Precise analysis of memory consumption using program logics

Gilles Barthe[1], Mariela Pavlova[1], and Gerardo Schneider[2]

[1] INRIA Sophia–Antipolis, France
[2] Department of Informatics, University of Oslo, Norway

Abstract. Memory consumption policies provide a means to control resource usage on constrained devices, and play an important role in ensuring the overall quality of software systems, and in particular resistance against resource exhaustion attacks. Such memory consumption policies have been previously enforced through static analyses, which yield automatic bounds at the cost of precision, or run-time analyses, which incur an overhead that is not acceptable for constrained devices.
In this paper, we study the use of logical methods to specify and verify statically precise memory consumption policies for Java bytecode programs. First, we demonstrate how the Bytecode Modeling Language BML (a variant of the Java Modeling Language JML tailored to bytecode) can be used to specify precise memory consumption policies for (sequential) Java applets, and how verification tools can be used to enforce such memory consumption policies. Second, we consider the issue of inferring some of the annotations required to express the memory consumption policy. We report on an inference algorithm and illustrate its applicability on non-trivial examples.
Our broad conclusion is that logical methods provide a suitable means to specify and verify expressive memory consumption policies, with a minimal overhead.

1 Introduction

Trusted personal devices (TPDs for short) such as smart cards, mobile phones, and PDAs commonly rely on execution platforms such as the Java Virtual Machine and the Common Language Runtime. Such platforms are considered appropriate for such TPDs since they allow applications to be developed in a high-level language without committing to any specific hardware and since they feature security mechanisms that guarantee the innocuousness of downloaded applications. For example, the Java security architecture ensures that applications will not perform illegal memory accesses through stack inspection, which performs access control during execution, and bytecode verification, which performs static type-checking prior to execution. On the other hand, current security architectures for TPDs do not provide any mechanism to control resource usage by downloaded applications, despite TPDs being subject to stringent resource constraints. Therefore, TPDs are particularly vulnerable to denial-of-service attacks, since executing a downloaded application may potentially lead to resource exhaustion.
Several approaches have been suggested to date to enforce memory consumption policies for programs; all approaches are automatic, but none of them is ideally suited for TPDs, either for their lack of precision, or for the runtime penalty they impose on programs:

– *Static analyses and abstract interpretations:* in such an approach, one performs an abstract execution of an approximation of the program. The approximation is chosen to be coarse enough to be computable, as a result of which it yields automatically bounds on memory consumption, but at the cost of precision. Such methods are not very accurate for recursive methods and loops, and often fail to provide bounds for programs that contain dynamic object creation within a loop or a recursive method;

– *Proof-carrying code:* here the program comes equipped with a specification of its memory consumption, in the form of statements expressed in an appropriate program logic, and a certificate that establishes that the program verifies the memory consumption specification attached to it. The approach potentially allows for precise specifications. However, existing works on proof carrying code for resource usage sacrifice the possibility of enforcing accurate policies in favor of the possibility of generating automatically the specification and the certificate, in line with earlier work on certifying compilation;
– *Run-time monitoring:* here the program also comes equipped with a specification of its memory consumption, but the verification is performed at run-time, and interrupted if the memory consumption policy is violated. Such an approach is both precise and automatic, but incurs a runtime overhead which makes it unsuitable for TPDs.

The objective of this work is to explore an alternative approach that favors precision of the analysis at the cost of automation. The approach is based on program logics, which originate from the seminal work on program verification by C.A.R. Hoare and E.W. Dijkstra and have been used traditionally to verify functional properties of programs. In earlier work, we have shown how general purpose logics can be used to enforce security properties of Java programs, including confidentiality [6] and high-level security rules [23]. In this paper, we demonstrate that program logics are also appropriate for performing a precise analysis of resource consumption for Java programs. Although our method is applicable both at source code level and bytecode level, our work has focused on bytecode level, since in many application domains verification has to be performed without access to the source code of the applet. (However, for the clarity of the explanations all examples in the introduction deal with source code level.)

In order to illustrate the principles of our approach, let us consider the following program:

```
public void m(A a){
  if (a  == null) {
    a = new A();
  }
   a.b = new B();
}
```

In order to model the memory consumption of this program, we introduce a *ghost* (or, *model*) variable `Mem` that accounts for memory consumption; more precisely, the value of `Mem` at any given program point is meant to provide an upper bound to the amount of memory consumed so far. To keep track of the memory consumption, we perform immediately after every bytecode that allocates memory an increment of `Mem` by the amount of memory consumed by the allocation. Thus, if the programmer specifies that `ka` and `kb` is the memory consumed by the allocation of an instance of class `A` and `B` respectively, the program must be annotated as:

```
public void m(A a) {
  if (a  == null) {
    a = new A();
    // set Mem = ka;
  }
   a.b = new B();
   // set Mem = kb;
}
```

Such annotations allow to compute at run-time the memory consumption of the program. However, we are interested in static prediction of memory consumption, and resort to preconditions and postconditions to this end. Even for a simple example as above, one can express the specification at different levels of granularity. For example, fixing the amount of memory that the the program may use `Max` one can specify that the method will use at most `ka + kb` memory units and will not overpass the authorized limit to use `Max` with the following specification:

```
//@  requires Mem + ka +  kb <= Max
//@ ensures  Mem <= \old(Mem) + ka +  kb
public void m(A a) {
  if (a  == null) {
    a = new A();
    // set Mem = ka;
  }
   a.b = new B();
   // set Mem = kb;
}
```

Or try to be more precise and relate memory consumption to inputs with the following specification:

```
//@ requires a == null ==> Mem + ka +  kb <= Max &&
   !(a  == null) ==> Mem + kb <= Max
//@ ensures \old(a) == null ==> Mem <= \old(Mem) + ka +  kb &&
    !(\old(a)  == null) ==> Mem <= \old(Mem) + kb
public void m(A a) {
  if (a  == null) {
    a = new A();
  }
   a.b = new B();
}
```

More complex specifications are also possible. For example, one can take into account whether the program will throw an exception or not. using (possibly several) exceptional postconditions stating that k_E memory units are allocated in case the method exits on exception `E`.
The main characteristics of our approach are:

- *Precision:* our analysis allows to specify and enforce very precise memory consumption policies, including policies that take into account the results of branching statements or the values of parameters in method calls. Being based on program logics, which are very versatile, the precision of our analysis can be further improved by using it in combination with other analyses, such as control flow analysis and exception analysis;
- *Correctness:* our analysis exploits existing program logics which are (usually) already known to be sound. In fact, it is immediate to derive the soundness of our analysis from the soundness of the program logic, provided ghost annotations that update memory consumption variables are consistent with an instrumented semantics that extends the language operational semantics with a suitable cost model that reflects resource usage;
- *Language coverage:* our analysis relies on the existence of a verification condition generator for the programming language at hand, and is therefore scalable to complex programming features. In the course of the paper, we shall illustrate applications of our approach to programs featuring recursive methods, method overriding and exceptions;

– *Usability:* our approach can be put to practice immediately using existing verification tools for program logics. We have applied our approach to annotated Java bytecode programs using a verification environment developed in [7], but it is also possible to use our approach on JML annotated Java source code [8], and more generally on programs that are written in a language for which appropriate support for contract-based reasoning exists;
– *Annotation generation:* in contrast to other techniques discussed above, our approach requires user interaction, both for specifying the program and for proving that it meets its specification. In order to reduce the burden of the user, we have developed heuristics that infer automatically part of the annotations;
– *Feasibility:* thanks to annotation generation mechanisms and powerful provers that help discharge many proof obligations automatically, our approach can be applied to realistic Java bytecode programs with a reasonable overhead.

In the course of the article, we illustrate the principles and characteristics of our approach in the context of Java bytecode programs. More specifically, the paper is organized as follows: Section 2 provides a brief introduction to Java bytecode programs and to the modeling language and weakest precondition calculus used to specify and verify such programs. Section 3 describes in some detail how the infrastructure described in Section 2 can be used to specify and verify precise memory consumption policies. Section 4 is devoted to a presentation of our algorithms for inferring automatically annotations. We conclude in Section 5 with related work and in Section 6 with directions for future work.

2 Preliminaries

2.1 Java class files

The standard format for Java bytecode programs is the so-called class file format which is specified in the Java Virtual Machine Specification [21]. For the purpose of this paper, it is sufficient to know that class files contain the definition of a single class or interface, and are structured into a hierarchy of different attributes that contain information such as the class name, the name of its superclass or the interfaces it implements, a table of the methods declared in the class. Moreover an attribute may contain other attributes. For example the attribute that describes a single method contains an `Local_Variable_Table` attribute that describes the method parameters and its local variables; further in this section we will denote the table of local variables by l and the i^{th} variable by $l[i]$. In addition to these attributes which provide all the information required by a standard implementation of the Java Virtual Machine, class files can accommodate user-defined attributes, which are not used by standard implementations of the Java Virtual Machine but can be used for other purposes. We take advantage of this possibility and introduce additional attributes that contain annotations such as method preconditions and postconditions, variants and invariants. Annotations are given in the Bytecode Modeling Language, which we describe below.

2.2 The Bytecode Modeling Language

The bytecode modeling language BML is a variant of the Java Modeling Language (JML) [20] tailored to Java bytecode; the BML specification language is described in [7]. For our purposes, we only need to consider a restricted fragment of BML, which is given in Fig. 1; we let $\mathcal{E}$ and $\mathcal{P}$ denote respectively the set of BML expressions and BML predicates. As for JML, BML specifications contain different forms of statements, in the form of BML predicates tagged with appropriate keywords. BML predicates are built from BML expressions using standard predicate logic; furthermore BML expressions are bytecode programs that correspond to effect-free Java expressions, or BML specific expressions. The latter include expressions of the form `\old(exp)` which refers to the value of the expression `exp` at

the beginning of the method, or $\texttt{exp}^{\texttt{pc}}$ which refers to the value of the expression `expr` at program point `pc`. Note that the latter is not a standard expression in JML but can be emulated introducing a ghost variable $\texttt{exp}^{\texttt{pc}}$ and performing the ghost assignment $\texttt{set exp}^{\texttt{pc}}\texttt{= exp}$ at program point `pc`.

Statements can be used for the following purposes:

- Specifying method preconditions, which following the design by contract principles, must be satisfied upon method invocation. Such preconditions are formulated using statements of the form `requires` $\mathcal{P}$;
- Specifying method postconditions, which following the design by contract principles, must be guaranteed upon returning normally from the method. Such postconditions are formulated using statements of the form `ensures` $\mathcal{P}$;
- Specifying method exceptional postconditions, which must be guaranteed upon returning exceptionally from the method. Such postconditions are formulated using statements of the form `exsures(Exception)` $\mathcal{P}$, that record the reason for exceptional termination;
- Stating loop invariants, which are predicates that must hold every time the program enters the loop;
- Guaranteeing termination of loops and recursive methods, using statements of the form `variant` $\mathcal{E}$ which provide a measure (in the case of BML a positive number) that strictly decreases at each iteration of the loop/recursive call;
- Local assertions, using `assert` $\mathcal{P}$, which asserts that $\mathcal{P}$ holds at the program point immediately after the assertion;
- Declaring and updating ghost variables, using statements of the form `declare Model` $Type\ name$ and `set` $\mathcal{E} = \mathcal{E}$;
- Keeping track of variables that are modified by a method or in a loop, using declarations of the form `modifies` var. During the generation of verification conditions, one checks that variables that are not declared as modifiable by the clause above will not be modified during the execution of the method/loop, and one also uses the information about modified variables to generate the verification conditions.

$$
\begin{aligned}
\text{BML} - \textsf{stmt} = \ & \texttt{requires}\ \mathcal{P} \\
| \ & \texttt{ensures}\ \mathcal{P} \\
| \ & \texttt{exsures(Exception)}\ \mathcal{P} \\
| \ & \texttt{assert}\ \mathcal{P} \\
| \ & \texttt{invariant}\ \mathcal{P} \\
| \ & \texttt{variant}\ \mathcal{E} \\
| \ & \texttt{declare Model}\ Type\ name \\
| \ & \texttt{modifies}\ var \\
| \ & \texttt{set}\ \mathcal{E} = \mathcal{E}
\end{aligned}
$$

Fig. 1. SPECIFICATION LANGUAGE

Note that, as alluded in the previous paragraph, annotations are not inserted directly into bytecode; instead they are gathered into appropriate user defined attributes of an extended class file. Such extended class files can be obtained either through direct manipulation of standard class files, or using an extended compiler that outputs extended class files from JML annotated programs, see [7].

2.3 Verification of annotated bytecode

In order to validate annotated Java bytecode programs, we resort to a verification environment for Java bytecode (described in [7]), which is an adaptation of JACK [8]. It consists of two main components:

- A verification condition generator, which takes as input an annotated applet and generates a set of verification conditions which are sufficient to guarantee that the applet meets its specification;
- A proof engine that attempts to discharge the verification conditions automatically using automatic tools such as B and Simplify, and then sends the remaining verification conditions to proof assistants where they can be discharged interactively by the user. We are currently generating verification conditions for the proof assistants Coq [11] and PVS [27].

Generating the Verification Conditions The verification condition generator, or VCGen for short, takes as input an extended class file and returns as outputs a set of proof obligations, whose validity guarantees that the program satisfies its annotations. The VCGen proceeds in a modular fashion in the sense that it addresses each method separately, and is based on computing weakest preconditions. More precisely, for every method *m*, postcondition ψ that must hold after normal termination of *m*, and exceptional postcondition ψ' that must hold after exceptional termination of *m* (for simplicity we consider only one exception in our informal discussion), the VCGen computes a predicate ϕ whose validity at the onset of method execution guarantees that ψ will hold upon normal termination, and ψ' will hold upon exceptional termination. The VCGen will then return several proof obligations that correspond, among other things, to the fact that the precondition of *m* given by the specification entails the predicate ϕ that has been computed, and to the fact that variants and invariants are correct.
The procedure for computing weakest preconditions is described in detail in [7]. In a nutshell, one first defines for each bytecode a predicate transformer that takes as input the postconditions of the bytecode, i.e. the predicates to be satisfied upon execution of the bytecode (different predicates can be provided in case the bytecode is a branching instruction), and returns a predicate whose validity prior to the execution of bytecode guarantees the postconditions of the bytecode. The definition of such functions is completely generic and independent of any program, so the next step is to use these functions to compute weakest preconditions for programs. This is done by building the control flow graph of the program, and then by computing the weakest preconditions of the program using its control flow graph.
Note that the verification condition generator operates on BML statements which are built from extended BML expressions. Indeed, predicate transformers for instructions need to refer to the operand stack and must therefore consider expressions of the form `st(top -+ i)` which represent the `st(top -+ i)`-th element from the stack top:

$$wp(\texttt{store}\ l(i), \psi, \psi') = \psi[\texttt{top} \leftarrow \texttt{top-1}][l[i] \leftarrow \texttt{st(top)}].$$

Discharging verification conditions Verification conditions are expressed in an intermediate language for which translations to automatic theorem provers and proof assistants exist.

2.4 Correctness of the method

The verification method is correct in the sense that one can prove that for all methods *m* of the program the postcondition (resp. exceptional postcondition) of the method holds upon termination (resp. exceptional termination) of the method provided the method is called in a state satisfying the method precondition and provided all verification conditions can be shown to be valid.
The correctness of the verification method is established relative to an operational semantics that describes the transitions to be taken by the virtual machine depending upon the state in which the machine is executed. There are many formalizations of the operational semantics of the Java Virtual Machine,

see e.g. [29, 14, 19, 28]. Such semantics manipulate states of the form $\langle\!\langle h, \langle m, pc, l, s\rangle, sf\rangle\!\rangle$, where h is the heap of objects, $\langle m, pc, l, s\rangle$ is the current *frame* and sf is the current call stack (a list of frames). A frame $\langle m, pc, l, s\rangle$ contains a method name m and a program point pc within m, a set of local variables l, and a local operand stack s. The operational semantics for each instruction is formalized as rules specifying transition between states, or between a state and some tag that indicates abnormal termination. For example, the semantics of the instruction `store` is given by the transition rule below, where $\text{InstAt}(m, pc)$ is the function that extracts the pc-th instruction from the body of method *m*:

$$\frac{\text{InstAt}(m, pc) = \texttt{store}\ i}{\langle\!\langle h, \langle m, pc, l, v :: s\rangle, sf\rangle\!\rangle \rightarrow_{\texttt{store}\ i} \langle\!\langle h, \langle m, pc + 1, l[i \mapsto v], s\rangle, sf\rangle\!\rangle.}$$

In order to establish the correctness of our method, one first needs to establish the correctness of the predicate transformer for each bytecode. For example for the instruction `store` we show that:

$$wp(\texttt{store}\ i, \psi)(\langle\!\langle h, \langle m, pc, l, v :: s\rangle, sf\rangle\!\rangle) \Rightarrow \psi(\langle\!\langle h, \langle m, pc + 1, l[i \mapsto v], s\rangle, sf\rangle\!\rangle)$$

In the above $\psi(\langle\!\langle h, \langle m, pc, l, v :: s\rangle, sf\rangle\!\rangle)$ is to be understood as the instance of the formula ψ in which all local variables l and field references are substituted with their corresponding values in state $\langle\!\langle h, \langle m, pc, l, v :: s\rangle, sf\rangle\!\rangle$.

The proof proceeds by a case analysis on the instruction to be executed, and makes an intensive use of auxiliary substitution lemmas that relate e.g. the stack of the pre-state with the stack of the post-state of executing an instruction. Then one proves the correctness of the method by induction on the length of the execution sequence. We have proved the correctness of our method for a fragment of the JVM that includes the following constructs:

- Stack manipulation: `push`, `pop`, `dup`, `dup2`, `swap`, `numop`, etc;
- Arithmetic instructions: type_`add`, type_`sub`, etc;
- Local variables manipulation: type_`load`, type_`store`, etc;
- Jump instructions: `if`, `goto`, etc;
- Object creation and object manipulation: `new`, `putfield`, `getfield`, `newarray`, etc;
- Array instructions: `arraystore`, `arrayload`, etc;
- Method calls and return: `invokevirtual`, `return`, etc;
- subroutines: `jsr` and `ret`.

Note however that our method imposes some mild restrictions on the structure of programs: for example, we require that `jsr` and `throw` instructions are not entry for loops in the control flow graph in order to prevent pathological recursion. Lifting such restrictions is left for future work.

3 Modeling memory consumption

The objective of this section is to demonstrate how the user can annotate and verify programs in order to obtain an upper bound on memory consumption. We begin by describing the principles of our approach, then turn to establish its soundness, and finally show how it can be applied to non-trivial examples involving recursive methods and exceptions.

3.1 Principles

Let us begin with a very simple memory consumption policy which aims at enforcing that programs do not consume more than some fixed amount of memory `Max`. To enforce this policy, we first introduce a ghost variable `MemUsed` that represents at any given point of the program the memory used so far. Then, we annotate the program both with the policy and with additional statements that will be used to check that the application respects the policy.

The precondition of the method *m* should ensure that there must be enough free memory for the method execution. Suppose that we know an upper bound of the allocations done by method *m* in any execution. We will denote this upper bound by `methodConsumption(m)`. Thus there must be at least `methodConsumption(m)` free memory units from the allowed `Max` when method *m* starts execution. Thus the precondition for the method *m* is:

$$\texttt{requires MemUsed} + \texttt{methodConsumption(}m\texttt{)} \leq \texttt{Max}.$$

The precondition of the program entry point (i.e., the method from which an application may start its execution) should state that the program has not allocated any memory, i.e. require that variable `MemUsed` is 0:

$$\texttt{requires MemUsed} == 0.$$

The normal postcondition of the method *m* must guarantee that the memory allocated during a normal execution of *m* is not more than some fixed number `methodConsumption(m)` of memory units. Thus for the method *m* the postcondition is:

$$\texttt{ensures MemUsed} \leq \texttt{old(MemUsed)} + \texttt{methodConsumption(}m\texttt{)}.$$

The exceptional postcondition of the method *m* must say that the memory allocated during an execution of *m* that terminates by throwing an exception `Exception` is not more than `methodConsumption(m)` units. Thus for the method *m* the exceptional postcondition is:

$$\texttt{exsures(Exception) MemUsed} \leq \texttt{old(MemUsed)} + \texttt{methodConsumption(}m\texttt{)}.$$

Loops must also be annotated with appropriate invariants. Let us assume that loop *l* iterates no more than $iter^l$ and let $loopConsumption(l)$ be an upper bound of the memory allocated per iteration in l. Below we give a general form of loop specification w.r.t. the property for constraint memory consumption. The loop invariant of a loop *l* states that at every iteration the loop body is not going to allocate more than $loopConsumption(l)$ memory units and that the iterations are no more than $iter^l$. We also declare an expression which guarantees loop termination, i.e. a variant (here an integer expression whose values decrease at every iteration and is always bigger or equal to 0):

$$\begin{array}{ll}
\texttt{modifies} & i, \texttt{MemUsed} \\
\texttt{invariant}: & \texttt{MemUsed} \leq \texttt{MemUsed}^{Before_l} + i * loopConsumption(l) \\
& \wedge \\
& i \leq iter^l \\
\texttt{variant}: & iter^l - i
\end{array}$$

A special variable appears in the invariant, $\texttt{MemUsed}^{Before_l}$. It denotes the value of the consumed memory just before entering for the first time the loop *l*. At every iteration the consumed memory must not go beyond the upper bound given for the body of loop.

For every instruction that allocates memory the ghost variable `MemUsed` must also be updated accordingly. For the purpose of this paper, we only consider dynamic object creation with the bytecode `new`; arrays are left for future work and briefly discussed in the conclusion.
The function `allocInstance` : $Class \rightarrow int$ gives an estimation of the memory used by an instance of a class. At every program point where a bytecode `new A` is found, the ghost variable `MemUsed` must be incremented by `allocInstance(A)`. This is achieved by inserting a ghost assignment immediately after any `new` instruction, as shown below:

```
new A
//set MemUsed = MemUsed+allocInstance(A).
```

3.2 Correctness

We want to guarantee that the memory allocated by a given program is bounded by a constant `Max`. We can prove that our annotation is correct w.r.t. to the policy for constraint memory use, by instrumenting the operational semantics of the bytecode language given in Section 2.4. The instrumented operational semantics manipulates states as before, but it is extended with the special variable `MemUsed`. Thus, states in the new semantics have the form:

$$\langle\!\langle h, \langle m, pc, l, s\rangle, sf, \texttt{MemUsed}\rangle\!\rangle.$$

The variable `MemUsed` changes its value only for instructions that allocate space in the heap, i.e. `new` instructions:

$$\frac{\text{InstAt}(m, pc) = \texttt{new}\ A,}{\begin{array}{c}\langle\!\langle h, \langle m, pc, l, v :: s\rangle, sf, \texttt{MemUsed}\rangle\!\rangle \rightarrow_{\texttt{new}\ A} \\ \langle\!\langle h + \texttt{allocInstance(A)}, \langle m, pc+1, l, s\rangle, sf, \texttt{MemUsed} + \texttt{allocInstance(A)}\rangle\!\rangle\end{array}}$$

The other instructions do not affect `MemUsed`, so the corresponding rules of the operational semantics are as before. As shown in the previous section, to every instruction of the form `new` A we attach the annotation `set MemUsed` $=$ `MemUsed` $+$ `allocInstance(A)`. The proof obligation generator converts this annotation into new value for the variable `MemUsed`:

$$\begin{array}{c}wp(\texttt{set MemUsed} = \texttt{MemUsed} + \texttt{allocInstance(A)}, \psi) = \\ \psi[\texttt{MemUsed} \leftarrow \texttt{MemUsed} + \texttt{allocInstance(A)}]\end{array}$$

We can prove that whenever the allocated space in the heap increments, the ghost variable `MemUsed` also increments, which is a sufficient condition to guarantee the correctness of the annotations. So far we do not deal with garbage collection (see discussion in Section 6).

3.3 Examples

We illustrate hereafter our approach by several examples, copying with recursive and overridden methods and with exceptions.

Inheritance and overridden methods Overriding methods are treated as follows: whenever a call is performed to a method *m*, we require that there is enough free memory space for the maximal consumption by all the methods that override or are overridden by *m*. In Fig. 2 we show a class `A` and its extending class `B`, where `B` overrides the method *m* from class `A`. Method *m* is invoked by n. Given that the dynamic type of the parameter passed to n is not known, we cannot know which of the two methods will be invoked. This is the reason for requiring enough memory space for the execution of any of these methods.

Recursive Methods In Fig. 3 the bytecode of the recursive method m and its specification is shown. For simplicity we show only a simplified version of the bytecode; we assume that the constructors for the class `A` and `C` do not allocate memory. Besides the precondition and the postcondition, the specification also includes information about the termination of the method: `variant localVar`(1), meaning that the local variable `localVar`(1) decreases on every recursive call down to and no more than 0, guaranteeing that the execution of the method will terminate.
We explain now the precondition. If the condition of line `1` is not true, the execution continues at line `2`. In the sequential execution up to line `7`, the program allocates at most `allocInstance(A)` memory

Specification of method m in class A:

$$\begin{array}{l}\texttt{requires}\ \texttt{MemUsed} + k \leq \texttt{Max}\\ \texttt{modifies}\ \texttt{MemUsed}\\ \texttt{ensures}\ \ \texttt{MemUsed} \leq \texttt{old}(\texttt{MemUsed}) + k\end{array}$$

Specification for method m in class B:

$$\begin{array}{l}\texttt{requires}\ \texttt{MemUsed} + l \leq \texttt{Max}\\ \texttt{modifies}\ \texttt{MemUsed}\\ \texttt{ensures}\ \ \texttt{MemUsed} \leq \texttt{old}(\texttt{MemUsed}) + l\end{array}$$

```
method n(A a)
...
//{ prove Mem <= Mem +max(l,k) }
invokevirtual m <A>
//{ assume Mem <= \old(Mem) + max(l,k)}
...
```

Fig. 2. EXAMPLE OF OVERRIDDEN METHODS

units and decrements by 1 the value of
`localVar`(1). The instruction at line 8 is a recursive call to m, which either will take the same branch if `localVar`$(1) > 0$ or will jump to line 12 otherwise, where it allocates at most `allocInstance(A)`$+$ `allocInstance(C)` memory units. On returning from the recursive call one more allocation will be performed at line 9. Thus m will execute, `localVar`(1) times, the instructions from lines 2 to 7, and it finally will execute all the instructions from lines 12 to 16.

The postcondition states that the method will perform no more than `old`$($`localVar`$(1))$ recursive calls (i.e., the value of the register variable in the pre-state of the method) and that on every recursive call it allocates no more than two instances of class A (one corresponding to line 2 and the other to line 9) and that it will finally allocate one instance of class A (line 12) and another of class C (line 14).

To give an idea about the order of complexity for proving the correctness of this method, 18 proof obligations were generated with Jack. The proof obligations were automatically proved in Coq using its standard tactics.

More precise specification We can be more precise in specifying the precondition of a method by considering what are the field values of an instance, for example. Suppose that we have the method *m* as shown in Fig. 4. We assume that in the constructor of the class A no allocations are done. The first line of the method *m* initializes one of the fields of field b. Since nothing guarantees that field b is not `null`, the execution may terminate with `NullPointerException`. Depending on the values of the parameters passed to *m*, the memory allocated will be different. The precondition establishes what is the expected space of free resources depending on if the field b is `null` or not. In particular we do not require anything for the free memory space in the case when b is `null`. In the normal postcondition we state that the method has allocated an object of class A. The exceptional postcondition states that no allocation is performed if `NullpointerException` causes the execution termination.

```
public class D {
  public void m( int i) {
    if (i > 0) {
      new A();
      m(i - 1);
      new A();
    } else {
      new C();
      new A();
   }
  }
}
```

```
requires (MemUsed + localVar(1) * 2 * allocInstance(A)+
         allocInstance(A) + allocInstance(C)) ≤ Max
variant  localVar(1)
ensures  localVar(1) ≥ 0
         ∧
         MemUsed <= old(MemUsed) + old(localVar(1)) * 2 * allocInstance(A) + allocInstance(A)
         +allocInstance(C))
```

```
public void m()
//local variable loaded on
//the operand stack of method m
0 load_1
//  if localVar(1) <= 0 jump
1 ifle 12
2 new <A> //  here localVar(1) > 0
//set MemUsed = MemUsed +  allocInstance(A)
3 invokespecial <A.<init>>
4 aload_0
5 iload_1
6 iconst_1
//localVar(1) decremented with 1
7 isub
//  recursive call with the new value of localVar(1)
8 invokevirtual <D.m>//
9 new <A>
//set MemUsed = MemUsed +  allocInstance(A)
10 invokespecial <A.<init>>
11 goto 16
//target of the jump at 1
12 new <A>
//set MemUsed = MemUsed +  allocInstance(A)
13 invokespecial <A.<init>>
14 new  <C>
//set MemUsed = MemUsed +  allocInstance(C)
15 invokespecial <C.<init>>
16 return
```

Fig. 3. EXAMPLE OF A RECURSIVE METHOD

4 Inferring memory allocation for methods

In the previous section, we have described how the memory consumption of a program can be modeled in BML and verified using an appropriate verification environment. While our examples illustrate the benefits of our approach, especially regarding the precision of the analysis, the applicability of our method is hampered by the cost of providing the annotations manually. In order to reduce the burden of manually annotating the program, one can rely on annotation assistants that infer automatically some of the program annotations (indeed such assistants already exist for loop invariants, loop variants, or class invariants). In this section, we describe an implementation of an annotation assistant dedicated to the analysis of memory consumption, and illustrate its functioning on an example.

4.1 Annotation assistant

The user must provide annotations about the memory required to create objects of the given classes. The variants for each loop may be given by the user or be synthesized through appropriate mechanisms. Based on this information, the annotation assistant inserts the ghost assignments on appropriate places, and then computes recursively the memory allocated on each loop and method. A pseudo-code of the algorithm for inferring an upper bound for method allocations is given in Fig. 5. Essentially, it finds the maximal memory that can be allocated in a method by exploring all its possible execution path. The algorithm computes the set of blocks contained in a loop, the loop entry block and the set of end blocks of a loop; see Section 10 of [5] for a description of the algorithms used.

The auxiliary function $allocPath(\cdot)$, which infers the maximal allocations done by the set of execution paths ending with the same `return` instruction, is given in Fig. 6. Inferring the memory allocated inside loops is done by the function $alloc\_loop\_path(\cdot,\cdot)$, which is invoked by $allocPath$ whenever the current instruction belong to a loop. The specification of the function is shown in Fig. 7.

The annotation assistant currently synthesize only simple memory policies (i.e., whenever the memory consumption policy does not depend on the values of inputs). Furthermore, it does not deal with arrays, subroutines, nor exceptions. Our approach may be extended to treat such cases (see the discussion in Section 6 about how to include arrays in our analysis). For sake of simplicity, we have also restricted

```
   requires                        localVar(1)! = null ⇒
                                              MemUsed + allocInstance(A) ≤ Max
   modifies                        MemUsed
   ensures                         MemUsed ≤ old(MemUsed) + allocInstance(A)
   exsures(NullPointerException)   MemUsed == old(MemUsed)
```

```
0 aload_0                          public class C {
1 getfield<C.b>                      B b;
2 iload_2                            public void m(A a, int i) {
3 putfield <B.i>                       b.i = i ;
4 new <A>                              a = new A();
//set MemUsed = MemUsed +             }
      allocInstance(A)               }
5 dup
6 invokespecial <A.<init>>
7 astore_1
8 return
```

Fig. 4. EXAMPLE OF A METHOD WITH POSSIBLE EXCEPTIONAL TERMINATION

function `methodConsumption(.)`
Input: Bytecode of a method m.
Output: Upper bound of the memory allocated by m.
Body:

1. Detect all the loops in m;
2. For every loop l determine $loopSet(l)$, $entry(l)$ and $loopEndSet()$;
3. Apply the function $alloc$ to each instruction i_k, such that $i_k =$ `return`;
4. Take the maximum of the results given in the previous step: $max_{i_k=\texttt{return}} allocPath(i_k)$.

Fig. 5. INFERENCE ALGORITHM

$allocPath(i_s) =$

$$\begin{cases} alloc\_instr(i_s) & if\ i_s\ has\ no\ predecessors \\ \\ loopConsumption(entry(l)) & \\ + & if\ i_s \in loopSet(l) \\ max_{i_k \in preds(i_s) - loopEndSet(l)}(allocPath(i_k)) & \\ \\ alloc\_instr(i_s) & \\ + & else \\ max_{i_k \in preds(i_s)}(allocPath(i_k)) & \end{cases}$$

Fig. 6. DEFINITION OF THE FUNCTION $allocPath(i_s)$

$alloc\_loop\_path(entry(l), i_s) =$

$$\begin{cases} alloc\_instr(entry(l)) & if\ i_s = entry(l) \\ \\ loopConsumption(entry(l')) & \\ + & if\ i_s \in loopSet(l') \\ max_{i_k \in preds(entry(l')) - loopEndSet(l')}(alloc\_loop\_path(entry(l), i_k)) & l'\ is\ nested\ in\ l \\ \\ alloc\_instr(i_s) & \\ + & else \\ max_{i_k \in preds(i_s)}(alloc\_loop\_path(entry(l), i_k)) & \end{cases}$$

Fig. 7. DEFINITION OF THE FUNCTION $alloc\_loop\_path(entry(l), i_s)$

the loop analysis only to those with a unique entry point, which is the case for code produced by non-optimizing compilers. A pre-analysis could give us all the entry points of more general loops, for instance by the algorithms given in [9]; our approach may be thus applied straightforwardly.

4.2 Example

Let us consider the bytecode given in Fig. 8, which is a simplified version of the bytecode corresponding to the source code given in the right of the figure. For simplicity of presentation, we do not

show all the instructions (the result of the inference procedure is not affected). Method m has two branching instructions, where two objects are created: one instance of class A and another of class B. Our inference algorithm gives that `methodConsumption(m)` $=$ `allocInstance(A)` $+$ `methodConsumption(A.init)`$+$`allocInstance(B)`$+$`methodConsumption(B.init)`. Due to limitation on space, we do not explain the details of such inference, which is given in Fig. 9 (i_k refers to the bytecode instruction at position k).

```
0 aload_1                          public void m(A a , B b ) {
1 ifnonnull 6                        if ( a == null ) {
2 new <A>                              a = new A();
...                                    }
4 invokespecial <A.<init>>           if (b == null) {
6 aload_2                              b = new B();
7 ifnonnull 12                       }
8 new <B>                          }
...
10 invokespecial <B.<init>>
...
12 return
```

Fig. 8. EXAMPLE

5 Related work

The use of type systems has been a useful tool for guaranteeing that well typed programs run within stated space-bounds. Previous work along these lines defined typed assembly languages, inspired on [22] while others emphasised the use of type systems for functional languages [4, 16, 18].

For instance in [2] the authors present a first-order linearly typed assembly language which allows the safe reuse of heap space for elements of different types. The idea is to design a family of assembly languages which have high-level typing features (e.g. the use of a special *diamond* resource type) which are used to express resource bound constraints. Closely related to the previous-mentioned paper, [30] describes a type theory for certified code, in which type safety guarantees cooperation with a mechanism to limit the CPU usage of untrusted code. Another recent work is [1] where the resource bounds problem is studied in a simple stack machine. The authors show how to perform type, size and termination verifications at the level of the byte-code.

An automatic heap space usage static analysis for first-order functional programs is given in [17]. The analysis both determines the amount of free cells necessary before execution as well as a safe (under)-estimate of the size of a *free-list* after successful execution of a function. These numbers are obtained as solutions to a set of linear programming (LP) constraints derived from the program text. Automatic inference is obtained by using standard polynomial-time algorithms for solving LP constraints. The correctness of the analysis is proved with respect to an operational semantics that explicitly keeps track of the memory structure and the number of free cells.

A logic for reasoning about resource consumption certificates of higher-order functions is defined in [12]. The certificate of a function provides an over-approximation of the execution time of a call to

$$
\begin{aligned}
&\texttt{methodConsumption(m)}\\
&=\\
&allocPath(i_{12})\\
&= max_{i_k \in preds(1_{12})}(allocPath(i_k)) + alloc\_instr(i_{12})\\
&\{alloc\_instr(i_{12}) = 0, preds(i_{i_{12}}) = \{i_{10}, i_7\}\}\\
&= max(allocPath(i_{10}), allocPath(i_7))\\
&= max(max_{i_k \in preds(i_{10})}(allocPath(i_k)) + alloc\_instr(i_{10}),\\
&\qquad max_{i_k \in preds(i_7)}(allocPath(i_k)) + alloc\_instr(i_7)\\
&\quad)\\
&\{preds(i_{10}) = \{i_8\}, preds(i_7) = \{i_6\}\}\\
&= max(allocPath(i_8) + alloc\_instr(i_{10}), allocPath(i_6) + alloc\_instr(i_7))\\
&\{alloc\_instr(i_{10}) = \texttt{methodConsumption(B.init)}, alloc\_instr(i_7) = 0\}\\
&= max(max_{i_k \in preds(i_8)}(allocPath(i_k)) + alloc\_instr(i_8) + \texttt{methodConsumption(B.init)},\\
&\qquad max_{i_k \in preds(i_6)}(allocPath(i_k)) + alloc\_instr(i_6)\\
&\quad)\\
&\{preds(i_8) = \{i_7\}, preds(i_6) = \{i_4\}\}\\
&= max(allocPath(i_7) + +alloc\_instr(i_8) + \texttt{methodConsumption(B.init)},\\
&\qquad allocPath(i_4) + alloc\_instr(i_6)\\
&\quad)\\
&\{alloc\_instr(i_8) = \texttt{allocInstance(B)}, alloc\_instr(i_6) = 0\}\\
&= max(max_{i_k \in preds(i_7)}(allocPath(i_k)) + alloc\_instr(i_7) + \texttt{allocInstance(B)} + \texttt{methodConsumption(B.init)},\\
&\qquad max_{i_k \in preds(i_4)}(allocPath(i_k)) + alloc\_instr(i_4)\\
&\quad)\\
&\{preds(i_7) = \{i_6\}, preds(i_4) = \{i_2\}\}\\
&= max(allocPath(i_6) + alloc\_instr(i_7) + \texttt{allocInstance(B)} + \texttt{methodConsumption(B.init)},\\
&\qquad allocPath(i_2) + alloc\_instr(i_4)\\
&\quad)\\
&\{alloc\_instr(i_7) = 0, alloc\_instr(i_4) = \texttt{methodConsumption(A.init)}\}\\
&= max(max_{i_k \in preds(i_6)}(allocPath(i_k)) + alloc\_instr(i_6) + \texttt{allocInstance(B)} + \texttt{methodConsumption(B.init)},\\
&\qquad max_{i_k \in preds(i_2)}(allocPath(i_k) + alloc\_instr(i_2) + \texttt{methodConsumption(A.init)}\\
&\quad)\\
&= \{preds(i_6) = \{i_4, i_1\}, preds(i_2) = \{i_1\}\}\\
&= max(max(allocPath(i_4), allocPath(i_1)) + alloc\_instr(i_6) + \texttt{allocInstance(B)} + \texttt{methodConsumption(B.init)},\\
&\qquad allocPath(i_1) + alloc\_instr(i_2) + \texttt{methodConsumption(A.init)}\\
&\quad)\\
&\{alloc\_instr(i_6) = 0, alloc\_instr(i_2) = \texttt{allocInstance(A)}\}\\
&= max(max(max_{i_k \in preds(i_4)}(allocPath(i_k)) + alloc\_instr(i_4),\\
&\qquad\quad max_{i_k \in preds(i_1)}(allocPath(i_k)) + alloc\_instr(i_1)\\
&\qquad) + \texttt{allocInstance(B)} + \texttt{methodConsumption(B.init)},\\
&\qquad allocPath(i_0) + \texttt{allocInstance(A)} + \texttt{methodConsumption(A.init)},\\
&\quad)\\
&\{preds(i_4) = \{i_2\}, preds(i_1) = \{i_0\}\}\\
&\{allocPath(i_0) = alloc\_instr(i_0) = 0, alloc\_instr(i_1) = 0, alloc\_instr(i_4) = \texttt{methodConsumption(A.init)}\}\\
&= max(max(allocPath(i_2) + \texttt{methodConsumption(A.init)},\\
&\qquad\quad allocPath(i_0)\\
&\qquad) + \texttt{allocInstance(B)} + \texttt{methodConsumption(B.init)},\\
&\qquad \texttt{allocInstance(A)} + \texttt{methodConsumption(A.init)},\\
&\quad)\\
&\{allocPath(i_0) = alloc\_instr(i_0) = 0\}\\
&= max(allocPath(i_2) + \texttt{methodConsumption(A.init)} + \texttt{allocInstance(B)} + \texttt{methodConsumption(B.init)},\\
&\qquad \texttt{allocInstance(A)} + \texttt{methodConsumption(A.init)},\\
&\quad)\\
&\dots\\
&= max(\texttt{allocInstance(A)} + \texttt{methodConsumption(A.init)} + \texttt{allocInstance(B)} + \texttt{methodConsumption(B.init)}\\
&\qquad \texttt{allocInstance(A)} + \texttt{methodConsumption(A.init)},\\
&\quad)\\
&= \texttt{allocInstance(A)} + \texttt{methodConsumption(A.init)} + \texttt{allocInstance(B)} + \texttt{methodConsumption(B.init)}
\end{aligned}
$$

Fig. 9. INFERENCE OF THE MEMORY ALLOCATED BY THE METHOD m OF FIG. 8

the function. The logic only defines what is a correct deduction of a certificate and has no inference algorithm associated with it. Although the logic is about computation time the authors claim it could be extended to measure memory consumption.
Another mechanical verification of a byte code language is [9], where a constraint-based algorithm is presented to check the existence of `new` instructions inside intra- and inter-procedural loops. It is completely formalised in Coq and a certified analyser is obtained using Coq's extraction mechanism. The time complexity of such analysis performs quite good but the auxiliary memory used does not allow it to be on-card. Their analysis is less precise than ours, since they work on an abstraction of the execution traces not considering the number of times a cycle is iterated (there are no annotations). Along these lines, a similar approach has been followed by [26]; no mechanical proof nor implementation is provided in such work.
Other related research direction concerns runtime memory analysis. The work [15] presents a method for analysing, monitoring and controlling dynamic memory allocation, using pointer and scope analysis. By instrumenting the source code they control memory allocation at run-time. In order to guarantee the desired memory allocation property, in [13] is implemented a runtime monitor to control the execution of a Java Card applet. The applet code is instrumented: a call to a monitor method is added before a `new` instruction. Such monitor method has as parameter the size of the allocation request and it halts the execution of the applet if a predefined allocation bound is exceeded.
There exists research on the definition of bytecode logics. In [24] a Hoare logics for bytecode is defined. Yet the approach there is based on searching structure in the bytecode programs which is not very natural for untsructured bytecode programs. In [31] a Hoare bytecode logics is defined in terms of weakest precondition calculus over the Jinja language (subset of Java). They use the logics for verifying bytecode against arithmetic overflow.
Upon completion of this work we became aware of a recent, and still unpublished, result along the same lines of ours. Indeed, a hybrid (i.e., static and dynamic) resource bound checker for an imperative language designed to admit decidable verification is presented in [10]. The verifier is based on a variant of Dijkstra's weakest precondition calculus using "generalized predicates", which keeps track of the resource units available. Besides adding loop invariants, pre- and post-conditions, the programmer must insert "acquires" annotations to reserve the resource units to be consumed. Our approach has the advantage of treating recursive methods and exceptions, not taken into account in [10]. Another difference with our work is that we operate on the bytecode instead of on the source code.

6 Conclusion

Program logics have traditionally been used to verify functional properties of applications, but we have shown that such logics are also appropriate to enforce security properties including memory consumption policies. We have shown that program logics complement nicely existing methods to verify memory consumption, over which they are superior in terms of the precision of the analysis (and inferior in terms of automation).
We intend to pursue our work in three directions. Firstly, we would like to extend our approach to arrays. In principle, it should be reasonably easy to extend the verification method to arrays; however, it seems more complicated to extend our inference algorithm to arrays. The main difficulty here is to provide an estimate of the size of an array, as it is given by the top value on the operand stack at the time of its creation. Our intuition is that this can be done using an abstract interpretation or a symbolic evaluation of the program. If we look at the example code below (in source code):

```
void m(int s){
 int len = s;
```

```
 int[] i = new int[len]
}
```

where `len` is a local variable to the method, one can infer by symbolic computation that its value is the value of the method parameter. Thus the method can be given the precondition `Mem + s <= Max`.
In a similar line of work, we would like to extend our results to concurrency using recent advances in program logics for multi-threaded Java programs [25]. Providing an appropriate treatment of arrays and multi-threading is an important step towards applying our results to mobile phone applications.
Secondly, we would like to adapt our approach to account for explicit memory management. More precisely, we would like to consider an extended language with a special instruction `free(o)` that deallocates the object `o`, and establish the correctness of our method under the assumption that deallocation is correct, i.e. that the object `o` is not reachable from the program point where `free(o)` is inserted. By combining our approach with existing compile-time analysis that infers for each program point which objects are not reachable, we should be able to provide more precise estimates of memory consumption.
Thirdly, we intend to apply our technique to other resources such as communication channels, bandwidth, and power consumption, as well as to more refined analyses that distinguish between different kinds of memory, such as RAM or non-volatile EEPROM. As suggested by the MRG project [3], it seems also interesting to consider policies that enforce limits on the interaction between the program and its environment, for example w.r.t. the number of system calls or the bounds on parameters passed to them.

References

1. R.M. Amadio, S. Coupet-Grimal, S. Dal Zilio, and L. Jakubiec. A Functional Scenario for Bytecode Verification of Resource Bounds. Research report 17-2004, LIF, Marseille, France, 2004.
2. D. Aspinall and A. Compagnoni. Heap-bounded assembly language. *J. Autom. Reason.*, 31(3-4):261–302, 2003.
3. D. Aspinall, S. Gilmore, M. Hofmann, D. Sannella, and I. Stark. Mobile Resource Guarantees for Smart Devices. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings of CASSIS'04*, volume 3362 of *LNCS*, pages 1–27, 2005.
4. D. Aspinall and M. Hofmann. Another type system for in-place update. In *ESOP*, volume 2305 of *LNCS*, pages 36–52, 2002.
5. J. Ullman A.V. Aho, R. Sethi. *Compilers Principles, Techniques and Tools*. Addison-Wesley Publiching Company, 1986.
6. G. Barthe, P. D'Argenio, and T. Rezk. Secure Information Flow by Self-Composition. In R. Foccardi, editor, *Proceedings of CSFW'04*, pages 100–114. IEEE Press, 2004.
7. L. Burdy and M. Pavlova. Annotation carrying code. Manuscript, 2004.
8. L. Burdy, A. Requet, and J.-L. Lanet. Java Applet Correctness: a Developer-Oriented Approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proceedings of FME'03*, number 2805 in LNCS, pages 422–439. Springer, 2003.
9. D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified memory usage analysis. Submitted, 2005.
10. A. Chander, D. Espinosa, N. Islam, P. Lee, and G. Necula. Enforcing resource bounds via static verification of dynamic checks. To appear, 2005.
11. Coq Development Team. *The Coq Proof Assistant User's Guide. Version 8.0*, January 2004.
12. K. Crary and S. Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–198. ACM Press, 2000.

13. L.-A. Fredlund. Guaranteeing correctness properties of a java card applet. In *RV'04*, ENTCS, 2004.
14. S. N. Freund and J. C. Mitchell. A Type System for the Java Bytecode Language and Verifier. *Journal of Automated Reasoning*, 30(3-4):271–321, December 2003.
15. D. Garbervetsky, C. Nakhli, S. Yovine, and H. Zorgati. Program instrumentation and run-time analysis of scoped memory in java. In *RV'04*, ENCS, 2004.
16. M. Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.
17. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. of 30th ACM Symp. on Principles of Programming Languages (POPL'03)*, pages 185–197. ACM Press, 2003.
18. J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *International Conference on Functional Programming*, pages 70–81, 1999.
19. G. Klein and T. Nipkow. Verified bytecode verifiers. *Theoretical Computer Science*, 298(3):583–626, April 2002.
20. G.T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*.
21. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. 1999.
22. G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.
23. M. Pavlova, G. Barthe, L. Burdy, M. Huisman, and J.-L. Lanet. Enforcing high-level security properties for applets. In P. Paradinas and J.-J. Quisquater, editors, *Proceedings of CARDIS'04*. Kluwer, 2004.
24. C.L. Quigley. A programming logic for java bytecode programs. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.
25. E. Rodrıguez, M.B. Dwyer, C. Flanagan, J. Hatcliff, G.T. Leavens, and Robby. Extending sequential specification techniques for modular specification and verification of multi-threaded programs. In *ECOOP 2005*, 2005. To appear.
26. G. Schneider. A constraint-based algorithm for analysing memory usage on java cards. Technical Report RR-5440, INRIA, December 2004.
27. N. Shankar, S. Owre, and J.M. Rushby. *The PVS Proof Checker: A Reference Manual*. Computer Science Laboratory, SRI International, February 1993. Supplemented with the PVS2 Quick Reference Manual, 1997.
28. I. Siveroni and C. Hankin. A proposal for the jcvmle operational semantics, 2001.
29. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine - Definition, Verification, Validation*. Springer, 2001.
30. J. C. Vanderwaart and K. Crary. Foundational typed assembly language for grid computing. Technical Report CMU-CS-04-104, CMU, February 2004.
31. M. Wildmoser and T. Nipkow. Asserting bytecode safety. In *Proceedings of the 15th European Symposium on Programming (ESOP05)*, 2005. To appear.