Predicate Abstraction in a Program Logic Calculus

Benjamin Weiß

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

Abstract. Predicate abstraction is a form of abstract interpretation where the abstract domain is constructed from a finite set of predicates over the variables of the program. This paper explores a way to integrate predicate abstraction into a calculus for deductive program verification, where it allows to infer loop invariants automatically that would otherwise have to be given interactively. The approach has been implemented as a part of the KeY verification system.

1 Introduction

Deductive verification of imperative programs typically requires hand-crafted *loop invariants*, i.e., assertions about the program states which can possibly occur at the beginning of each iteration of a loop. Finding sufficiently strong loop invariants can be difficult, and today this is often one of only a few human interactions necessary in an otherwise heavily automated verification environment.

On the other hand, there are methods which can automatically determine loop invariants. Leaving aside testing-based approaches like Daikon [9], such methods are predominantly based on *abstract interpretation* [6], a theoretical framework for static program analysis which can roughly be described as symbolic execution of the program, using an abstract (i.e., approximative) domain for the variable values, together with fixed-point iteration.

Predicate abstraction [11] is a variant of abstract interpretation where the abstract domain is constructed from a finite set of predicates over the variables of the program. Here, the symbolic execution is itself done in a precise fashion, and the necessary approximation is performed in between by explicit abstraction steps, in which an automated theorem prover is used to determine a valid boolean combination of the predicates. Compared with other forms of abstract interpretation, a fundamental disadvantage of predicate abstraction is that it is limited to *finite* abstract domains. On the other hand, an advantage is that its abstract domain can be flexibly adapted by simply changing the set of predicates. In the same vein, predicate abstraction can quite easily support complex, quantified invariants [10]. It can be extended with an iterative refinement process that automatically adapts the domain to the particular problem [5].

This paper presents an approach for integrating predicate abstraction into a deductive program verification calculus. This allows to infer loop invariants within this calculus, on demand and as an integral part of constructing the overall correctness proof.

Outline. Sect. 2 gives an overview of relevant related work. Necessary background on the underlying program logic and calculus is provided in Sect. 3. A high level explanation of the approach follows in Sect. 4. In Sect. 5, new calculus rules are introduced, and how these rules are to be used is described in Sect. 6. Sect. 7 gives details on the predicate abstraction scheme. The overall method is further illustrated with the help of an example in Sect. 8, and practical experience with an implementation is reported in Sect. 9. Finally, Sect. 10 contains conclusions and future work.

2 Related Work

This paper draws much inspiration from Flanagan and Qadeer's approach for using predicate abstraction in program verification [10]. Both in their approach and in ours, a set of predicates is associated with each loop in a program, and used to abstract specifically at loop entry points. Quantified loop invariants are supported by allowing the loop predicates to contain free variables which are later quantified over. The main difference is that in our setting, the inference is done within a logical calculus, the same that is used for the verification itself. This also distinguishes our technique from the one used in the Boogie verifier [2], where a separate abstract interpretation component is used to infer needed loop invariants, leading to a duplication of knowledge between the verifier and the abstract interpreter.

Several related approaches in striving for a closer integration between deductive verification and static analysis based invariant inference exist. In the "loop invariants on demand" technique [14], first-order verification conditions are generated from programs, which include placeholder predicates for the loop invariants. These are then passed to a first-order theorem prover. When an invariant is necessary for a sub-proof, the prover tries to infer it by repeatedly invoking an abstract interpreter with successively more precise abstract domains. Still, the verification condition generator, theorem prover, and abstract interpreter, are all separate components. In [15], parts of the invariant generation are moved inside the theorem prover, with the verification condition generation remaining separated. In our approach, all three tasks—especially generation of verification conditions and generation of invariants, which are closely related as they both deal with programs—can be performed within one program logic theorem prover. Logical interpretation [19] goes the other way round by embedding theorem proving techniques in an abstract interpretation framework.

The results presented in this paper are based on earlier work reported in [18]. Compared to [18], there are significant improvements: no unsoundness issues remain; the integration of invariant inference into the calculus is more natural, as proofs are no longer necessarily tree-shaped; and the transformation of state updates into formulas is now lazy instead of eager, which improves performance.

3 Program Logic

The verification framework used in this paper is *dynamic logic with non-rigid functions (DL)* [4,3], a generalisation of Hoare logic [12]. DL extends first-order logic by modal operators $[\mathtt{p}]$, where $\mathtt{p}$ can be any legal sequence of statements in some programming language. Additionally, it features modal operators $\{u\}$, where u is a so-called *update* [17], representing a state change in a language-independent, logical way. A core DL for a minimalist object-oriented language is formally defined in [4], and a full-blown version for Java in [3].

Formulas are evaluated in program states, which are first-order structures. The formula $[\mathtt{p}]\psi$ holds in a state if all states reachable by executing $\mathtt{p}$ in this state satisfy ψ. Similarly, $\{u\}\psi$ holds in a state if ψ holds in the state produced by the update u. A formula is *valid* if it holds in all states. A typical program verification task is to prove the validity of a formula $\varphi \rightarrow [\mathtt{p}]\psi$, which is equivalent to the Hoare triple $\{\varphi\}\mathtt{p}\{\psi\}$. Object attributes are represented as non-rigid function symbols, i.e., symbols whose interpretation may be changed by programs.

The validity of DL formulas can be proven using a sequent calculus. A *sequent* is a construct $\Gamma \vdash \Delta$, where Γ and Δ are finite sets of formulas, and whose semantics is the same as that of $\bigwedge \Gamma \rightarrow \bigvee \Delta$. A sequent calculus *rule* deduces the validity of a sequent (the rule's *conclusion*) from the validity of one or more other sequents (the rule's *premises*). In order to prove the validity of a sequent, one constructs a *proof tree*: its root is the original sequent itself, and in each step, it is extended by applying a rule to one of its leaves (called *goals*). Applying a rule means matching its conclusion to the goal, and adding its premises as children of the goal. If a proof goal is obviously valid (e.g., $\Gamma \vdash$ *true*), it is *closed.* If all goals of a proof tree are closed, this means that the root sequent is valid as well.

Formulas with programs in them may be handled by rules which operate on the *active statement*, i.e., the first basic command in the modal operator, and stepwise shorten the program until only a first-order problem remains. Intuitively, this process can be understood as *symbolic execution*: the program is "executed", but with symbolic instead of concrete values for its variables. It is similar to the *verification condition generation* in related verification approaches, but differs in that it is intertwined with other forms of reasoning, in particular first-order reasoning and arithmetic simplification, within the same calculus.

Such symbolic execution rules formalise the semantics of the underlying programming language. In the following, we take a look at rules for the three elementary programming constructs of assignments, conditional statements, and loops, in a Java-like language. The basic assignment rule is

$$\textsf{assign}\ \frac{\Gamma \ \vdash\ \{u; \mathtt{x} := \mathtt{se}\}[\omega]\psi,\ \Delta}{\Gamma \ \vdash\ \{u\}[\mathtt{x\ =\ se;}\ \omega]\psi,\ \Delta}$$

where Γ and Δ are sets of formulas; u is an update; $\mathtt{se}$ is a "simple expression", i.e., an expression without side effects; ω is the rest of the program after the assignment; and ψ is a formula. The rule simply transforms the *program assignment* `x = se;` into an equivalent *update* $\mathtt{x} := \mathtt{se}$. The update $u; \mathtt{x} := \mathtt{se}$ is the sequential composition of the updates u and $\mathtt{x} := \mathtt{se}$. Parallel composition of updates is also possible; for example, $(\mathtt{x} := 1 \| \mathtt{y} := \mathtt{x})$ sets `x` to 1 and `y` to the value of `x` simultaneously. Finally, the update language allows quantified updates such as $(\mathit{for}\ x; \mathtt{a}[x] := 0)$, which sets all elements of the array `a` to 0 in parallel.

The assign rule reduces assignments to updates. In the course of symbolic execution, a composite update accumulates in this way in front of the modal operator. This update can be simplified aggressively using update rewriting rules [17], which for simplicity we use as a monolithic rule simplifyUpdate here. Once the program has been dealt with completely, the final update can be applied to the postcondition as a substitution (also by simplifyUpdate). As an example, consider the following unclosed proof tree (with the root at the bottom):

$$
\begin{array}{rc}
 & \vdash\ \mathit{if}(\mathtt{a} \doteq \mathtt{b})\mathit{then}(2)\mathit{else}(1) \doteq 1 \\ \cline{2-2}
(\mathsf{simplifyUpdate}) & \vdash\ \{\mathtt{a.f} := 1; \mathtt{b.f} := 2\}(\mathtt{a.f} \doteq 1) \\ \cline{2-2}
(\mathsf{assign}) & \vdash\ \{\mathtt{a.f} := 1;\}[\mathtt{b.f\ =\ 2;}]\mathtt{a.f} \doteq 1 \\ \cline{2-2}
(\mathsf{simplifyUpdate}) & \vdash\ \{\mathtt{a.f} := 0; \mathtt{a.f} := 1;\}[\mathtt{b.f\ =\ 2;}]\mathtt{a.f} \doteq 1 \\ \cline{2-2}
(\mathsf{assign}) & \vdash\ \{\mathtt{a.f} := 0\}[\mathtt{a.f\ =\ 1;\ b.f\ =\ 2;}]\mathtt{a.f} \doteq 1 \\ \cline{2-2}
(\mathsf{assign}) & \vdash\ [\mathtt{a.f\ =\ 0;\ a.f\ =\ 1;\ b.f\ =\ 2;}]\mathtt{a.f} \doteq 1
\end{array}
$$

Terms like `f(a)`, where `f` is a non-rigid function symbol, are written as `a.f` in order to resemble the usual object attribute access notation. One after the other, the three assignments are turned into updates. Since the first is overridden by the second, it can be simplified away. Finally, the update is applied to the postcondition $\mathtt{a.f} \doteq 1$ (expressing equality of `a.f` and 1). This last step creates a syntactical case distinction on whether `a` and `b` refer to the same object. Delaying and sometimes avoiding such *aliasing* related case distinctions is the primary motivation for handling assignments via updates in this way.

Conditional statements are symbolically executed by branching the proof on whether the guard is true or false, and loops by unwinding them:

$$
\mathsf{ifElse}\ \frac{\begin{array}{ll}\Gamma,\ \{u\}\mathtt{se} \doteq \mathtt{true}\ \vdash\ \{u\}[\mathtt{p}\ \omega]\psi,\ \Delta & \text{(then branch)} \\ \Gamma,\ \{u\}\mathtt{se} \doteq \mathtt{false}\ \vdash\ \{u\}[\mathtt{q}\ \omega]\psi,\ \Delta & \text{(else branch)}\end{array}}{\Gamma\ \vdash\ \{u\}[\mathtt{if(se)\ p\ else\ q}\ \omega]\psi,\ \Delta}
$$

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

Using loopUnwind is sufficient only for loops which terminate after a fixed, statically known number of iterations. General loops can be handled with loopInvariant (both loop rules are shown in a simplified form which assumes that the loop body does not terminate abruptly, e.g by throwing an exception):

$$
\mathsf{loopInvariant}\ \frac{\begin{array}{ll}\Gamma\ \vdash\ \{u\}\mathit{Inv},\ \Delta & \text{(initially valid)} \\ \mathit{Inv},\ \mathtt{se} \doteq \mathtt{true}\ \vdash\ [\mathtt{p}]\mathit{Inv} & \text{(preserved by body)} \\ \mathit{Inv},\ \mathtt{se} \doteq \mathtt{false}\ \vdash\ [\omega]\psi & \text{(use case)}\end{array}}{\Gamma\ \vdash\ \{u\}[\mathtt{while(se)\ p}\ \omega]\psi,\ \Delta}
$$

Here, *Inv* is a loop invariant which has to be provided from the outside. The first two branches ensure that *Inv* is indeed an invariant, i.e., that it holds both when initially encountering the loop and after an arbitrary number of loop iterations. In the third branch, symbolic execution continues behind the loop.

4 Approach

A program logic calculus like the one introduced in the previous section bears many similarities to abstract interpretation style program analysis; both use symbolic execution to infer and check properties about programs. Unlike usual abstract interpretations, the deductive approach can, at least in principle, handle arbitrarily precise properties. This comes at the cost of sometimes needing human interaction for proving the resulting first-order problems, and at the cost of requiring manually specified loop invariants. This paper aims to address the latter issue by integrating abstract interpretation concepts into the deductive setting.

A difference between abstract interpretation and our calculus is in the treatment of control flow splits: the calculus handles them by branching the proof tree, where the created branches remain separated permanently. On the other hand, abstract interpretations typically use a "merge" operator to combine properties at junction points in the control flow graph. This corresponds to accumulating properties for every program point, instead of treating the execution paths separately. For loops, the infinite number of paths makes such an accumulation necessary; deductive verification "cheats" here by assuming to be given a loop invariant, which already is an accumulated description of all paths through the loop. We can overcome this difference rather straightforwardly by introducing a rule into the calculus which merges several proof branches into one.

With this change, loops can be treated by applying loopUnwind and ifElse, symbolically executing the body, and then merging the resulting sequent (where the loop entry is again the active statement) with the previous such sequent. For example, we might begin with a sequent $\mathtt{i} \doteq 0 \vdash [\mathtt{while(i<j)\ \ldots}]\psi$. After one iteration, we might arrive at $\mathtt{i} \doteq 0 \lor \mathtt{i} \doteq 1 \vdash [\mathtt{while(i<j)\ \ldots}]\psi$, reflecting the fact that after this iteration, `i` has been incremented by one. Every such iteration leads to a larger set of states possible for the loop entry point. In principle, we only have to repeat this iterative process until this set of states stabilises, i.e., until it is a fixed point of the process: once this happens, it covers all states which are possible for the loop entry on any execution path, or in other words, its representation as a formula then is a loop invariant.

In the terminology of abstract interpretation, this would correspond to a computation of the *static semantics*. Obviously, the infinite number of states means that for most loops, such a computation would not terminate. To change this, we need to introduce *approximation*. A form of approximation particularly suitable in our context is that of predicate abstraction [11,10]: We assume that for each loop we are given a finite set P of predicates (formulas). Then, the abstraction of a formula for the entry point of this loop is a boolean combination of elements of P which is implied by the original formula. That is, the abstraction retains the information from the formula which is expressible by the predicates in P, and approximates away everything else. Since there are only finitely many boolean combinations of the predicates, performing such an abstraction before each unwinding step ensures convergence after a finite number of iterations. The found invariant can then be used to apply loopInvariant.

With predicate abstraction, the predicates P associated with a loop form the building blocks for the invariants which can be found for that loop. Such predicates can either be specified manually—which is easier than having to specify whole, correct loop invariants—or be generated heuristically based on the particular program and specification to be verified.

5 Rules

In this section, we define new sequent calculus rules which extend a rule base like the one sketched in Sect. 3 with predicate abstraction based loop invariant inference as described in Sect. 4. The soundness proofs for these rules are omitted here for space reasons. First is a rule for merging execution paths at junction points in the control flow graph, called merge:

$$\mathsf{merge}\ \frac{\bigwedge(\Gamma_1 \cup \neg\Delta_1) \vee \cdots \vee \bigwedge(\Gamma_n \cup \neg\Delta_n)\ \vdash\ \psi}{\Gamma_1\ \vdash\ \psi, \Delta_1 \qquad \ldots \qquad \Gamma_n\ \vdash\ \psi, \Delta_n}$$

This rule is unusual in that it has several conclusions, or in other words, in that it is applied to several proof goals at once. To allow such rules means to generalise the structure of proofs from trees to directed acyclic graphs (DAGs) which are connected and rooted. Apart from that, merge is a rather simple rule operating on the propositional logic level. A typical application (to be read, intuitively, from bottom to top) is

$$(\mathsf{merge})\frac{\varphi_1 \vee \varphi_2 \vdash [\mathtt{while(e)\ p}]\psi}{\varphi_1 \vdash [\mathtt{while(e)\ p}]\psi \qquad \varphi_2 \vdash [\mathtt{while(e)\ p}]\psi}$$

The next rule is responsible for the predicate abstraction step:

$$\mathsf{predicateAbstraction}\ \frac{\alpha_P(\bigwedge(\Gamma \cup \neg\Delta))\ \vdash\ [\mathtt{while(e)\ p}\ \omega]\psi}{\Gamma\ \vdash\ [\mathtt{while(e)\ p}\ \omega]\psi, \Delta}$$

where P is the set of predicates associated with the loop `while(e)p`, and where α_P is a meta-operator which computes for any formula φ a predicate abstraction using P. This means that $\alpha_P(\varphi)$ is some boolean combination of the predicates in P such that $\varphi \rightarrow \alpha_P(\varphi)$ is valid. The details of computing $\alpha_P(\varphi)$ depend on the particular predicate abstraction scheme (Sect. 7); usually, this computation itself requires first-order reasoning modulo several theories.

Both above rules operate on sequents without updates in front of the modal operators containing the programs. Thus, we need a way to transform typical sequents $\varphi \vdash \{u\}[\mathtt{p}]\psi$ such that the update u is removed from a modality $[\mathtt{p}]$. This can be achieved with the shiftUpdate rule:

$$\mathsf{shiftUpdate}\ \frac{\{u'\}\Gamma,\ \mathit{Upd}\ \vdash\ [\mathtt{p}]\psi,\ \{u'\}\Delta}{\Gamma\ \vdash\ \{u\}[\mathtt{p}]\psi,\ \Delta}$$

where:

– *targets*(u) is the set of all (non-rigid) function symbols f occurring as top level operators of the left hand side of an elementary update $(f(\bar{t}) := t)$ in u

– for each $f \in \mathit{targets}(u)$: f' is a fresh rigid function symbol with the same arity as f
– the update u' is the parallel composition of the updates $\big(\mathit{for}\,\bar{x}; f(\bar{x}) := f'(\bar{x})\big)$ for all such pairs (f, f'), where $\bar{x} = x_1, \ldots, x_n$, n being the arity of f and f'
– $\mathit{Upd} = \bigwedge_{f \in \mathit{targets}(u)} \forall \bar{y}; f(\bar{y}) \doteq \{u'\}\{u\}f(\bar{y})$

Intuitively, the update u' substitutes for each updated function symbol f a fresh symbol f' which represents the old, pre-update, instance of f. The formula *Upd* links the old instances with the current ones. The following proof tree is an example:

$$\begin{array}{rc}
 & f'(\mathtt{a}) \doteq 27, \\
 & \forall y; y\,.\mathtt{f} \doteq \mathit{if}(y \doteq \mathtt{b})\mathit{then}(42)\mathit{else}(f'(y)) \\
 & \vdash\ [\mathtt{p}]\psi \\
\cline{2-2}
(\mathsf{simplifyUpdate}) & \{\mathit{for}\ x; x\,.\mathtt{f} := f'(x)\}\mathtt{a.f} \doteq 27, \\
 & \forall y; y\,.\mathtt{f} \doteq \{\mathit{for}\ x; x\,.\mathtt{f} := f'(x)\}\{\mathtt{b.f} := 42\}y\,.\mathtt{f} \\
 & \vdash\ [\mathtt{p}]\psi \\
\cline{2-2}
(\mathsf{shiftUpdate}) & \mathtt{a.f} \doteq 27\ \vdash\ \{\mathtt{b.f} := 42\}[\mathtt{p}]\psi
\end{array}$$

Since the updates resulting from this application of shiftUpdate are attached to formulas without modalities, they can be simplified away immediately, leading to a sequent without updates at all. This example also shows the disadvantage of using shiftUpdate, which is that it indirectly introduces quantifications and case distinctions for the possible aliasing situations. Using updates, instead of handling assignments in the style of shiftUpdate right away, allows to delay these complications as long as possible.

Finally, we introduce an operation setBack, which is defined as "replace a goal by one of its dominators in the proof graph". This is not strictly expressible as a sequent calculus rule, but it preserves the overall meaning of the proof: if all goals are valid, then the root must be valid. It is useful for "cutting off" proof branches which do not contribute to the loop invariant of the current loop. Such irrelevant branches for example occur when the loop body may throw an uncaught exception; the execution paths where this happens never return to the loop entry, and thus do not affect the loop invariant. Another example is the loop termination branch which is created when applying loopUnwind and subsequently ifElse. Instead of considering these side branches in every iteration of symbolic execution, they can be reverted to the loop entry with setBack. This is exemplified by the proof graph below:

$$(\mathsf{loopUnwind,\ ifElse})\,\dfrac{\varphi_1 \vdash [\mathtt{p;\ while(e)\ p}]\psi \qquad (\mathsf{setBack})\dfrac{\varphi \vdash [\mathtt{while(e)\ p}]\psi}{\varphi_2 \vdash [\,]\psi}}{\varphi \vdash [\mathtt{while(e)\ p}]\psi}$$

Instead of continuing on the right branch, it is set back to the loop entry. Once the loop body p has been symbolically executed on the left branch, merge can be used to combine both branches.

6 Proof Search Strategy

Sect. 4 has sketched the overall idea for how to apply the rules defined in Sect. 5. In this section, we concretise this aspect by defining a corresponding *proof search strategy*, i.e., an algorithm which automatically chooses the next rule to apply to a given unclosed proof. Our strategy extends a strategy able to do regular symbolic execution and first-order reasoning with the capability to infer a loop invariant whenever an invariant-less loop is encountered during proof construction.

The strategy is defined semi-formally in Fig. 1. The first three functions are helpers for the main function *chooseRuleApplication*. This function returns a pair of a goal node and a rule, with the meaning that the returned rule should be applied to the returned goal. The presentation is a bit imprecise in this respect, because in general there may of course be multiple ways to apply a single rule to a particular goal. However, for the rules that matter here, the exact application focus is either unique or it is explained in the paragraphs below. We assume that the occurring sequents are of the form $(\Gamma \vdash \{u\}[\mathsf{p}]\psi, \Delta)$, where p is the only program occurring in the sequent.

We consider a symbolic execution state, as captured by a node of the proof graph, to be "in" a loop when that loop has previously been "entered" by applying loopUnwind but not yet "left" by applying loopInvariant. Accordingly, the *entryNode* function determines the node where a specific loop, passed as a parameter to the function, has last been entered. Function *innermostLoop* returns the loop that has last been entered but not yet left.

Function *waiting* tells whether the symbolic execution of the passed node should not be continued yet, because operations on other branches have to be performed first. This is the case if the active statement is a loop, and if from the entry node of that loop it is possible to reach in the graph open goals where the active statement is not that loop: in this case, we first want to continue symbolic execution of these other goals until they get back to the loop as active statement. Only then do we continue with all of them, by combining them with merge.

The main function *chooseRuleApplication* now works as follows. First, it picks an arbitrary open goal which is not waiting for other branches. Then, it checks whether the innermost loop that symbolic execution is "in" does not occur in the program contained in the modal operator anymore. If so, this indicates that the current branch will not return to the loop entry, for example because an exception has been thrown which is not caught within the loop body. The next step is then to revert it to the entry point of the innermost loop with setBack. Otherwise, the choice of the rule depends on whether the active statement is a loop or not. If not, the strategy chooses a regular applicable symbolic execution rule or a first-order rule (abbreviated as SE in Fig. 1).

If the active statement is a loop, and if an invariant is already known for this loop, the invariant is used to apply loopInvariant. If no invariant is known, special rules are applied in a fixed order. First after reaching the loop entry via regular symbolic execution, shiftUpdate is used to get rid of any update preceding the modal operator. Then, merge can be applied to merge the current proof branch with all other branches that have been waiting for it. The next step

Pseudocode

```
//returns the node where symbolic execution entered a loop
Node entryNode(node, loop)
    if(activeStatement(node) = loop)
        if(appliedRule(node) = loopUnwind) return node;
        else if(appliedRule(node) = loopInvariant) return none;
    return entryNode(firstParent(node), loop);

//returns the innermost loop which symbolic execution is in
Loop innermostLoop(node, leftLoops)
    if(activeStatement(node) is a loop)
        if(appliedRule(node) = loopUnwind and loop $\notin$ leftLoops)
            return loop;
        else if(appliedRule(node) = loopInvariant) leftLoops := leftLoops $\cup$ {loop};
    return innermostLoop(firstParent(node), leftLoops);

//tells whether a node has to wait for other merge parents
boolean waiting(node)
    if(activeStatement(node) is a loop)
        foreach(goal reachable from entryNode(node, loop))
            if(open(goal) and activeStatement(goal) $\neq$ loop) return true;
    return false;

//main: chooses a goal and a rule which should be applied to the goal
(Node, Rule) chooseRuleApplication()
    goal := any goal with open(goal) and not waiting(goal);
    if(not occursIn(innermostLoop(goal, $\emptyset$), goal)) rule := setBack;
    else if(activeStatement(goal) is a loop)
        if(knownInvariant(loop) $\neq$ none) rule := loopInvariant;
        else lastRule := appliedRule(firstParent(goal));
            if(lastRule = SE) rule := shiftUpdate;
            else if(lastRule = shiftUpdate) rule := merge;
            else if(lastRule = merge) rule := predicateAbstraction;
            else if(lastRule = predicateAbstraction)
                if(fixed point) rule := loopInvariant;
                else rule := loopUnwind;
    else rule := SE;
    return (goal, rule);
```

Pseudocode

Fig. 1. Proof search strategy for predicate abstraction

is to perform predicate abstraction. Finally, if the iterative unwinding process has reached a fixed point, i.e., if the current abstraction (as returned by α_P) is logically equivalent to the previous abstraction for this loop, then this abstraction is an invariant for the loop. This invariant is then used to apply loopInvariant. Otherwise, one more iteration is initiated with loopUnwind.

7 Predicate Abstraction Scheme

The details of the predicate abstraction operator α_P have been left open in Sect. 5, because the approach does not depend on the use of any particular predicate abstraction algorithm. It is only necessary that $\varphi \rightarrow \alpha_P(\varphi)$ is always valid, and that the image of α_P is finite. Existing algorithms which can be used include those presented in [7] and in [10].

The approach has been implemented prototypically as an extension of the KeY system [1,3], a partly automated dynamic logic theorem prover for the verification of Java programs. This implementation uses the following simple predicate abstraction scheme: the abstraction of φ is the conjunction of all predicates from P which are implied by φ, i.e., $\alpha_P(\varphi) = \bigwedge\{p \in P \mid (\varphi \rightarrow p) \textit{ is found to be valid}\}$. This only allows conjunctions of the predicates, which is less flexible than supporting arbitrary boolean combinations. On the other hand, it is much cheaper to compute, which allows to handle a significantly higher number of predicates.

For efficiency, the implementation uses Simplify [8] instead of KeY itself for checking the validity of the formulas $\varphi \rightarrow p$. In order to keep the number of such checks down, known implication relationships between predicates are exploited: if $p_1 \rightarrow p_2$ is known to be valid a priori, and if we have been unsuccessful in proving $\varphi \rightarrow p_2$, then there is no need to check $\varphi \rightarrow p_1$.

Another aspect of practical importance are heuristics for automatically generating predicates. Our implementation features an ad hoc set of such heuristics. These take into consideration the program, the manually specified predicates, and the pre- and postcondition, and create in an exhaustive way many typical invariant components, such as ordering comparisons between pairs of integer variables, or that the value of a reference type variable is `null` or different from `null`. Extending such heuristics to cover more invariant elements is easily possible; however, increasing the number of predicates of course has an adverse effect on performance, so one has to strike a balance between power and efficiency.

8 Example

As an extended example, we walk through a proof for the Java implementation of *selection sort* shown in Fig. 2. The code is annotated with specifications written in the Java Modeling Language (JML) [13]. The `requires` and `ensures` clauses give a pre- and a postcondition for `sort`, respectively. The clause `diverges true` states that `sort` must not necessarily terminate; it is present because we are not concerned with termination issues in this paper.

No loop invariants are specified for the two loops of `sort`, instead only loop predicates are given. The syntax used for this has been proposed as an extension of JML in [10]: loop annotations starting with `loop_predicate` contain an arbitrary number of user-specified predicates for the loop, and free variables can be declared with `skolem_constant`. Fig. 2 gives exactly those predicates which are minimally necessary to make our implementation arrive at an invariant strong enough for proving the given method contract. These are supplemented by the

Java + JML

```
class Sorter {
    static int[] a;
    //@ public normal_behaviour
    //@ requires a != null;
    //@ ensures (\forall int x; 0 < x && x < a.length; a[x-1] <= a[x]);
    //@ diverges true;
    public static void sort() {
        //@ skolem_constant int x, y;
        //@ loop_predicate a[x] <= a[y];
        for(int i = 0; i < a.length; i++) {
            int minIndex = i;
            //@ skolem_constant int x;
            //@ loop_predicate a[minIndex] <= a[x];
            for(int j = i + 1; j < a.length; j++)
                if(a[j] < a[minIndex]) minIndex = j;
            int temp = a[i];
            a[i] = a[minIndex];
            a[minIndex] = temp;
} } }
```

Java + JML

Fig. 2. Java implementation of selection sort

heuristically generated predicates; for example, based on the specified predicate `a[minIndex]` $\leq$ `a[`x`]`, the essential predicate $\forall x; (0 \leq x <$ `i` $\rightarrow$ `a[minIndex]` $\leq$ `a[`x`]`) is generated automatically, together with many similar quantified formulas using different guards.

The JML specification can be translated into a DL sequent of the form $\varphi \vdash$ [`Sorter.sort();`]ψ, where φ and ψ are essentially DL representations of the `requires` clause and the `ensures` clause, respectively. Applying the predicate abstraction proof search strategy to this root sequent yields the proof graph sketched in Fig. 3.

The first step in the construction of this proof is to perform symbolic execution of the program (abbreviated as SE in the figure) until the outer loop becomes the active statement. After applying shiftUpdate and merge (in this first iteration, to only one predecessor), we perform predicate abstraction for the outer loop. Since no fixed point has yet been reached, we unwind the outer loop, creating one branch where the loop body is entered and one where the loop terminates. The latter is immediately cut off with setBack, since it will not return to the loop entry and is therefore irrelevant for the loop invariant. On the former, the body is symbolically executed, which entails dealing with the inner loop (shown in the right half of Fig. 3) and finally leads to two branches where the outer loop is again the active statement. After applying shiftUpdate to each of them, these branches can be merged, and predicate abstraction is done again. Assuming that the resulting abstraction is not equivalent to the previous one, another identical iteration is performed.

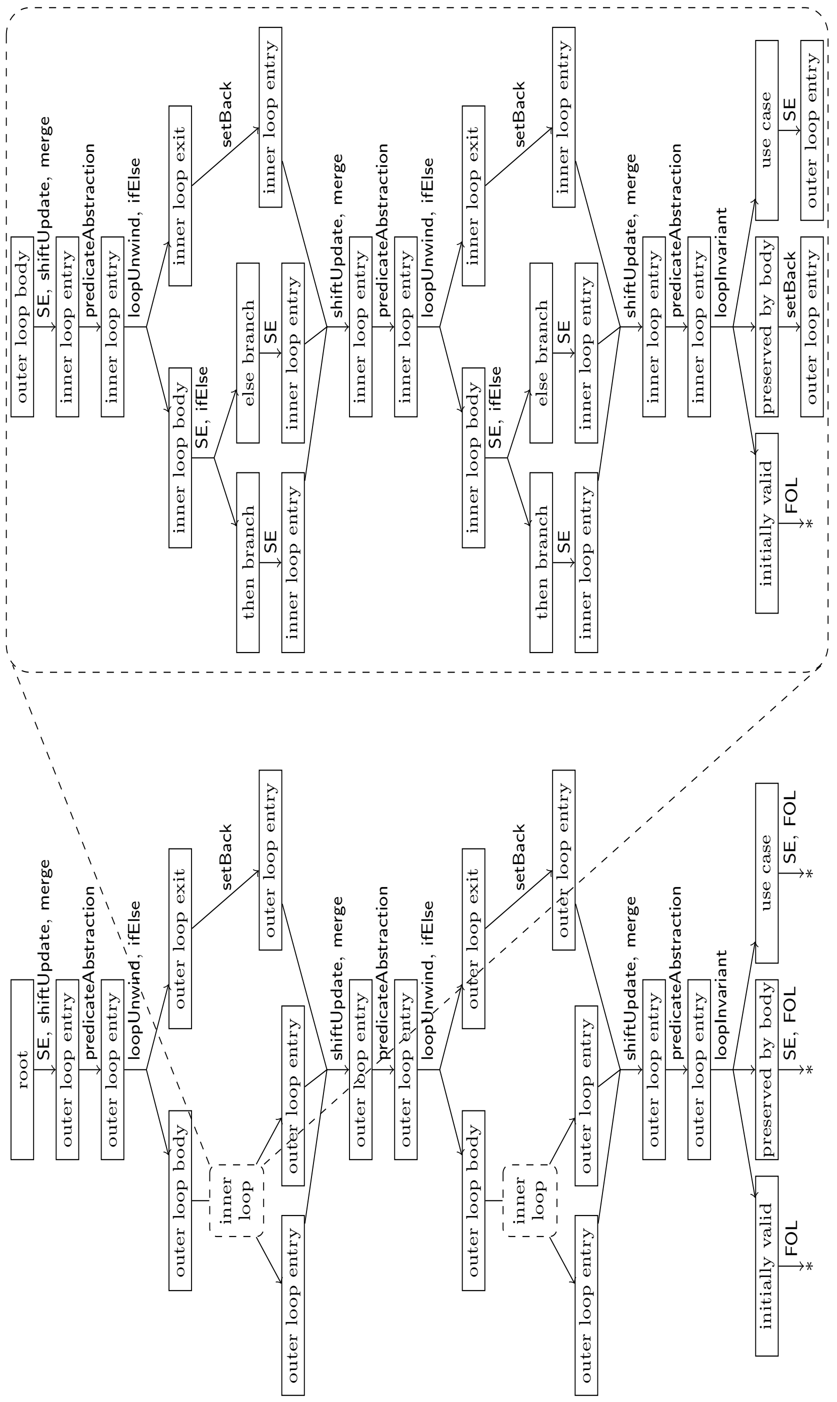

Fig. 3. Proof graph for selection sort

We assume that after this second iteration, a fixed point has been reached: the current antecedent, resulting from an application of predicateAbstraction, is logically equivalent to its counterpart in the first iteration, and is thus a loop invariant. With our implementation this inferred invariant is

$$\begin{aligned}&\forall x; \forall y; (0 \leq x \land x < y \land y < \texttt{i} \rightarrow \texttt{a[}x\texttt{]} \leq \texttt{a[}y\texttt{]})\\ &\land \forall x; \forall y; (0 \leq x < \texttt{i} \land \texttt{i} \leq y < \texttt{a.length} \rightarrow \texttt{a[}x\texttt{]} \leq \texttt{a[}y\texttt{]})\\ &\land 0 \leq \texttt{a.length} \land \texttt{i} \leq \texttt{a.length} \land 0 \leq \texttt{i} \land \texttt{a} \not\doteq \texttt{null} \land \texttt{exc} \doteq \texttt{null}\end{aligned}$$

where `exc` is a temporary variable introduced in the course of symbolic execution to buffer a possibly thrown exception. Using this for *Inv*, we apply loopInvariant. This creates three branches: the "initially valid" branch is trivial to close, because u is empty and *Inv* is identical to Γ. Proving the "preserved by body" branch entails applying loopInvariant to the inner loop, using the invariant inferred for that loop in the last iteration. As the inferred invariant is strong enough to imply the postcondition, the "use case" is closeable by further symbolic execution of the remaining program and first-order reasoning (abbreviated FOL in the figure).

The structure of the subgraph for the inner loop is analogous to the structure of the overall graph. Each time the inner loop is encountered, an invariant is inferred for it by repeated unwindings and predicate abstraction steps. The invariants inferred in the first and the second occurrence of the inner loop are different; they are dependent on the initial states occurring for the inner loop in each iteration for the outer loop. Of the three branches created by loopInvariant, the first one is again trivially closeable; the "preserved by body" branch is set back to the outer loop entry, because it does not return to that loop; and the use case is where symbolic execution actually continues back to the outer loop.

In practice, additional proof branches occur, dealing e.g. with the situation where the accessed array `a` is `null`. These are left out in Fig. 3 for simplicity. In this example, they can always be closed immediately (because the corresponding execution path is obviously infeasible), or cut off with setBack (because the execution path never returns to the respective loop entry).

9 Experiments

To give an indication of the feasibility of the approach, the results of applying the prototypical implementation to eight Java methods are listed in Table 1. For each method, the table shows its lines of combined code and specifications; the number of predicates that had to be given manually; the number of predicates that were generated automatically by the heuristics; the number of rule applications; the number of calls to Simplify for computing the predicate abstraction; and an approximate overall running time (obtained on a 1.5 GHz, 2 GB laptop).

The `getMaximumRecord` method is a simple loop which retrieves the "largest" element out of an array of objects. The second example is selection sort, as discussed in Sect. 8. The next four methods are from the Java Card API reference implementation described in [16]. These methods are simpler than selection sort

Table 1. Experimental results

	Lines	Man. prds.	Gen. prds.	Rule apps.	Simplify	Time
`LogFile::getMaximumRecord`	22	1	30	1362	41	10 s
`Sorter::sort`	22	1	1092	4594	431	90 s
`Dispatcher::dispatch`	70	0	297	2434	338	85 s
`Dispatcher::removeService`	67	1	159	3607	229	55 s
`KeyImpl::clearKey`	74	1	105	1777	252	115 s
`KeyImpl::initialize`	69	1	104	1746	242	95 s
`IntervalSeq::incSize`	33	2	178	3666	231	120 s
`Subject::registerObserver`	36	2	185	4431	242	125 s

algorithmically, but more technically involved. The last two examples are the two methods requiring loop invariants in the tutorial [1].

In all listed cases, the found invariant was strong enough to complete the verification task at hand (except for proving termination), without interaction. Manually specifying the necessary zero to two loop predicates appeared notably easier than having to provide the invariant as a whole. On the negative side, there are three additional loops in [16] for which a strong enough invariant could not be inferred. Two of them require invariants of a form (involving, e.g., existentially quantified subformulas) which are not covered by the implemented predicate abstraction scheme. The third contains deeply nested case distinctions in the loop body, which lead to large disjunctive formulas that overwhelmed Simplify.

10 Conclusions

This paper has investigated an approach for integrating abstract interpretation techniques, in particular predicate abstraction, into a calculus for deductive program verification. This allows to take advantage of the power of a deductive framework, while selectively introducing the approximation characteristic for abstract interpretation to find loop invariants automatically when necessary.

The approach consists of adding a small number of additional rules, and a dedicated proof search strategy to drive the invariant inference process. As is common for abstract interpretation, this process always finds an invariant for a loop, but this invariant is not in all cases expressive enough to be useful. Its strength heavily depends on the underlying set of loop predicates, whose elements are either generated heuristically or provided manually instead of the loop invariants themselves.

Experience with an implementation in the KeY system demonstrates the general feasibility of the approach. A line of future work is combining it with more sophisticated predicate abstraction algorithms and heuristics for generating predicates. Another possible direction is the integration of an abstraction-refinement mechanism, which would aim at systematically deriving predicates from failed proof attempts. Also, it should be possible to generalise the approach to also support other abstract domains, in addition to predicate abstraction.

References

1. Ahrendt, W., Beckert, B., Hähnle, R., Rümmer, P., Schmitt, P.H.: Verifying object-oriented programs with KeY: A tutorial. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2006. LNCS, vol. 4709, pp. 70–101. Springer, Heidelberg (2007)
2. Barnett, M., Chang, B.Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
3. Beckert, B., Hähnle, R., Schmitt, P.H. (eds.): Verification of Object-Oriented Software. The KeY Approach. LNCS, vol. 4334. Springer, Heidelberg (2007)
4. Beckert, B., Platzer, A.: Dynamic logic with non-rigid functions: A basis for object-oriented program verification. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS, vol. 4130, pp. 266–280. Springer, Heidelberg (2006)
5. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
6. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977, pp. 238–252. ACM Press, New York (1977)
7. Das, S., Dill, D.L., Park, S.: Experience with predicate abstraction. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 160–171. Springer, Heidelberg (1999)
8. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: A theorem prover for program checking. Journal of the ACM 52, 365–473 (2005)
9. Ernst, M.D., Cockrell, J., Griswold, W.G., Notkin, D.: Dynamically discovering likely program invariants to support program evolution. IEEE Transactions on Software Engineering 27, 99–123 (2001)
10. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. In: POPL 2002, pp. 191–202. ACM Press, New York (2002)
11. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
12. Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12, 576–580 (1969)
13. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. ACM Soft. Eng. Notes 31, 1–38 (2006)
14. Leino, K.R.M., Logozzo, F.: Loop invariants on demand. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 119–134. Springer, Heidelberg (2005)
15. Leino, K.R.M., Logozzo, F.: Using widenings to infer loop invariants inside an SMT solver, or: A theorem prover as abstract domain. In: WING 2007 (2007)
16. Mostowski, W.: Fully verified Java Card API reference implementation. In: Beckert, B., Beckert, B. (eds.) VERIFY 2007, vol. 259, pp. 136–151. CEUR-WS.org (2007)
17. Rümmer, P.: Sequential, parallel, and quantified updates of first-order structures. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS, vol. 4246, pp. 422–436. Springer, Heidelberg (2006)
18. Schmitt, P.H., Weiß, B.: Inferring invariants by symbolic execution. In: Beckert, B. (ed.) VERIFY 2007, vol. 259, pp. 195–210. CEUR-WS.org (2007)
19. Tiwari, A., Gulwani, S.: Logical interpretation: Static program analysis using theorem proving. In: Pfenning, F. (ed.) CADE 2007. LNCS, vol. 4603, pp. 147–166. Springer, Heidelberg (2007)