for reasoning about JAVA programs annotated with JML specifications. A main issue is to deal with possible *aliasing* between objects and to handle correctly the *frame conditions* limiting the part of memory that a method is allowed to modify. Tools designed for automatic use (like ESC/JAVA) are not complete and even not necessarily correct. On the other side, tools which offer a full modeling of the program require a heavy user interaction for discharging proof obligations. In this paper, we present the modeling of JAVA programs used in the KRAKATOA tool, which generates proof obligations expressed in a logic language suitable for both automatic and interactive reasoning. Using the SIMPLIFY automatic theorem prover, we are able to establish automatically more properties than static analysis tools, with a method which is guaranteed to be sound, assuming only the correctness of our logical interpretation of programs and specifications.

1 Introduction

KRAKATOA [16] is a prototype tool for verifying that a JAVA program meets its JML specification (JML stands for Java Modeling Language [14, 15]). It is built on top of the WHY tool [11], which generates proof obligations from annotated programs written in a basic ad-hoc programming language with higher-order functions, references, exceptions, and a simple modular specification language [10]. These proof obligations can be generated for various interactive proof assistants and automatic theorem provers.

KRAKATOA expresses the operational semantics of a JAVA program by producing a translation into the WHY programming language as well as translating the JML specification into logical assertions. KRAKATOA needs also to provide a theory corresponding to the program which expresses the representation of the memory, and the dynamic typing information. The memory modeling of the first version of KRAKATOA was built on a unique heap where the objects and arrays

[0] Research partly supported by the EU IST project IST-2000-26328-VERIFICARD (`www.verificard.org`), the GECCOO project of the "ACI Sécurité Informatique" (`geccoo.lri.fr`) and the EU coordinated action TYPES (`www.cs.chalmers.se/Cs/Research/Logic/Types/`)

were stored, and the theory was only generated for the Coq proof assistant [21]. The tool was successfully used for proving small examples like Dijkstra's Dutch Flag algorithm or basic properties of a method in a JavaCard applet provided by the Schlumberger company for the IST VerifiCard project [8, 13]. The modular proof architecture appeared well-suited for dealing with large programs, however the manual work for proving proof obligations was complicated due to the lack of automation, and the too naive memory representation which was not taking enough static typing information into account.

To cope with this problem, we changed the modeling and adopted a more local representation of the memory. This alternative approach, already present in early work on general pointer programs by Burstall [7] is emphasized by Bornat [4]. It amounts to associate to each structure field a map from adresses to value, access or modification of the field of some structure being just interpreted as the corresponding access or update of the map at the index corresponding to the address of the structure. This approach works perfectly well with Java object fields, because the corresponding cell can only be accessed using the field name, but we also had to extend the approach to Java arrays, and also to support new memory allocation. This was not a trivial task because the Krakatoa tool has to perform an accurate static analysis of the program in order to deal properly with frame conditions (JML `modifiable` clauses, specifying the only part of the memory that can be changed by a method). Another extension was to provide a first-order theory for the logical aspects of the programs in order to use automatic provers such as Simplify [19] for solving proof obligations. Building a first-order theory for Java programs was also non-trivial and we used our higher-order Coq modeling for validating the axioms in order to avoid building an inconsistent theory that will trivially solve all proof obligations. The new version of Krakatoa together with Simplify can automatically check simple properties of programs (no null object dereferencing, no out-of-bounds array access, etc.): it validates proof obligations or provides counter-examples. Unlike ESC/Java which does not properly check frame properties leading to accept programs which are wrong, the Krakatoa approach never gives wrong positive answers.

Section 2 gives an overview of the languages and notations, mainly JML and the specification language of the Why tool. Section 3 explains our modeling of memory states and describes the background theory needed to solve proof obligations. Section 4 explains how Java programs are interpreted in our model. Section 5 studies in more details how the frame conditions are handled. Section 6 illustrates our method on some examples. We conclude in Sect. 7, with comparison to related works.

2 Preliminaries

2.1 Considered Fragment of JML

In JML, Java programs can be annotated using a special class of comments. Logical formulas are written as Java boolean expressions using only *pure* methods

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

    /*@ normal_behavior
      @  requires s >= 0;
      @  modifiable balance;
      @  ensures balance == \old(balance)+s; */
    public void credit(int s) { balance += s; }

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

Fig. 1. JML specification of a simplified electronic purse

(i.e. without side effects). Furthermore some special operators such as `\forall`, `\exists` are introduced. In a post-condition, the construction `\result` refers to method's returned value, and `\old`(e) is the value of expression e before the execution of the method.

In this paper, we focus on the most important features of JML: declarations of class invariants and method annotations describing their functional behavior: a `requires` (resp. `ensures`) clause gives the pre-condition (resp. the post-condition), and a `modifiable` clause, also called *frame condition*, specifies the set of memory locations where the method can write. A `signals` clause similar to `ensures` specifies which exception can be raised and which properties are true in that case. This is illustrated on Figure 1 which introduces a simple class `Purse` with a field `balance` which should be non-negative (invariant), a method `credit` (resp. `withdraw`) which adds (resp. removes) money to the purse.

Krakatoa also supports other essential JML constructs such as loop invariants. Regarding exceptions, it is noticeable that the Krakatoa approach, whatever the JML specification is, inserts pre-conditions to access operations which prevent to raise an exception in the class `RunTimeException` (mainly `NullPointerException` or `ArrayIndexOutOfBounds`). Also, we interpret `byte`, `short`, etc. into unbounded integer arithmetic (arithmetic overflow could be forbidden by insertion of suitable preconditions on integer operations [16]).

2.2 Interpretation into the Why Tool

Why's core language includes higher-order functions, references and exceptions but rejects any variable aliasing. Programs can be annotated using pre, post-

conditions, loop invariants and intermediate assertions. Logical formulas are written in a typed (sorts, possibly parametric) first-order logic, with built-in equality and integer arithmetic. WHY performs an *effect analysis* (see below) on programs, a classical weakest-precondition computation, and generates proof obligations for various provers.

WHY has a primitive notion of exceptions which is integrated in its weakest pre-condition calculus. It is used for the interpretation of `break` and `continue` statements or the JAVA exception mechanism and JML exceptional behavior (see [16]). WHY has a modular approach: new logic functions or predicate symbols can be introduced and freely used in the specification part, and sub-programs can also be introduced abstractly, giving just a full specification: type of input variables and result, pre- and post-conditions, as well as its *effects*, that is giving which of the global references can be read and/or written by the sub-program.

In order to translate a JAVA program annotated with JML into WHY, one needs to proceed the following way:

1. Find an appropriate modeling of JAVA memory states using global WHY variables which will never be aliased ;
2. Translate JAVA constructs into WHY statements, with assignments over the global variables defined before ;
3. Translate JML formulas into WHY predicates, over those variables.

Our JAVA modeling contains a generic part which is the same for all JAVA programs and a specific part which depends on the particular class hierarchy. The generic part introduces a global variable `alloc` which keeps track of allocated objects, it defines the set of values, memory segments (mapping addresses to values), JAVA types, and operations such as access or update of memory as well as logical relations like to be an instance of a class (`instanceof`), or to preserve part of the memory (`modifiable`). These notions will be formally introduced in the next sections.

The specific part introduces constants for the different classes (`Purse` in the example) and global variables for the fields (`balance` in the example), it also defines a predicate for each class invariant.

Because JAVA methods in a class can be mutually recursive, we first give a specification of all the methods as abstract WHY sub-programs with specifications. This abstract view is used to interpret method calls, and suffices to ensure partial correctness of programs.

The general form of a WHY program specification f is the following :

parameter f: x_1:$\mathrm{type}_1 \rightarrow \cdots \rightarrow \mathrm{x}_n$:$\mathrm{type}_n \rightarrow \{$ *Precondition* $\}$ *return_type*
reads *input_vars* **writes** *output_vars*
$\{$ *Postcondition* $\mid$ $\mathit{exception}_1 \Rightarrow \mathit{condition}_1 \ \ldots \mathit{exception}_n \Rightarrow \mathit{condition}_n$ $\}$

Figure 2 shows the WHY abstract declaration of the sub-program corresponding to the `Purse.credit` method of Figure 1. It has two arguments (the object *this* and the integer parameter s) which are supposed to satisfy the pre-condition (first formula between curly braces). There is no result (output type *unit*), it

```
parameter Purse_credit : this : value → s : int →
  { s ≥ 0 ∧ this ≠ Null ∧ (instanceof alloc this (ClassType Purse))
    ∧(Purse_invariant balance this)}
  unit reads balance, alloc writes balance
  { (access balance this) = (access balance@ this) + s ∧ (Purse_invariant balance this)
    ∧(modifiable alloc@ balance@ balance (value_loc this)) }
```

Fig. 2. WHY interpretation of `Purse.credit`

accesses two global variables (*balance* and *alloc*) and writes one (*balance*). The post-condition (second pair of curly braces) uses the notations *balance*@ and *alloc*@ to denote the value of these variables before the method application.

The WHY language has a formally defined semantics [10], and the fact that the generated proof obligations are sufficient conditions for the program to meet its specification is furthermore guaranteed by a *validation* which is built for each function and can be automatically checked. However, the main source of error in our method could be that our translation of JAVA programs or JML specification does not respect the JAVA/JML semantics. For these reasons, it is important for these interpretations to be clearly stated. This is the purpose of the following sections.

3 Modeling Java Memory States

We have to represent states of JAVA memory by a finite set of WHY variables, and describe the state transitions corresponding to JAVA statements as modifications of those variables.

For local variables of constructors or methods, since such variables are allocated in JAVA memory stack, and cannot be aliased to another local variable or a cell of memory heap, it is sound to represent them as local variables in WHY intermediate language.

3.1 Modeling the Heap

JAVA values are either direct values (integers, booleans or floats) or references to objects or arrays which are represented as adresses allocated in the heap. As mentioned in Sect. 1, the first version of KRAKATOA described in [16] used a naive method considering JAVA memory heap as a single large array mapping addresses to values, but this modeling is very low-level, and proving properties with it amounts to reason all the time on whether two addresses are aliased or not: for example, if a field x is modified, one should expect that it is known for free that any different field y is unchanged ; or when an array cell is modified, the length of the array is not. This is why we adapted Burstall's approach for pointer programs with structures to JAVA programs.

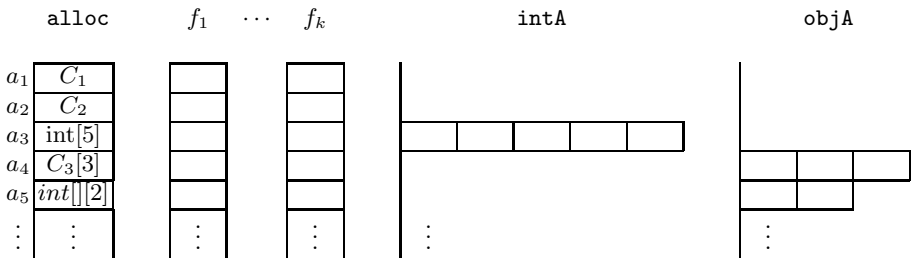

Fig. 3. Modeling of Java memory heap

The set of variables representing a state of Java memory heap is displayed on Fig. 3. All these variables can be seen as maps indexed by addresses $a_1, a_2, \ldots$ belonging to some abstract data type addr.

The variable `alloc` on the left of the figure, is an *allocation store* (type store) which tells for each address whether it is allocated, and if yes what is the type of the structure at this address: an object of some class C or an array of some length l of values of some type t. This variable will be accessed when length of an array is sought and for dynamic typing (`instanceof` and casts), and modified only by `new` statements. The variables $f_1, f_2, \ldots$ represent dynamic fields of objects, in a very similar way as Burstall.

The variables `intA` and `objA` represent the memory locations where arrays of integers and references respectively are allocated (for simplicity, we only consider the basic type `int` here, but in practice booleans and floats are also handled, with variables `boolA` and `floatA`). When an array arr of integers of size l is allocated at address a, then $alloc(a)$ will be equal to `int`$[l]$ and for all integer $0 \leq i < l$, `intA`(a, i) will be an integer corresponding to $arr[i]$. An array of objects will be allocated similarly in the variable `objA`. Notice that the array `objA` cannot be split further because it is possible for any two arrays of objects to be aliased (think of arguments of `System.arraycopy` method of JAVA API).

3.2 First-order Modeling of Values, Classes and Memories

The next step is to design a first-order theory, introducing function and predicate symbols and axioms for them, to model JAVA execution in terms of formulas over the variables introduced in the previous section.

A very useful consequence of our splitting of the heap is that it is statically known whether the contents of some memory cell is an integer or a reference; hence in the logical modeling, instead of a unique sort for representing any memory cell value, we use primitive integers of the logic for values of arithmetic expressions plus a sort value for values corresponding to references (objects or arrays). This sort value is equipped with a function Ref : addr $\rightarrow$ value which builds an object from an address, and a logical constant Null of type value for representing JAVA's `null` value.

We introduce a sort javaType for representing JAVA types of references. With classId being the type of class names of the given JAVA program, javaType is constructed by:

$\mathsf{ClassType} : \mathsf{classId} \rightarrow \mathsf{javaType}$ // for class C
$\mathsf{ArrIntType} : \mathsf{javaType}$ // for array int[]
$\mathsf{ArrayType} : \mathsf{javaType} \rightarrow \mathsf{javaType}$ // for array t[], with t a reference type

The `alloc` variable of type `store` corresponds to a finite partial map which contains the currently allocated values with type information. In practice we shall use the following functions to access type information.

$\mathsf{fresh} : \mathsf{store} \rightarrow \mathsf{value} \rightarrow \mathsf{prop}$
$\mathsf{typeof} : \mathsf{store} \rightarrow \mathsf{value} \rightarrow \mathsf{javaType} \rightarrow \mathsf{prop}$
$\mathsf{arraylength} : \mathsf{store} \rightarrow \mathsf{value} \rightarrow \mathsf{int}$

fresh and typeof corresponds respectively to JML's `\fresh` and `\typeof`; arraylength corresponds to `.length` in JAVA or JML. Derived from the typeof relations, we shall introduce logical interpretation of JAVA predicate instanceof and JML function `\elemtype`:

$\mathsf{instanceof} : \mathsf{store} \rightarrow \mathsf{value} \rightarrow \mathsf{javaType} \rightarrow \mathsf{prop}$
$\mathsf{array\_elemtype} : \mathsf{store} \rightarrow \mathsf{value} \rightarrow \mathsf{javaType}$

Useful properties on these functions are

$$\forall\ s : \mathsf{store}, v : \mathsf{value},\ 0 \leq (\mathsf{arraylength}\ s\ v),$$
$$\forall\ s : \mathsf{store}, t : \mathsf{javaType}, v : \mathsf{value},\ (\mathsf{typeof}\ s\ v\ t) \rightarrow (\mathsf{instanceof}\ s\ v\ t)$$
$$\forall\ s : \mathsf{store}, t : \mathsf{javaType}, v : \mathsf{value},\ (\mathsf{instanceof}\ s\ v\ t) \rightarrow \neg(\mathsf{fresh}\ s\ v)$$
$$\forall\ s : \mathsf{store}, t : \mathsf{javaType}, v : \mathsf{value},$$
$$v \neq \mathsf{Null} \wedge (\mathsf{typeof}\ s\ v\ (\mathsf{ArrayType}\ t)) \rightarrow (\mathsf{array\_elemtype}\ s\ v) = t$$

And for each class C extending class D:

$$\forall\ s : \mathsf{store}, v : \mathsf{value},$$
$$(\mathsf{instanceof}\ s\ v\ (\mathsf{ClassType}\ C)) \rightarrow (\mathsf{instanceof}\ s\ v\ (\mathsf{ClassType}\ D))$$

The store will only be changed when a new object is allocated. The main information we need is that the new store contains the objects allocated in the old store. This is achieved by the introduction of a binary relation store_extends on stores with the following properties (among others):

$$\forall\ s_1, s_2 : \mathsf{store}, v : \mathsf{value},\ (\mathsf{store\_extends}\ s_1\ s_2) \wedge (\mathsf{fresh}\ s_2\ v) \rightarrow (\mathsf{fresh}\ s_1\ v)$$
$$\forall\ s_1\ s_2 : \mathsf{store}, t : \mathsf{javaType}, v : \mathsf{value},$$
$$(\mathsf{store\_extends}\ s_1\ s_2) \wedge (\mathsf{instanceof}\ s_1\ v\ t) \rightarrow (\mathsf{instanceof}\ s_2\ v\ t)$$

KRAKATOA introduces one additional WHY variable (f_i) for each static field of the JAVA program. Variable f_i will have type α memory where α is either int or value, depending of the static type declared for f_i. The basic operations on objects of type α memory are access and update:

$\mathsf{access} : \ \alpha\ \mathsf{memory} \to \mathsf{value} \to \alpha$
$\mathsf{update} : \ \alpha\ \mathsf{memory} \to \mathsf{value} \to \alpha \to \alpha\ \mathsf{memory}$

To reason with combination of access and updates, we have the classical properties of the theory of arrays:

$$\forall\ m : \alpha\ \mathsf{memory}, v : \mathsf{value}, w : \alpha,\ (\mathsf{access}\ (\mathsf{update}\ m\ v\ w)\ v) = w$$
$$\forall\ m : \alpha\ \mathsf{memory}, v_1, v_2 : \mathsf{value}, w : \alpha,$$
$$v_1 \neq v_2 \to (\mathsf{access}\ (\mathsf{update}\ m\ v_1\ w)\ v_2) = (\mathsf{access}\ m\ v_2)$$

The variables `intA` and `objA` have type $\mathsf{int}\ \mathsf{arraymem}$ and $\mathsf{value}\ \mathsf{arraymem}$ respectively. These maps are indexed by both a value and an integer, with operations

$\mathsf{array\_access} : \alpha\ \mathsf{arraymem} \to \mathsf{value} \to \mathsf{int} \to \alpha$
$\mathsf{array\_update} : \alpha\ \mathsf{arraymem} \to \mathsf{value} \to \mathsf{int} \to \alpha \to \alpha\ \mathsf{arraymem}$

and the expected properties for combination of access and updates.

3.3 Coq Realization

We use Why's ability to translate formulas into several prover output formats, in particular for the Coq proof assistant and for the Simplify automatic theorem prover. With Coq output, we furthermore built a *realization* of all axioms of our theory. This has the very important consequence that the original first-order theory is proven consistent.

Our Coq development is structured as a functor, whose parameter is a signature representing the class structure of an arbitrary Java program, made of:

- a set of class identifiers classId;
- a distinguished class identifier ObjectClass (for the Java `Object` class);
- a partial function super associating to its class its superclass ;
- three axioms for assuming decidability of equality on classId, that ObjectClass as no superclass, and that super is a well-founded relation.

From this signature, the functor builds a module which provides a realization of the previous first-order theory, using inductive data types (for representing sorts value, javaType, etc.) and higher-order functions, intensively used in order to represent the memory types and operations, and the sets of memory locations presented further in Sect. 5. It contains 1500 lines of Coq code. We have also introduced 200 lines of specialized tactics in order to mechanize simple reasoning.

On each particular program, Krakatoa generates an instance of the signature above, thus providing a proven consistent modeling of this program.

4 Translating Java Programs

We now give the semantical interpretation of Java statements which access and/or updates the memory heap. The interpretation of complex statements in Java (sequence, if, while, exception throwing and catching, etc.) is not different from [16] so we focus here only on atomic statements of access, assignment, and memory allocation.

4.1 Analysis of Variable Effects

Once we have fixed the set of variables representing memory states, one very important step for being able to deal modularly with function calls in WHY's intermediate language, is to statically compute for any statements the variable effects, as it is shown for example in **reads** and **writes** clauses of Figure 2. This analysis of effects allows to interpret JML `modifiable` clauses, as it will be shown in Sect. 5.

Because JAVA methods in a class can be mutually recursive, the effect analysis uses an iterative process in order to compute the maximal effects of each method, starting from an empty effect. We introduce an environment Γ which associates to each method its (currently known) effects, we write $\Gamma \vdash e : R, W$ to mean that expression (or statement) e reads variables R and writes variables W assuming the methods have effects as given in Γ. We give here a few rules for computing $\Gamma \vdash e : R, W$:

$$\frac{\Gamma \vdash e : R, W}{\Gamma \vdash e.f : R \cup \{f\}, W}$$

$$\frac{e_1 \text{ has type } \texttt{int[]} \qquad \Gamma \vdash e_1 : R_1, W_1 \qquad \Gamma \vdash e_2 : R_2, W_2 \qquad \Gamma \vdash e_3 : R_3, W_3}{\Gamma \vdash e_1[e_2] = e_3 : R_1 \cup R_2 \cup R_3 \cup \{\texttt{intA}, \texttt{alloc}\}, W_1 \cup W_2 \cup W_3 \cup \{\texttt{intA}\}}$$

$$\frac{m : R, W \text{ in } \Gamma \qquad \Gamma \vdash e_1 : R_1, W_1 \qquad \cdots \qquad \Gamma \vdash e_k : R_k, W_k}{\Gamma \vdash m(e_1, \ldots, e_k) : \bigcup_i R_i \cup R, \bigcup_i W_i \cup W}$$

$$\frac{C : R, W \text{ in } \Gamma \qquad \Gamma \vdash e_1 : R_1, W_1 \qquad \cdots \qquad \Gamma \vdash e_k : R_k, W_k}{\Gamma \vdash \texttt{new } C(e_1, \ldots, e_n) : \bigcup_i R_i \cup R \cup \{\texttt{alloc}\}, \bigcup_i W_i \cup W \cup \{\texttt{alloc}\}}$$

From a given Γ, and a given method m with body e, we compute R, W such that $\Gamma \vdash e : R, W$ and update consequentely the information associated to method m in Γ until a fixpoint is reached, which happens in finite time because set of effects are bounded: the number of variables is fixed.

4.2 Memory Access

In our logical model of JAVA program, we introduced total functions in order to be able to represent any well-typed JAVA program. But we want also to detect and avoid any possible runtime exception such as access to a null pointer, or outside the bounds of an array.

Let's consider a Java access expression `v.f`, where v is a variable (the general case of any expression can be dealt by adding a temporary variable) and f a field name. The logical interpretation views f as something of type α memory where α is either int or value, depending on the static type of f. The logical interpretation of `v.f` is (access f v) using the access function introduced in our model (see Sect. 3.2). But this access will generate a runtime exception if v is the null pointer. It is necessary to produce a proof obligation $v \neq \mathsf{Null}$. One possibility could be to introduce in WHY an `access` function with the corresponding precondition. We prefer to use the possibility in WHY to associate pre

or post-condition to any expression. The JAVA `v.f` expression is consequentely translated into the WHY annotated expression: $\{\ v \neq \mathsf{Null}\ \}\ (\mathsf{access}\ f\ v)$. The precondition is omitted when v is the self reference `this`.

Similarly, an array access expression `v[i]` is interpreted as the annotated expression $\{\ v \neq \mathsf{Null} \wedge 0 \leq i < (\mathsf{arraylength}\ \texttt{alloc}\ v)\ \}\ (\mathsf{array\_access}\ A\ v\ i)$, where $A =$ `intA` or `objA` depending on static type of the array. The precondition ensures that no null pointer access and no out-of-bounds access occur.

4.3 Memory Assignments

A field assignment `v.f = w` is interpreted as $f := \{\ v \neq \mathsf{Null}\ \}\ (\mathsf{update}\ f\ v\ w)$. Updating the field f of object v is protected by the condition that v should not be `null`, another natural condition is to check that the type of the object w is an instance of the type of the field f. However, this is a consequence of static typing and type-safety in JAVA so we do not need to add extra checking.

The situation is different for array assignement `v[i] = w`. Assume D is a subclass of C, it can be the case that v is statically an array of C, but dynamically an array of D in which case updating v with an object w in the class C is statically correct but fails at runtime. In order to avoid this error, we interpret `v[i] = w` into

$$A := \{\ v \neq \mathsf{Null} \wedge 0 \leq i < (\mathsf{arraylength}\ \texttt{alloc}\ v) \wedge (\mathsf{instanceof}\ \texttt{alloc}\ w\ (\mathsf{array\_elemtype}\ \texttt{alloc}\ v))\ \}\ (\mathsf{array\_update}\ A\ v\ i\ w)$$

4.4 Memory Allocation

A JAVA object creation expression `new C`$(v_1, \ldots, v_n)$ is interpreted as

$\texttt{let}\ this = (\mathsf{alloc\_obj}\ C)\ \texttt{in}\ C_{fun}(this, v_1, \ldots, v_n); this$

where C_{fun} is the WHY function for the coresponding constructor. Unlike access and update functions, alloc_obj is a function with side-effects, specified in WHY:

parameter alloc_obj : c : classId $\rightarrow \{\}$ value **reads** `alloc` **writes** `alloc`
$\{\ result \neq \mathsf{Null} \wedge (\mathsf{fresh}\ \texttt{alloc@}\ result) \wedge (\mathsf{typeof}\ \texttt{alloc}\ result\ (\mathsf{ClassType}\ c))$
$\wedge(\mathsf{store\_extends}\ \texttt{alloc@}\ \texttt{alloc})\}$

Array creation `new C[l]` is interpreted as $(\mathsf{alloc\_array}\ C\ l)$ where alloc_array is specified as

parameter alloc_array : t : javaType $\rightarrow n$: int $\rightarrow \{0 \leq n\}$ value
reads `alloc` **writes** `alloc`
$\{\ result \neq \mathsf{Null} \wedge (\mathsf{fresh}\ \texttt{alloc@}\ result) \wedge (\mathsf{typeof}\ \texttt{alloc}\ result\ (\mathsf{ArrayType}\ t))$
$\wedge(\mathsf{arraylength}\ \texttt{alloc}\ result) = n \wedge (\mathsf{store\_extends}\ \texttt{alloc@}\ \texttt{alloc})\}$

There is also a special variant alloc_int_array for `new int[l]`.

Notice that in order to prove JAVA programs, we do not need to know how these functions are implemented. However, in order to avoid axioms in our model,

```
/*@ normal_behavior
  @   requires p1 != null && p2 != null && p1 != p2;
  @   modifiable p2.balance;
  @   ensures \result == \old(p1.balance); */
public static int test(Purse p1,Purse p2) {
    p2.credit(100);
    return p1.balance;
}
```

Fig. 4. An example of reasoning on aliases and frame conditions

the alloc_obj and alloc_array functions have also been implemented in WHY using more primitive functional operations for computing a non allocated value in a store and creating an updated store. We have a COQ proof of correctness of these implementations (technically, this requires the addr type to be infinite, since we do not consider memory overflow).

5 Modeling Frame Conditions

JML `modifiable` clauses are essential for reasoning modularly when several methods are involved. As a toy example, let's imagine a new method in our class Purse, given Fig. 4. Proof of the post-condition needs the fact that `p1.balance` is not modified by the call to `p2.credit(100)`. Our modeling allows to prove this using the modifiable predicates in the post-condition of `credit` and the pre-condition `p1 != p2`, that forbids aliasing of `p1` and `p2`.

We model `modifiable` clauses using the predicates

$$\mathsf{modifiable} : \ \mathsf{store} \to \alpha\ \mathsf{memory} \to \alpha\ \mathsf{memory} \to \mathsf{set\_loc} \to \mathsf{prop}$$
$$\mathsf{array\_modifiable} : \ \mathsf{store} \to \alpha\ \mathsf{arraymem} \to \alpha\ \mathsf{arraymem} \to \mathsf{array\_set\_loc} \to \mathsf{prop}$$

respectively for objects locations and array locations. set_loc (resp. array_set_loc) are logic types representing sets of modifiable locations for objects (resp. for arrays).

In general, the post-condition of a method will have one modifiable predicate for each of the WHY variables it modifies, as they are computed by the analysis of effects of Sect. 4.1. Splitting of the JML `modifiable` clause into modifiable predicate for each modified variable is computable automatically.

According to JML informal semantics, a `modifiable` clause with set of locations *loc* specifies that in the post-state of the considered method, every memory location which is already allocated in the pre-state and is not included in *loc* is unchanged. This formally results in the following:

$$\begin{array}{l}\forall\ s : \mathsf{store}, m_1, m_2 : \alpha\ \mathsf{memory}, loc : \mathsf{set\_loc},\\ \quad (\mathsf{modifiable}\ s\ m_1\ m_2\ loc) \leftrightarrow\\ \qquad \forall\ v : \mathsf{value},\ \neg(\mathsf{fresh}\ s\ v) \wedge (\mathsf{notin}\ v\ loc) \to (\mathsf{access}\ m_1\ v) = (\mathsf{access}\ m_2\ v)\end{array}$$

$$\forall\ s : store, m_1, m_2 : \alpha\ \mathsf{arraymem}, loc : \mathsf{array\_set\_loc},$$
$$(\mathsf{array\_modifiable}\ s\ m_1\ m_2\ loc) \leftrightarrow$$
$$\forall\ v : \mathsf{value},\ n : \mathsf{int}, \neg(\mathsf{fresh}\ s\ v) \wedge (\mathsf{array\_notin}\ v\ n\ loc) \rightarrow$$
$$(\mathsf{array\_access}\ m_1\ v\ n) = (\mathsf{array\_access}\ m_2\ v\ n)$$

and it remains to give axioms for the notin (resp. array_notin) functions, depending on the form of the locations specified.

The JML clause `modifiable \nothing` specifies that nothing is modified. It is interpreted with a new constant empty_loc of type set_loc (resp. array_empty_loc of type array_set_loc) with the axioms:

$$\forall v : \mathsf{value}, \quad (\mathsf{notin}\ v\ \mathsf{empty\_loc})$$
$$\forall v : \mathsf{value}, n : \mathsf{int}, \quad (\mathsf{array\_notin}\ v\ n\ \mathsf{array\_empty\_loc})$$

The JML clause `modifiable v.f` specifies that the field f of v is modified. It is interpreted with a new function value_loc of type value → set_loc with the axiom:

$$\forall v'\ v : \mathsf{value}, \quad (\mathsf{notin}\ v'\ (\mathsf{value\_loc}\ v)) \leftrightarrow v' \neq v$$

Analogously, The JML clauses `modifiable t[i]`, `modifiable t[i..j]` and `modifiable t[*]` are interpreted using functions array_loc, array_sub_loc and array_all_loc with axioms

$$\forall\ v\ t : \mathsf{value},\ n\ i : \mathsf{int}, (\mathsf{array\_notin}\ v\ n\ (\mathsf{array\_loc}\ t\ i)) \leftrightarrow (v \neq t \vee i \neq n)$$
$$\forall\ v\ t : \mathsf{value},\ n\ i\ j : \mathsf{int},$$
$$(\mathsf{array\_notin}\ v\ n\ (\mathsf{array\_sub\_loc}\ t\ i\ j)) \leftrightarrow v \neq t \vee n < i \vee n > j$$
$$\forall\ v\ t : \mathsf{value},\ n : \mathsf{int}, (\mathsf{array\_notin}\ v\ n\ (\mathsf{array\_all\_loc}\ t)) \leftrightarrow v \neq t$$

When a JML clause `modifiable l1,l2` specifies several (say two) locations l_1 and l_2, then two cases may occur: either l_1 and l_2 refer to locations represented by different variables of the memory heap representation, and in that case a conjunction of two modifiable assertions is built; or they refer to the same variable, and then the clause is interpreted using the function union_loc (resp. array_union_loc) with axioms

$$\forall v : \mathsf{value},\ l_1, l_2 : \mathsf{set\_loc},$$
$$(\mathsf{notin}\ v\ (\mathsf{union\_loc}\ l_1\ l_2)) \leftrightarrow (\mathsf{notin}\ v\ l_1) \wedge (\mathsf{notin}\ v\ l_2)$$
$$\forall v : \mathsf{value}\ n : \mathsf{int},\ l_1, l_2 : \mathsf{array\_set\_loc},$$
$$(\mathsf{array\_notin}\ v\ n\ (\mathsf{array\_union\_loc}\ l_1\ l_2))$$
$$\leftrightarrow (\mathsf{array\_notin}\ v\ n\ l_1) \wedge (\mathsf{array\_notin}\ v\ n\ l_2)$$

Notice also that if a variable is detected as written by the analysis of effects, but there is no `modifiable` location refering to it, then we have to add an assertion (modifiable $\cdots$ empty_loc) for it. Finally, the JML clause `modifiable \everything` is interpreted simply by building no modifiable predicate, for specifying no information at all.

These constructions have been implemented in our Coq realization. The sort set_loc is interpreted as the functional type value → Prop representing intentionally a set of locations. We interpret a set of locations directly as the predicate which is true for values which are not in this set of locations, such that the predicate notin can be interpreted directly without extra negation.

```
class Q {
  int i; int[] a;

 /*@ normal_behavior
   @  requires 0<=i && i+1 < a.length;
   @  modifiable i,a[i]; */
  void q() { i++; a[i]=3; }
}
```

```
class O { int i; }
class P {
  O x; O y;
 /*@ normal_behavior
   @  requires y != null;
   @  modifiable x,x.i;  */
  void p() { x=y; x.i=7; }
}
```

Fig. 5. Two programs where ChAsE gives the wrong answer

6 Examples

ESC/Java does not check that a method meets the frame condition written in its specification, but it assumes that this condition is fulfilled when the method is called. This is one of the major sources of unsoundness. The ChAsE [9] tool was designed for automatically checking frame conditions, but it works at a syntactic level and consequently can give incorrect diagnosis: two such examples are given [9] (see Figure 5). ChAsE accepts these programs with incorrect frame conditions, but not with the appropriate ones which are `a[i+1]` instead of `a[i]` for method `q` and `y.i` instead of `x.i` for method `p`. On the other hand, Krakatoa gives automatically the correct diagnosis for both programs. For instance for `p` with clause `modifiable y.i`, which, according to JML semantics, denotes the field i at address $this.y$ in the pre-state of the method: Krakatoa interprets this clause as (modifiable `alloc`@ i@ i (value_loc (access y@ $this$))) which is easily provable.

In practice, most proof obligations generated are solved automatically using Simplify, for example all obligations of the Krakatoa tutorial (a simple electronic purse, maximum of an array, etc.) and the Dijkstra's Dutch Flag program of [16]. In comparaison, using Coq with simple ad-hoc tactics, the proofs for the `Purse` (resp. `Flag`, resp. `Arrays`) programs require 20 (resp. 60, resp. 100) lines of tactics.

7 Conclusions, Related Works, and Future Work

7.1 Combining First-order and Higher-order Models

The Why tool generates proof obligations written in a first-order multi-sorted theory, using the Why primitive operations on basic types such as integers or booleans and also model-specific symbols for constants, functions and predicates. In order to prove properties involving these symbols, we provide an axiomatic first-order theory, used to discharge the proof obligations with an automatic prover such as Simplify. We developed a Coq realization of that theory. This model uses higher-order constructions for representing the memories operations. If we assume that deduction steps performed by Simplify are correct, then they

could be translated into Coq, leading to a complete proof in Coq of the original proof obligations.

By designing a suitable modeling of Java memory states, together with a static computation of effects and a suitable background first-order theory, we obtained a powerful method for proving functional properties of Java programs. Combined with an automatic theorem prover, we are able to establish automatically more properties than static checkers like ESC/Java or ChAsE, with a method whose soundness only rely on the soundness of the translation provided in Section 4.

We believe we made a significant step in filling the gap between static checking techniques, fully automatic but unsound, and true formal verification which requires user interaction: our approach is a compromise between safety of the global approach and push-button technology.

We took advantage of the modular architecture of Why which does all the work of generation of proof obligations for different provers. We believe that this modular architecture is a good approach, that can be easily reused for different input languages than Java. For example we have been able to build, in a quite short time, a similar modeling for C programs [12], with full support for pointer arithmetic.

7.2 Related Work

In [17], F. Mehta and T. Nipkow used the Isabelle proof assistant in order to prove an imperative program involving pointers using a model similar to ours, but without arrays nor memory allocation.

Several tools exist which manipulate Java programs annotated with JML specifications [6]. 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. We already mentioned ESC/Java which is fully automatic but does not guaranty correctness. The memory modeling of ESC/Java seems similar to ours. LOOP [22, 23], Jive [18], JACK [5] or our tool Krakatoa are intended to generate for any JML specification of the program, sufficient verification conditions for these properties to hold. These tools are based on different techniques: both Jive and LOOP use a global memory modeling ; Jive is based on a weakest precondition generator ; in LOOP, the semantics of JML-annotated Java programs is translated into functional PVS expressions which represent the denotational semantics of the program, and properties of these programs can be established using specialized PVS tactics. The JACK environment [5], initially developed by the Gemplus company and now by INRIA, uses a memory model similar to ours, and was initially designed for generating proof obligations for the B system [1] but now also has an output for Simplify.

7.3 Future Work

Automatic provers are useful for early detection of errors in code or specification. We plan to be able to analyze counter-examples in order to suggest proof annotations. A partial analysis of correctness of loops could also help in finding appropriate loop invariants.

One very interesting future work is to be able to build, with the underlying automatic prover, a proof trace which could be double-checked by a proof assistant: in this way, only proof obligations that cannot be solved automatically would need to be proved manually. To obtain such a trace, the use of the haRVey [20] and CVC-lite[3] tools is currently under investigation.

There are still important Java and JML features not yet supported by the Krakatoa tool. Handling the class invariants may become heavy when they are many objects involved, and their combination with inheritance causes important theoretical issues [2].

Acknowledgements We thank Gary T. Leavens and David Cok for their useful comments on a preliminary version of this paper.

References

[1] Jean-Raymond Abrial. *The B-Book, assigning programs to meaning.* Cambridge University Press, 1996.

[2] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, June 2004.

[3] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Rajeev Alur and Doron A. Peled, editors, *16th International Conference on Computer Aided Verification*, volume 3114 of *Lecture Notes in Computer Science*, Boston, MA, USA, July 2004. Springer-Verlag.

[4] Richard Bornat. Proving pointer programs in Hoare logic. In *Mathematics of Program Construction*, pages 102–126, 2000.

[5] Lilian Burdy. JACK: Java Applet Correctness Kit. Gemplus Developer Conference, 2002.

[6] Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. Technical Report NIII-R0309, Dept. of Computer Science, University of Nijmegen, 2003.

[7] Rod Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.

[8] Néstor Cataño, M. Gawlowski, Marieke Huisman, Bart Jacobs, Claude Marché, Christine Paulin, Erik Poll, Nicole Rauch, and Xavier Urbain. Logical techniques for applet verification. Deliverable 5.2, IST VerifiCard project, 2003. `http://www.cs.kun.nl/VerifiCard/files/deliverables/deliverable_5_2.pdf`.

[9] N. Cataño and M. Huisman. Chase: A static checker for JML's assignable clause. In Lenore D. Zuck, Paul C. Attie, Agostino Cortesi, and Supratik Mukhopadhyay, editors, *Proc. VMCAI*, volume 2575 of *Lecture Notes in Computer Science*, pages 26–40, New York, USA, January 2003. Springer-Verlag.

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

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

[12] Jean-Christophe Filliâtre and Claude Marché. Multi-prover verification of C programs. In Jim Davies, Wolfram Schulte, and Mike Barnett, editors, *Sixth International Conference on Formal Engineering Methods*, volume 3308 of *Lecture Notes in Computer Science*, pages 15–29, Seattle, WA, USA, November 2004. Springer-Verlag.

[13] Bart Jacobs, Claude Marché, and Nicole Rauch. Formal verification of a commercial smart card applet with multiple tools. In *Algebraic Methodology and Software Technology*, volume 3116 of *Lecture Notes in Computer Science*, Stirling, UK, July 2004. Springer-Verlag.

[14] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, Iowa State University, 2000.

[15] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, and Clyde Ruby. *JML Reference Manual*, April 2003. draft.

[16] Claude Marché, Christine Paulin-Mohring, and Xavier Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004. `http://krakatoa.lri.fr`.

[17] Farhad Mehta and Tobias Nipkow. Proving pointer programs in higher-order logic. In Franz Baader, editor, *19th Conference on Automated Deduction*, Lecture Notes in Computer Science. Springer-Verlag, 2003.

[18] J. Meyer, P. Müller, and A. Poetzsch-Heffter. The JIVE system. `http://www.informatik.fernuni-hagen.de/pi5/publications.html`, 2000.

[19] Greg Nelson. Techniques for program verification. Research Report CSL-81-10, Xerox Palo Alto Research Center, 1981. `http://research.compaq.com/SRC/esc/Simplify.html`.

[20] Silvio Ranise and David Deharbe. Light-weight theorem proving for debugging and verifying units of code. In *Proc. SEFM'03*, Canberra, Australia, September 2003. IEEE Computer Society Press. `http://www.loria.fr/equipes/cassis/softwares/haRVey/`.

[21] The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.0*, April 2004. `http://coq.inria.fr`.

[22] J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. In D. Bert, C. Choppy, and P. Mosses, editors, *Recent Trends in Algebraic Development Techniques*, volume 1827 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 2000.

[23] Joachim van den Berg and Bart Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *Proc. TACAS'01*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer-Verlag, 2001. `http://www.cs.kun.nl/~bart/LOOP`.