Inferring Invariants by Symbolic Execution

Peter H. Schmitt and Benjamin Weiß

University of Karlsruhe
Institute for Theoretical Computer Science
D-76128 Karlsruhe, Germany
{pschmitt,bweiss}@ira.uka.de

Abstract. In this paper we propose a method for inferring invariants for loops in JAVA programs. An example of a simple while loop is used throughout the paper to explain our approach. The method is based on a combination of symbolic execution and computing fixed points via predicate abstraction. It reuses the axiomatisation of the JAVA semantics of the KeY system. The method has been implemented within the KeY system which allows to infer invariants and perform verification within the same environment. We present in detail the results of a non-trivial example.

1 Introduction

A notorious difficulty in the formal verification of programs is the treatment of loop constructs. An array of techniques has been developed to address this problem. Among these techniques the use of loop invariants is particularly attractive since it does not compromise on the rigour of verification and runs completely automatically, once the correct invariants are provided. In this paper we will try to answer the question how to find invariants.

Several techniques for automatically inferring invariants of a program exist. One general approach is *dynamic analysis*, i.e., analysing the program by executing it with concrete input values. A tool implementing dynamic invariant inference is Daikon [10]: to infer invariants for a program, Daikon first instruments it with state saving code at interesting program points. The instrumented program is then run through a user-specified test suite. Finally, the resulting data base of program states is analysed for properties which held in all of the test runs. These properties are somewhat likely to be invariants, but this is not guaranteed, because the test suite in general cannot cover all cases.

Stronger guarantees can be provided by *static analysis*, i.e., analysing the program by examining its source code without actually executing it on concrete inputs. A common paradigm in static analysis, which is also used in program verification, is *symbolic execution* [16]: the analysed program is "executed", but with symbolic instead of concrete values for the program variables. Static invariant inference techniques are usually based on *abstract interpretation* [7]. Abstract interpretation can be understood as an approximative ("abstract") symbolic execution of the program, which deals with loops through fixed-point iteration.

Termination of this fixed-point iteration is ensured by the approximative nature of the used symbolic execution. An example of a tool which uses abstract interpretation for invariant inference is the static verifier Boogie [17].

Predicate abstraction [12] is a special variant of abstract interpretation, which has been used for invariant inference [11]. Here, the symbolic execution is itself not approximative, but as precise as possible (which corresponds to computing strongest postconditions). Instead, the necessary approximation is performed by explicit "abstraction" operations, which make use of an arbitrary, finite set of predicates over the variables of the program. The inferred invariants are constructed from these predicates. Thus, the problem of finding invariants is reduced to the simpler problem of guessing potentially useful predicates, which can be done heuristically or, when necessary, manually by the user.

In the invariant inference method described in [11], the semantics of the programming language is implicitly incorporated in the algorithms of the system. In our approach we start from the axiomatic semantics for Java Card developed within the KeY project. Java Card [15] is roughly a subset of Java which contains all object-oriented features but lacks concurrency, floating-point arithmetic, and dynamic class loading. The KeY system [1, 3] is a deductive verification system for Java Card programs. It is based on an axiomatisation of the Java Card semantics within a program logic calculus, which follows the symbolic execution paradigm. The axiomatisation covers 100% of the Java Card language specification and a bit more with great precision. It has so far mainly been used for program verification, e.g., the Demoney case study, an electronic purse application provided by Trusted Logic S.A. [3, Chapt. 14], the verification of a Java Card implementation of the Schorr-Waite graph marking algorithm [3, Chapt. 15], and the verification of a part of a flight management system from Thales Avionics [14]. The most recent targets of verification with KeY have been an implementation of the Mondex banking card case study [23, 22] and an implementation of the Java Card API [19]. The KeY symbolic execution rules for Java Card have lately also been used for model-based test generation [2, 9]. In this paper we explore the possibility to use the KeY calculus and prover for inferring invariants, by incorporating fixed-point iteration and predicate abstraction. Besides the obvious benefit of reusing an existing formal semantics, this also has the advantage that the generation of loop invariants is an integrated part of the verification effort.

The organisation of this paper is as follows: in Sect. 2, we review our program logic for Java Card and its axiomatisation of the programming language semantics. In Sect. 3, we introduce a simple example for a program and its invariants. Using this example, we then explain our approach for inferring invariants in Sect. 4. The implementation of the approach within the KeY system is sketched in Sect. 5, and the results of initial experiments with the implementation are documented in Sect. 6. Finally, we conclude in Sect. 7.

2 Background

Our approach is based on the program logic Java Card DL [3, Chapt. 3], which is a version of dynamic logic [13]. It extends first-order predicate logic by modal operators $[\mathtt{p}]$ ("box") and $\langle \mathtt{p} \rangle$ ("diamond") for every legal sequence of Java Card statements p: the formula $[\mathtt{p}]\psi$ states that if execution of p terminates in a state s, then ψ holds in state s; the formula $\langle \mathtt{p} \rangle \psi$ additionally requires that p does indeed terminate. In the following we only make use of the box modality $[\mathtt{p}]$. The reason is that in this paper, we are interested in invariants, and invariants are not concerned with the issue of termination: they are safety properties, whereas termination is a liveness property.

Formulas of the form $\varphi \rightarrow [\mathtt{p}]\psi$ are similar to Hoare triples $\{\varphi\}\mathtt{p}\{\psi\}$: they express that if p terminates after being started in a state which satisfies φ, then the resulting state satisfies ψ. For example, the meaning of the formula $\mathtt{o.f} \doteq 27 \rightarrow [\mathtt{o.f++;}]\mathtt{o.f} \doteq 28$ is: if the current value of the field f of the object pointed to by the program variable o is 27, then after executing the statement `o.f++;` the value of the field has changed to 28. This formula is *valid*, i.e., it holds in all possible states.

Proofs of the validity of Java Card DL formulas can be performed by means of a sequent calculus. A sequent is a construct $\Gamma \vdash \Delta$, where Γ and Δ are sets of formulas. Its semantics is the same as that of the formula $\bigwedge \Gamma \rightarrow \bigvee \Delta$, and in the following we do not strictly distinguish between sequents and their equivalent formulas. An example for a sequent calculus rule is andRight:

$$\frac{\Gamma \vdash \varphi,\ \Delta \qquad \Gamma \vdash \varphi',\ \Delta}{\Gamma \vdash \varphi \wedge \varphi',\ \Delta}\ \mathsf{andRight}$$

The (schematic) sequents $\Gamma \vdash \varphi, \Delta$ and $\Gamma \vdash \varphi', \Delta$ are the *premisses* of the rule, and the sequent $\Gamma \vdash \varphi \wedge \varphi', \Delta$ is the *conclusion* of the rule. A rule is *sound* if validity of its premisses implies validity of its conclusion. A proof for a sequent is constructed by applying rules from bottom to top; if all leaves of the resulting tree are valid, then the root sequent must be valid as well. The particular rule andRight deals with a conjunction on the right side of the sequent arrow by splitting the proof tree into two branches.

Formulas with modal operators are handled by rules which perform a symbolic execution of the Java Card program within the modality. These rules operate on the *active statement* of the program, i.e., the first basic statement following a non-active prefix of opening braces, beginnings of `try` blocks and the like. This prefix is denoted by π, and the rest of the program behind the active statement by ω. For example, the active statement of the following program is `i = 0`:

$$\underbrace{\mathtt{\{\ try\ \{}}_{\pi}\ \mathtt{i\ =\ 0;}\ \underbrace{\mathtt{i{+}{+};\ \}\ catch(Exception\ e)\ \{\ i\ =\ 27;\ \}\ \}}}_{\omega}$$

The Java Card DL calculus, as it is implemented in the KeY system, currently contains approximately 1700 rules, of which about 1300 formalise the semantics of Java Card. As we cannot describe all of them here, we restrict our presentation to representative rules for the three basic programming constructs in imperative languages: assignments, conditional statements, and loops. We begin with assignments, which can be symbolically executed with

$$\frac{\Gamma',\ \mathtt{x} \doteq \mathtt{e}' \ \vdash\ [\pi\, \omega]\psi,\ \Delta'}{\Gamma\ \vdash\ [\pi\ \mathtt{x\ =\ e;}\ \omega]\psi,\ \Delta}\ \mathsf{assignment}$$

where the expression $\mathtt{e}$ must not have side effects, and where Γ', $\mathtt{e}'$ and Δ' result from Γ, $\mathtt{e}$ and Δ, respectively, by substituting a fresh program variable x' for $\mathtt{x}$. This rule replaces the assumptions about the initial state of the program by their strongest postcondition under the assignment statement. For example, it transforms the sequent $\mathtt{i} \doteq 27 \vdash [\mathtt{i=i+1;}]\mathtt{i} \doteq 28$ into $i' \doteq 27, \mathtt{i} \doteq i'+1 \vdash \mathtt{i} \doteq 28$.

As formulated here, the assignment rule only works for assignments to local program variables. Assignments to fields or array slots are more complex because of *aliasing*, i.e., the phenomenon that the same memory location may be referred to by different names. They can nevertheless be handled along the same lines as assignments to local variables, but this leads to somewhat complicated formulas containing case distinctions for the possible aliasing situations. For example, symbolically executing $\mathtt{o1.f} \doteq 27 \vdash [\mathtt{o2.f} = 0;]\mathtt{o1.f} \doteq 27$ in this way yields $\mathtt{o1.}f' \doteq 27, \forall x.(x\mathtt{.f} \doteq \mathit{if}(x \doteq \mathtt{o2})\mathit{then}(0)\mathit{else}(x\mathtt{.}f')) \vdash \mathtt{o1.f} \doteq 27$. The KeY system normally avoids these complications as far as possible by treating assignments in a different way, which is based on a concept called *updates* [21]. However, for the purpose of inferring invariants, the classical way to handle assignments fits our needs better. The details of how this works out for complex assignments are given in [24]. In the following we restrict ourselves to the simple case of assignments to local variables, which amply suffices to explain our method.

Conditional statements can be handled with this rule:

$$\frac{\Gamma\ \vdash\ (\mathtt{e} \doteq \mathtt{true} \rightarrow [\pi\ \mathtt{p}\ \omega]\psi) \wedge (\neg\mathtt{e} \doteq \mathtt{true} \rightarrow [\pi\ \mathtt{q}\ \omega]\psi),\ \Delta}{\Gamma\ \vdash\ [\pi\ \mathtt{if(e)\ p\ else\ q}\ \omega]\psi,\ \Delta}\ \mathsf{ifElse}$$

Again, the occurring Java Card expression must not have side effects. This restriction is never severe, because a program can always be transformed such that an expression is separated from its side effects; the Java Card DL calculus contains rules which perform such transformations. The ifElse rule symbolically executes a conditional statement by creating two conjuncts which describe the case that the guard expression is true and the case that it is false, respectively. Typically, the next step is to split the proof tree by applying the andRight rule.

Loops can be symbolically executed with the loopUnwind rule

$$\frac{\Gamma\ \vdash\ [\pi\ \mathtt{if(e)\{p\ while(e)\ p\}}\ \omega]\psi,\ \Delta}{\Gamma\ \vdash\ [\pi\ \mathtt{while(e)\ p}\ \omega]\psi,\ \Delta}\ \mathsf{loopUnwind}$$

where, as usual, the expression `e` must not have side effects. This is a simplified version of the actual rule which is sound only if the loop body does not contain `break` or `continue` statements. It transforms the loop into a conditional statement: if the guard expression is satisfied, then the loop body is executed once before getting to the loop again; otherwise, the loop is not entered. Since its premiss again contains the loop, the unwind rule on its own only works for loops which terminate after a statically known and sufficiently small number of iterations. In the general case, loops cannot be handled by symbolic execution alone. Instead, an *invariant rule* can be used, i.e., a rule which makes use of a loop invariant. This loop invariant normally has to be provided manually from the outside.

3 Running Example

As a simple example for a program and its invariants, we will use the following piece of Java Card code, which computes the maximal positive element of an integer array `a`:

```
max = 0;
i = 0;
while(i < a.length) {
  if(a[i] > max) max = a[i];
  i++;
}
```

The program is visualised as a control flow graph in Fig. 1. The nodes of this graph represent the basic commands and guard expressions of the program, and the edges stand for flow of control between the nodes. Control enters the program through the node marked `entry`, and leaves it through the node marked `exit` (for the sake of readability, we ignore here that the program terminates abruptly if $\texttt{a} \doteq \texttt{null}$ holds). The control flow graph is annotated with exemplary invariants at interesting program points. Note that in this paper we are talking about invariants in the classical sense, i.e., first-order formulas which always hold when control flow reaches a specific program point such as a loop entry. The notion of "class" or "object" invariants for object-oriented programs [18] is related but lives at a different level of abstraction.

Our example invariants for the program are as follows: at the `entry` node, we assume nothing about the program state, so the invariant here is *true*. After the first assignment statement, $\texttt{max} \doteq 0$ always holds–this is the strongest postcondition of *true* under the assignment statement. Next is the loop invariant: every time control flow reaches the loop entry, $\forall x.(0 \leq x < \texttt{i} \rightarrow \texttt{a[}x\texttt{]} \leq \texttt{max})$ is satisfied. Unlike the loop invariant, the remaining invariants can again easily be derived from their predecessors as strongest postconditions. In particular, the

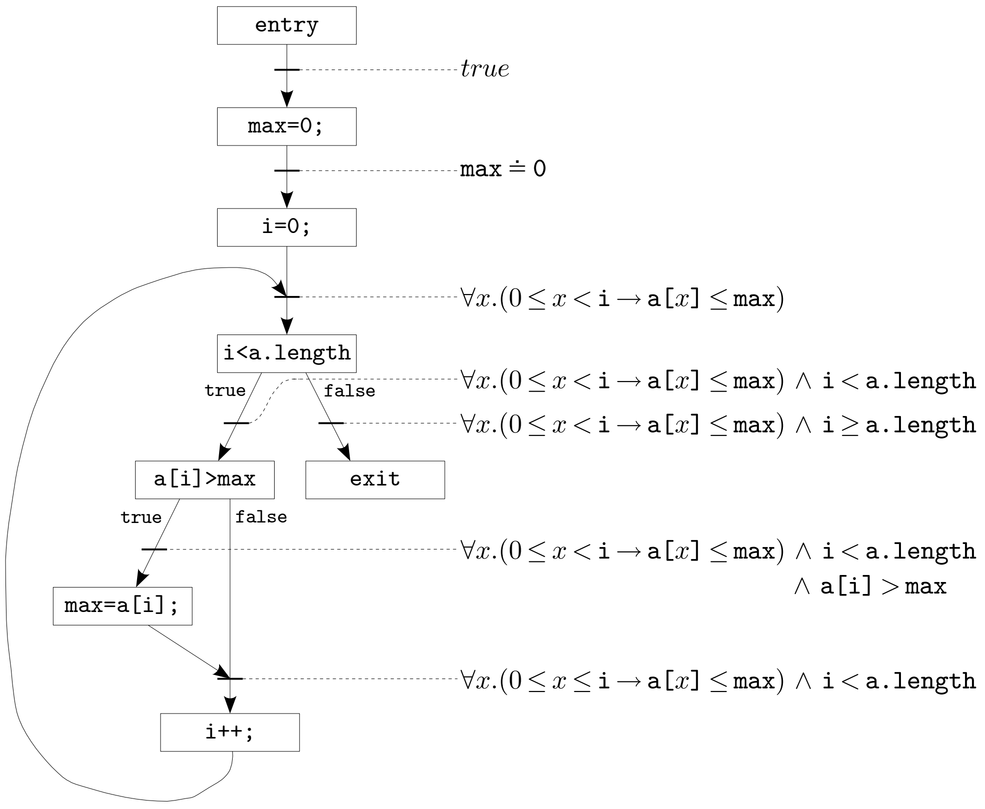
entry
true
max=0;
max ≐ 0
i=0;
∀x.(0 ≤ x < i → a[x] ≤ max)
i<a.length
true
false
∀x.(0 ≤ x < i → a[x] ≤ max) ∧ i < a.length
∀x.(0 ≤ x < i → a[x] ≤ max) ∧ i ≥ a.length
a[i]>max
exit
true
false
∀x.(0 ≤ x < i → a[x] ≤ max) ∧ i < a.length ∧ a[i] > max
max=a[i];
∀x.(0 ≤ x ≤ i → a[x] ≤ max) ∧ i < a.length
i++;

Fig. 1. Control flow graph for the example program, annotated with invariants.

invariant attached to the `exit` node, i.e., the postcondition of the program as a whole, immediately follows from the loop invariant and the negation of the loop guard expression.

4 Our Approach

Our approach is embedded in the overall process of program verification which we imagine has reached a state where a typical goal $\varphi \rightarrow [\mathtt{p}]\psi$ has to be proved. The basic idea is to extend the usual symbolic execution of `p` in the Java Card DL calculus by fixed-point iteration and predicate abstraction, and thereby turn the proving process into a form of abstract interpretation. To help the reader understand how this works out exactly, we outline our approach with the particular instantiations of φ, `p` and ψ given in Fig. 2(1): the goal is to prove that after executing the program introduced in Sect. 3, all elements of the array are less than or equal to `max`. Symbolic execution of the program begins with applying the assignment rule to the first two assignments, which leads to the new proof obligation shown in Fig. 2(2). The effect of the two assignments manifests itself in the two additional assumptions `max` $\doteq 0$ and `i` $\doteq 0$. Now, the active statement of the program is the while loop.

The guiding principle of our construction is to always view the formula on the left hand side of the implication as a candidate for an invariant at the program point reached before the active statement of the program occurring in the modality on the right hand side of the implication. According to this principle, the formula φ_1 in Fig. 2(2) is a first candidate for the loop invariant.

The next step in the symbolic execution has to deal with two cases: the loop is not entered and the loop is unfolded at least once. Technically, we apply the loopUnwind rule followed by the ifElse rule, and obtain the conjunction of the two implications shown in Fig. 2(3).

We refrain from splitting the proof by applying the andRight rule, and for the moment concentrate on the first conjunct. The active statement of its box modality on the right hand side is a conditional statement. Thus, the next symbolic execution step is to apply the ifElse rule, which produces two conjuncts in place of one. Again, we do not split this conjunction with the andRight rule. Instead, the assignment `max = a[i];` in the body of the conditional statement is executed, yielding the proof goal shown in Fig. 2(4). Notice that for the first time the assignment rule necessitates the introduction of a new program variable max' to hold the previous value of `max`.

There is a fundamental difference between the first two conjuncts in Fig. 2(4) and the third: the first two refer to the same point in program execution. More precisely, the right hand sides of the first two implications coincide, so we can apply the merge rule

$$\frac{\varGamma \ \vdash \ (\varphi \vee \varphi') \rightarrow \psi, \ \varDelta}{\varGamma \ \vdash \ (\varphi \rightarrow \psi) \wedge (\varphi' \rightarrow \psi), \ \varDelta}\ \textsf{merge}$$

which replaces the first two implications by one, logically equivalent, implication, Fig. 3(5). This one implication describes the combined effects of the two execution paths, just like in the control flow graph (Fig. 1) there is only one node for the assignment `i++;`, even though it can be reached via several paths. Making such merging steps possible is the reason why we did not and will not apply the andRight rule to split the proof.

After symbolic execution of `i++;`, the last statement of the loop body, we reach in Fig. 3(6) the same program point again that we had already considered in Fig. 2(2), namely the loop entry. Now, we have two invariant candidates φ_1 from (2) and φ_2 from (6) for the same program point. Naturally, we consider their disjunction $\varphi_1 \vee \varphi_2$ as our new invariant candidate, which is shown in Fig. 3(7). Technically this is achieved by applying the loopMerge rule

$$\frac{\varGamma \ \vdash \ (\varphi \vee \varphi') \rightarrow [\pi\ \texttt{while(e) p}\ \omega]\psi, \ \varDelta}{\varGamma \vdash (\varphi \rightarrow [\pi\ \texttt{while(e) p}\ \omega]\psi) \wedge (\varphi' \wedge \neg\texttt{e} \doteq \texttt{true} \rightarrow [\pi\ \omega]\psi), \varDelta}\ \textsf{loopMerge}$$

$$\begin{array}{l}
\mathit{true} \rightarrow [\{\ \texttt{max = 0;} \\
\qquad\quad \texttt{i = 0;} \\
\qquad\quad \texttt{while(i < a.length) \{} \\
\qquad\qquad \texttt{if(a[i] > max) max = a[i];} \\
\qquad\qquad \texttt{i++;} \\
\qquad\quad \texttt{\}} \\
\qquad \texttt{\}}]\,\underbrace{\forall x.(0 \leq x < \texttt{a.length} \rightarrow \texttt{a}[x] \leq \texttt{max})}_{\psi_0}
\end{array} \tag{1}$$

$$\begin{array}{l}
\underbrace{\texttt{max} \doteq 0 \wedge \texttt{i} \doteq 0}_{\varphi_1} \rightarrow [\{\ \texttt{while(i < a.length) \{} \\
\qquad\qquad\qquad\qquad \texttt{if(a[i] > max) max = a[i];} \\
\qquad\qquad\qquad\qquad \texttt{i++;} \\
\qquad\qquad\qquad\quad \texttt{\}} \\
\qquad\qquad\qquad \texttt{\}}]\psi_0
\end{array} \tag{2}$$

$$\begin{array}{l}
\big(\texttt{max} \doteq 0 \wedge \texttt{i} \doteq 0 \wedge \texttt{i} < \texttt{a.length} \rightarrow [\{\ \texttt{if(a[i] > max) max = a[i];} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{i++;} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{while(i < a.length) \{} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{if(a[i] > max) max = a[i];} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{i++;} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \texttt{\}} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad \texttt{\}}]\psi_0\big) \\
\wedge\ \big(\texttt{max} \doteq 0 \wedge \texttt{i} \doteq 0 \wedge \texttt{i} \geq \texttt{a.length} \rightarrow [\{\}]\psi_0\big)
\end{array} \tag{3}$$

$$\begin{array}{l}
\big(\mathit{max}' \doteq 0 \wedge \texttt{i} \doteq 0 \wedge \texttt{i} < \texttt{a.length} \wedge \texttt{a[i]} > \mathit{max}' \wedge \texttt{max} \doteq \texttt{a[i]} \\
\ \rightarrow [\{\ \texttt{i++;} \\
\qquad\quad \texttt{while(i < a.length) \{} \\
\qquad\qquad \texttt{if(a[i] > max) max = a[i];} \\
\qquad\qquad \texttt{i++;} \\
\qquad\quad \texttt{\}} \\
\qquad \texttt{\}}]\psi_0\big) \\
\wedge\ \big(\texttt{max} \doteq 0 \wedge \texttt{i} \doteq 0 \wedge \texttt{i} < \texttt{a.length} \wedge \texttt{a[i]} \leq \texttt{max} \\
\quad \rightarrow [\{\ \texttt{i++;} \\
\qquad\qquad \texttt{while(i < a.length) \{} \\
\qquad\qquad\quad \texttt{if(a[i] > max) max = a[i];} \\
\qquad\qquad\quad \texttt{i++;} \\
\qquad\qquad \texttt{\}} \\
\qquad\quad \texttt{\}}]\psi_0\big) \\
\wedge\ \big(\texttt{max} \doteq 0 \wedge \texttt{i} \doteq 0 \wedge \texttt{i} \geq \texttt{a.length} \rightarrow [\{\}]\psi_0\big)
\end{array} \tag{4}$$

Fig. 2. Invariant inference for the example program (first part).

$$\begin{array}{l}
\big((max' \doteq 0 \land \texttt{i} \doteq 0 \land \texttt{i} < \texttt{a.length} \land \texttt{a[i]} > max' \land \texttt{max} \doteq \texttt{a[i]}\\
\quad \lor\, \texttt{max} \doteq 0 \land \texttt{i} \doteq 0 \land \texttt{i} < \texttt{a.length} \land \texttt{a[i]} \leq \texttt{max})\\
\quad \to [\texttt{\{ i++;}\\
\qquad\quad \texttt{while(i < a.length) \{}\\
\qquad\qquad \texttt{if(a[i] > max) max = a[i];}\\
\qquad\qquad \texttt{i++;}\\
\qquad\quad \texttt{\}}\\
\qquad \texttt{\}}]\psi_0\big)\\
\land\ \big(\texttt{max} \doteq 0 \land \texttt{i} \doteq 0 \land \texttt{i} \geq \texttt{a.length} \to [\texttt{\{\}}]\psi_0\big)
\end{array} \tag{5}$$

$$\begin{array}{l}
\left.\begin{array}{l}
\big((max' \doteq 0 \land i' \doteq 0 \land i' < \texttt{a.length} \land \texttt{a[}i'\texttt{]} > max' \land \texttt{max} \doteq \texttt{a[}i'\texttt{]}\\
\quad \lor\, \texttt{max} \doteq 0 \land i' \doteq 0 \land i' < \texttt{a.length} \land \texttt{a[}i'\texttt{]} \leq \texttt{max})\\
\quad \land\, \texttt{i} \doteq i' + 1
\end{array}\right\}\varphi_2\\
\quad \to [\texttt{\{ while(i < a.length) \{}\\
\qquad\qquad \texttt{if(a[i] > max) max = a[i];}\\
\qquad\qquad \texttt{i++;}\\
\qquad\quad \texttt{\}}\\
\qquad \texttt{\}}]\psi_0\big)\\
\land\ \big(\underbrace{\texttt{max} \doteq 0 \land \texttt{i} \doteq 0}_{\varphi_1} \land \texttt{i} \geq \texttt{a.length} \to [\texttt{\{\}}]\psi_0\big)
\end{array} \tag{6}$$

$$\begin{array}{l}
\varphi_1 \lor \varphi_2 \to [\texttt{\{ while(i < a.length) \{}\\
\qquad\qquad\qquad \texttt{if(a[i] > max) max = a[i];}\\
\qquad\qquad\qquad \texttt{i++;}\\
\qquad\qquad\quad \texttt{\}}\\
\qquad\qquad \texttt{\}}]\psi_0
\end{array} \tag{7}$$

$$\begin{array}{l}
0 \leq \texttt{i} \land \texttt{i} \leq 1 \land \forall x.(0 \leq x < \texttt{i} \to \texttt{a[}x\texttt{]} \leq \texttt{max})\\
\to [\texttt{\{ while(i < a.length) \{}\\
\qquad\quad \texttt{if(a[i] > max) max = a[i];}\\
\qquad\quad \texttt{i++;}\\
\qquad \texttt{\}}\\
\quad \texttt{\}}]\psi_0
\end{array} \tag{8}$$

$$0 \leq \texttt{i} \land \forall x.(0 \leq x < \texttt{i} \to \texttt{a[}x\texttt{]} \leq \texttt{max}) \land \texttt{i} \geq \texttt{a.length} \to [\texttt{\{\}}]\psi_0 \tag{9}$$

Fig. 3. Invariant inference for the example program (second part).

where e must not have side effects. The same principle guides both the merge and the loopMerge rule: the effects of several execution paths are combined in one implication.

If φ_1 was logically equivalent to $\varphi_1 \vee \varphi_2$, we could stop here and declare φ_1 to be our prime candidate for the loop invariant: it would be a fixed point of our iterative inference process. But as you can easily see, this is not the case in our example. So, we have to go on, unfold the loop body once more, obtain a loop invariant candidate φ_3 after the second iteration, check whether $\varphi_1 \vee \varphi_2$ is logically equivalent to $\varphi_1 \vee \varphi_2 \vee \varphi_3$, and then stop or go on accordingly. The problem with this plan of action is that it might (and, in the example, would) not terminate. This is where predicate abstraction as it is, e.g., described in [12], comes into play. To apply this method we first need to fix a set P of predicates. For the example, we choose

$$P = \{\underbrace{\mathtt{i} \doteq 0}_{p_1},\ \underbrace{0 \leq \mathtt{i}}_{p_2},\ \underbrace{\mathtt{i} \leq 1}_{p_3},\ \underbrace{\forall x.(0 \leq x < \mathtt{i} \rightarrow \mathtt{a[}x\mathtt{]} \leq \mathtt{max})}_{p_4}\} \ .$$

In general P might be chosen by following heuristics, e.g., include all parts of the invariant candidate accumulated before the first unfolding of the loop, the loop guard, and parts of the postcondition ψ. In addition one might include in P all the *usual suspect* invariants, as is, e.g., done in [10]. As a final resort P could be customised by user interaction and trial and error. Once P is agreed upon we continue in the above example by replacing $\varphi_1 \vee \varphi_2$ with its abstraction, which is the conjunction of all predicates $p \in P$ for which $(\varphi_1 \vee \varphi_2) \rightarrow p$ is a tautology. Formally, we apply the abstraction rule

$$\frac{\Gamma \ \vdash \ \varphi' \rightarrow \psi, \ \Delta}{\Gamma \ \vdash \ \varphi \rightarrow \psi, \ \Delta} \ \textsf{abstraction}$$

where $\varphi' = \bigwedge\{p \in P \ \mid \ \varphi \rightarrow p \ \text{ is valid}\}$. In our example we get $\varphi' = p_2 \wedge p_3 \wedge p_4$, see Fig. 3(8).

After symbolically executing the loop body for the second time and again applying loopMerge and abstraction, we arrive at $p_2 \wedge p_4$. Since P is of finite size, this process of eliminating predicates must eventually terminate. In the example, it does so after just one more iteration: if we symbolically execute the loop body a third time, the new invariant candidate reached after applying loopMerge and abstraction is again $p_2 \wedge p_4$. We have reached a *fixed point* and stop iterating the while loop. Technically speaking, instead of applying the loopUnwind rule we apply

$$\frac{\Gamma \ \vdash \ \varphi \wedge \neg \mathtt{e} \doteq \mathtt{true} \rightarrow [\pi\, \omega]\psi, \ \Delta}{\Gamma \ \vdash \ \varphi \rightarrow [\pi \ \mathtt{while(e)p}\ \omega]\psi, \ \Delta} \ \textsf{loopEnd}$$

where e must not have side effects. In our running example this rule application results in Fig. 3(9).

There is one problem with the loopEnd rule: Unlike the other rules introduced in this section, it is not sound, as you can easily see. Applying it is sound if the formula φ is an invariant for the loop. In situations like the one in the example, we have good reason to believe that this is the case: φ is a fixed point of accumulated symbolic executions of the loop body, so it should hold at the loop entry in all possible concrete executions. This is however not quite guaranteed; for example, non-symbolic-execution rules might have been applied in between, disrupting the inference process. Formally prohibiting the application of the loopEnd rule in such situations is conceivable, but complicated. In our context, this is not a necessity: Our implementation prevents unsound applications in a heuristic manner, and if in very rare cases these heuristics should fail and an inferred "invariant" not really be an invariant, this error would be caught when trying to use the false invariant for a regular proof with the invariant rule.

What is a good loop invariant? After all the logical constant *true* is always an invariant. In our scenario where invariant inference is just one part of an overall program verification effort the answer is easy: a loop invariant is good if it allows to successfully complete the overall proof goal. This is the case in our example, since Fig. 3(9) can easily be seen to be universally valid.

In summary, our approach is as follows. The analysed program is symbolically executed with the program rules of the JAVA CARD DL calculus, but without intertwining this process with applications of other rules such as andRight. The symbolic execution is coordinated such that it follows the structure of the control flow graph; in particular, at confluences in the graph, the conjuncts describing the predecessors are combined using the merge and loopMerge rules. Each application of loopMerge is followed by an application of the abstraction rule. After applying abstraction, it is checked whether the resulting abstracted loop invariant candidate is a fixed point, i.e., whether the previous such candidate consisted of the same predicates. If so, it is taken as the inferred loop invariant, and loopEnd is applied. Otherwise, the loop is symbolically executed once again. This process works completely automatically, except that the user may choose to help it find better invariants by specifying predicates for each loop once in the beginning.

5 Implementation

We have implemented our method as an extension of the KeY system. The core element of this implementation are the rules introduced in Sect. 4. The bottleneck of the approach clearly lies in the abstraction rule: it requires checking for each predicate $p \in P$ whether p is implied by the invariant candidate. These checks are implemented as calls to an external automated first-order theorem prover such as Simplify [8]. Decision procedures which support input in the SMT-LIB format [20] can also be used. Of course, such theorem prover calls are neither always successful nor computationally cheap. The lack of completeness is mitigated by the fact that the predicates tend to be simple and thus manageable by the theorem prover. Acceptable performance can only be achieved by employing optimisations which reduce the number of necessary calls. Our implementation features several such optimisations. For example, it exploits implication relationships between predicates: if $p_1 \rightarrow p_2$ is known to be valid, and the theorem prover has not been able to prove $\varphi \rightarrow p_2$, then there is no point in checking the validity of $\varphi \rightarrow p_1$. Possibly, performance could be improved further by using an existing predicate abstraction algorithm such as the one from [11] at this place.

Besides the rules themselves, the implementation also comprises a heuristic predicate generator, which automatically creates many of the "usual suspect" invariant elements, such as ordering comparisons between integer program variables, or equality and inequality between variables of a reference type. The predicate generator is complemented by the possibility to manually enter predicates. No quantified formulas are generated automatically, as the number of predicates would get too large to be manageable. However, manually entered predicates are allowed to contain free variables; such predicates are then universally closed by the predicate generator, using various guards. For example, if the user specifies the predicate $\texttt{a[}x\texttt{]} \leq \texttt{max}$, the predicate generator adds $\forall x.(0 \leq x < i \rightarrow \texttt{a[}x\texttt{]} \leq \texttt{max})$, together with many other similar predicates.

The final element of the implementation is a proof search strategy, which controls the automatic application of the rules as it is necessary for invariant inference. In particular, the strategy prohibits the application of non-symbolic-execution rules, and it coordinates symbolic execution of several modalities such that it follows the control flow graph: if, e.g., the current proof goal is $(\varphi_1 \rightarrow [\texttt{max = a[i]; i++;}]\psi) \wedge (\varphi_2 \rightarrow [\texttt{i++;}]\psi)$, symbolic execution of the second conjunct is stopped until `max = a[i];` has been symbolically executed in the first conjunct and the merge rule can be applied.

6 Experiments

We tested our implementation on selection sort, a well-known and comparatively simple sorting algorithm with quadratic time complexity. The basic idea of the algorithm is to find the smallest element of the array to be sorted, swap it with the first element, and iteratively repeat this process on the subarray starting at the second position. The exact proof obligation supplied to the KeY system is shown in Fig. 4. It states that, after invoking the program contained in the box modality on an integer array `a` which is not `null` and which has a positive length, the array is sorted. This specification is not strong enough to ensure that the program actually sorts the array; for example, a program could satisfy it by simply setting all array elements to zero. It is however sufficient for our purposes. The program itself is a straightforward Java Card rendering of selection sort. The temporary boolean variables `condOuter` and `condInner` are used to buffer the values of the loop guard expressions, which is necessary because our implementation of the invariant inference currently requires that the guard expressions must be simple program variables.

```
a ≠ null ∧ a.length > 0 → [{ i = 0;
                             condOuter = i < a.length;
                             while(condOuter) {
                               minIndex = i;
                               j = i + 1;
                               condInner = j < a.length;
                               while(condInner) {
                                 if(a[j] < a[minIndex]) minIndex = j;
                                 j++;
                                 condInnder = j < a.length;
                               }
                               temp = a[i];
                               a[i] = a[minIndex];
                               a[minIndex] = temp;
                               i++;
                               condOuter = i < a.length;
                             }
                           }]∀x.(0 < x ∧ x < a.length → a[x − 1] ≤ a[x])
```

Fig. 4. Java Card DL proof obligation for selection sort.

We manually entered the predicates ($\texttt{condOuter} \doteq \texttt{true} \leftrightarrow \texttt{i} < \texttt{a.length}$), ($\texttt{condInner} \doteq \texttt{true} \leftrightarrow \texttt{j} < \texttt{a.length}$), ($\texttt{a[}x\texttt{]} \leq \texttt{a[}y\texttt{]}$), and ($\texttt{a[minIndex]} \leq \texttt{a[}x\texttt{]}$), where x and y are free variables. The first two of these are necessary only because of the introduction of condInner and condOuter, and their generation could easily be automated. The other two require more ingenuity, but are still significantly easier to guess than the loop invariants themselves. Together with the automatically generated predicates, this lead to 8794 predicates for the inner loop and 16950 predicates for the outer loop.

Using Simplify as the external first-order theorem prover, the invariant inference process terminated after 3 iterations for the outer loop, containing 4, 4 and 2 iterations for the inner loop, respectively. 652 rules were applied in total. The overall running time on a 1.5 GHz Pentium M machine was about 8.5 minutes. Approximately 65% of this time was spent running Simplify, which was called exactly 800 times. The resulting loop invariants for the inner and the outer loop are shown in Fig. 5 and Fig. 6.

In addition to loop invariants, the invariant rule of the Java Card DL calculus [4] makes use of modifier sets for loops, i.e., information about which memory locations may be modified by a loop. In the case of selection sort, appropriate modifier sets are $\{\texttt{minIndex}, \texttt{j}, \texttt{condInner}\}$ for the inner loop, and $\{\texttt{minIndex}, \texttt{j}, \texttt{condInner}, \texttt{temp}, \texttt{a[*]}, \texttt{i}, \texttt{condOuter}\}$ for the outer loop. When supplied with these modifier sets (which are quite obvious from the program

$$
\begin{aligned}
&\forall x.\forall y.(0 \leq x \land x < y \land y \leq i \rightarrow \mathtt{a}[x] \leq \mathtt{a}[y])\\
&\land\ \forall x.(\mathtt{i} \leq x \land x < \mathtt{minIndex} \rightarrow \mathtt{a[minIndex]} \leq \mathtt{a}[x])\\
&\land\ \forall x.(\mathtt{i} < x \land x < \mathtt{j} \rightarrow \mathtt{a[minIndex]} \leq \mathtt{a}[x])\\
&\land\ \forall x.(\mathtt{minIndex} < x \land x < \mathtt{j} \rightarrow \mathtt{a[minIndex]} \leq \mathtt{a}[x])\\
&\land\ \mathtt{a[0]} \leq \mathtt{a[i]}\\
&\land\ \mathtt{a[minIndex]} \leq \mathtt{a[i]}\\
&\land\ 0 < \mathtt{a.length}\\
&\land\ \mathtt{j} \leq \mathtt{a.length}\\
&\land\ 0 < \mathtt{j}\\
&\land\ \mathtt{minIndex} < \mathtt{a.length}\\
&\land\ 0 \leq \mathtt{minIndex}\\
&\land\ \mathtt{minIndex} < \mathtt{j}\\
&\land\ \mathtt{i} < \mathtt{a.length}\\
&\land\ 0 \leq \mathtt{i}\\
&\land\ \mathtt{i} < \mathtt{j}\\
&\land\ \mathtt{i} \leq \mathtt{minIndex}\\
&\land\ \forall x.\forall y.(0 \leq x \land x < \mathtt{i} \land \mathtt{i} \leq y \land y < \mathtt{a.length} \rightarrow \mathtt{a}[x] \leq \mathtt{a}[y])\\
&\land\ \forall x.\forall y.(0 \leq x \land x < \mathtt{i} \land \mathtt{i} \leq y \land y < \mathtt{j} \rightarrow \mathtt{a}[x] \leq \mathtt{a}[y])\\
&\land\ \forall x.\forall y.(0 \leq x \land x < \mathtt{i} \land \mathtt{i} \leq y \land y < \mathtt{minIndex} \rightarrow \mathtt{a}[x] \leq \mathtt{a}[y])\\
&\land\ \mathtt{a} \not\doteq \mathtt{null}\\
&\land\ \mathtt{condOuter} \doteq \mathtt{true}\\
&\land\ (\mathtt{condOuter} \doteq \mathtt{true} \leftrightarrow \mathtt{i} < \mathtt{a.length})\\
&\land\ (\mathtt{condInner} \doteq \mathtt{true} \leftrightarrow \mathtt{j} < \mathtt{a.length})
\end{aligned}
$$

Fig. 5. Inferred invariant for the inner loop of selection sort.

$$
\begin{aligned}
&\forall x.\forall y.(0 \leq x \land x < y \land y < \mathtt{i} \rightarrow \mathtt{a}[x] \leq \mathtt{a}[y])\\
&\land\ 0 < \mathtt{a.length}\\
&\land\ \mathtt{i} \leq \mathtt{a.length}\\
&\land\ 0 \leq \mathtt{i}\\
&\land\ \forall x.\forall y.(0 \leq x \land x < \mathtt{i} \land \mathtt{i} \leq y \land y < \mathtt{a.length} \rightarrow \mathtt{a}[x] \leq \mathtt{a}[y])\\
&\land\ (\mathtt{condOuter} \doteq \mathtt{true} \leftrightarrow \mathtt{i} < \mathtt{a.length})\\
&\land\ \mathtt{a} \not\doteq \mathtt{null}
\end{aligned}
$$

Fig. 6. Inferred invariant for the outer loop of selection sort.

code), and, crucially, the loop invariants from Fig. 5 and Fig. 6, the KeY system (in normal mode) was able to automatically prove the validity of the formula from Fig. 4 in about 2 minutes.

7 Conclusions

We have presented a method for inferring invariants for while loops in JAVA programs that can seamlessly be integrated in program verification based on the symbolic execution paradigm. To do so this paradigm had to be adapted in two aspects. First, when using symbolic execution for program verification, intermediate proof goals that involve a case distinction are split into subgoals that are then proved separately. For invariant inference we have to avoid this splitting. Second, the ideas of fixed-point iteration and approximation are not present in the symbolic execution paradigm for program verification. So, we had to introduce them.

The approach has been implemented as an addition to the KeY verification system. The results of first experiments are very encouraging. But, since the success to a great deal depends on the heuristic choice of the set P of abstraction predicates, much more experience is needed to arrive at a dependable evaluation.

Approximation in static analysis typically takes the form of "erring on the safe side", i.e., precision may be lost, but not correctness. In principle, our invariant inference is no exception: the inferred invariants may sometimes not be useful, but they should always indeed be invariants. However, since we introduced a rule which is not strictly sound, this is not guaranteed with the same high degree of confidence that is carried by the axioms from the KeY calculus. Remedying this imperfectness is one direction for future work. Nevertheless, the success rate of suggesting true invariants is already a lot higher than in dynamic analysis methods such as Daikon.

Another line of future work, which is more speculative, concerns the generation of the abstraction predicates. One could investigate the use of CEGAR (counterexample-guided abstraction refinement) techniques [6, 5] to arrive at useful predicates in a less heuristic, more systematic manner.

References

1. B. Beckert, M. Giese, R. Hähnle, V. Klebanov, P. Rümmer, S. Schlager, and P. H. Schmitt. The KeY System 1.0 (deduction component). In F. Pfenning, editor, *Proceedings, 21st International Conference on Automated Deduction (CADE)*, LNCS. Springer, 2007. To appear.
2. B. Beckert and C. Gladisch. White-box testing by combining deduction-based specification extraction and black-box testing. In Y. Gurevich, editor, *Proceedings, International Conference on Tests and Proofs (TAP), Zürich, Switzerland*, LNCS. Springer, 2007. To appear.
3. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *LNCS*. Springer, 2007.

4. B. Beckert, S. Schlager, and P. H. Schmitt. An improved rule for while loops in deductive program verification. In K.-K. Lau, editor, *Proceedings, 7th International Conference on Formal Engineering Methods (ICFEM)*, volume 3785 of *LNCS*, pages 315–329. Springer, 2005.
5. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. Checking memory safety with Blast. In M. Cerioli, editor, *Proceedings, 8th International Conference on Fundamental Approaches to Software Engineering (FASE)*, volume 3442 of *LNCS*, pages 2–18. Springer, 2005.
6. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proceedings, 12th International Conference on Computer Aided Verification (CAV)*, pages 154–169. Springer, 2000.
7. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings, 4th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 238–252. ACM Press, 1977.
8. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories Palo Alto, 2003.
9. C. Engel and R. Hähnle. Generating unit tests from formal proofs. In Y. Gurevich, editor, *Proceedings, International Conference on Tests and Proofs (TAP), Zürich, Switzerland*, LNCS. Springer, 2007. To appear.
10. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.
11. C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *Proceedings, 29th Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 191–202. ACM Press, 2002.
12. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proceedings, 9th International Conference on Computer Aided Verification (CAV)*, pages 72–83. Springer, 1997.
13. D. Harel, D. Kozen, and J. Tiuryn. *Dynamic Logic.* MIT Press, 2000.
14. J. J. Hunt, E. Jenn, S. Leriche, P. Schmitt, I. Tonin, and C. Wonnemann. A case study of specification and verification using JML in an avionics application. In M. Rochard-Foy and A. Wellings, editors, *Proceedings, 4th Workshop on Java Technologies for Real-time and Embedded Systems (JTRES).* ACM Press, 2006.
15. Java Card platform specification 2.2.1. Sun Microsystems, Inc., October 2003.
16. J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
17. K. R. M. Leino and F. Logozzo. Loop invariants on demand. In K. Yi, editor, *Proceedings, 3rd Asian Symposium on Programming Languages and Systems (APLAS)*, volume 3780 of *LNCS*, pages 119–134. Springer, 2005.
18. B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, 1992.
19. W. Mostowski. Fully verified Java Card API reference implementation. In *Proceedings, 4th International Verification Workshop (VERIFY'07), Workshop at CADE-21, Bremen, Germany.* CEUR Workshop Proceedings, 2007. To appear.
20. S. Ranise and C. Tinelli. The SMT-LIB standard: Version 1.2. Technical report, University of Iowa, 2006.
21. P. Rümmer. Sequential, parallel, and quantified updates of first-order structures. In *Proceedings, 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 4246 of *LNCS*, pages 422–436. Springer, 2006.
22. P. H. Schmitt and I. Tonin. Verifying the Mondex case study. In M. Hinchey and T. Margaria, editors, *Proceedings, 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM).* IEEE Press, 2007. To appear.
23. S. Stepney, D. Cooper, and J. Woodcock. An electronic purse: Specification, refinement, and proof. Technical monograph PRG-126, Oxford University Computing Laboratory, July 2000.
24. B. Weiß. Inferring invariants by static analysis in KeY. Diplomarbeit, University of Karlsruhe, March 2007.