Predicate Abstraction in a Program Logic Calculus

Benjamin Weiß

Institute for Theoretical Computer Science, Karlsruhe Institute of Technology, Germany

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 based on symbolic execution, where it allows us 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 [1], such methods are predominantly based on *abstract interpretation* [2], 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 [3] 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. It is interspersed with explicit abstraction steps, which introduce the necessary approximation with the help of an automated theorem prover that determines 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 [4]. 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

Email addresses: bweiss@ira.uka.de (Benjamin Weiß)

complex, quantified invariants [5]. It can be extended with an iterative refinement process that automatically adapts the domain to the particular problem [6].

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

The present paper is an extended version of [7]. The most notable extensions are that the formal definitions of the underlying logic are included here, as well as proofs of the main theorems and a more detailed discussion of heuristics for generating loop predicates. The work underlying both papers is based on earlier work reported in [8]. Changes compared to [8] include: the soundness of all rules can now be proven, and has been; proofs are no longer necessarily tree-shaped, allowing the integration as a whole to be more natural; and the transformation of state updates into formulas is now lazy instead of eager, which improves performance.

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 more detail in Sect. 6. Sect. 7 gives some technical details on the predicate abstraction scheme used in a prototypical implementation of the approach. The overall method is further illustrated with the help of an example in Sect. 8, and practical experience with the 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 [5]. 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 [9], 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.

There are several related approaches that also aim at a closer integration between deductive verification and invariant inference. In the "loop invariants on demand" technique [10], 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 [11], 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 [12] goes the other way round by embedding theorem proving techniques in an abstract interpretation framework.

3. Background on Program Logic

The verification framework used in this paper is a program logic called *dynamic logic (DL)* [13], which is a generalisation of Hoare logic [14]. DL extends first-order logic by modal operators $[\mathtt{p}]$, where $\mathtt{p}$ can be any legal sequence of statements in some imperative programming language. A typical program verification task is to prove that under the assumption of some precondition φ, some program $\mathtt{p}$ establishes a postcondition ψ; in DL, this amounts to proving logical validity of the formula $\varphi \to [\mathtt{p}]\psi$, which is equivalent to the Hoare triple $\{\varphi\}\mathtt{p}\{\psi\}$. Unlike Hoare logic, DL is closed under its modal and logical operators; for example, the precondition φ and postcondition ψ in the above example might themselves contain programs. In the software verification systems KIV [15] and KeY [16, 17], DL is used for reasoning about Java programs.

Our flavour of DL goes beyond classical DL by featuring another form of modal operator called *updates* [18, 19]. Updates serve to express state changes in a way which is free from side effects and independent of the programming language used to write the program under verification.

In the following we formally introduce dynamic logic with updates as far as it is relevant for this work. We begin with *syntax* in Sect. 3.1, continue with *semantics* in Sect. 3.2, and conclude with a look at a suitable *calculus* in Sect. 3.3.

3.1. Syntax

Definition 1 (Signatures)**.** *A* signature *is a tuple* $(\mathcal{V}, \mathcal{F}, \mathcal{P}, \mathsf{P})$*, where* $\mathcal{V}$ *is a set of (logical) variables,* $\mathcal{F}$ *is a set of* function symbols*,* $\mathcal{P}$ *a set of* predicate symbols*, and* P *a set of* programs*. Function and predicate symbols have fixed arities. We demand that* $\mathcal{F}$ *and* $\mathcal{P}$ *contain an infinite supply of symbols of every arity.*

In the following we assume to be given a fixed signature. Based on this signature, we define the syntactical categories of *terms*, *formulas*, and *updates*.

Definition 2 (Syntax)**.** *Terms* t*, formulas* φ*, and updates* u *are defined by the following grammar, where* $x \in \mathcal{V}$ *ranges over logical variables,* $f \in \mathcal{F}$ *over*

function symbols, $p \in \mathcal{P}$ over predicate symbols, and $\mathtt{p} \in \mathit{P}$ over programs:

$$
\begin{aligned}
t &::= x \mid f(t,\ldots,t) \mid \mathit{if}(\varphi)\mathit{then}(t)\mathit{else}(t) \mid \{u\}t \\
\varphi &::= \mathit{true} \mid \mathit{false} \mid p(t,\ldots,t) \mid t \doteq t \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \rightarrow \varphi \\
&\quad\ \forall x;\varphi \mid \exists x;\varphi \mid [\mathtt{p}]\varphi \mid \{u\}\varphi \\
u &::= f(t,\ldots,t) := t \mid u \parallel u \mid \mathit{for}\ x; u
\end{aligned}
$$

Terms $f(t_1,\ldots,t_n)$, formulas $p(t_1,\ldots,t_n)$ and updates $f(t_1,\ldots,t_n) := t$ must respect the arities n of the symbols f and p.

DL formulas are evaluated in program states, which are interpretations of the function and predicate symbols. Both programs $\mathtt{p}$ and updates u are state transformers which change the interpretation of function symbols. Intuitively, an update $f(t_1,\ldots,t_n) := t$ modifies the interpretation of the function symbol f at position $(t_1,\ldots,t_n)$ to the value of t. A "parallel" update $u_1 \parallel u_2$ executes u_1 and u_2 simultaneously, while a "quantified" update $\mathit{for}\ x; u$ executes in parallel all instances of u where variable x has been instantiated with some value of the universe. For example, an update $c := d \parallel \mathit{for}\ x; f(x) := c$ sets the value of the constant symbol c to the value of d, and the value of all $f(x)$ to the old value of c. We will call the function symbols that are potentially affected by an update the *targets* of the update:

Definition 3 (Update Targets)**.** *For every update u, the* targets *function returns a set of function symbols:*

$$
\begin{aligned}
\mathit{targets}(f(\bar{t}) := t) &= \{f\} \\
\mathit{targets}(u_1 \parallel u_2) &= \mathit{targets}(u_1) \cup \mathit{targets}(u_2) \\
\mathit{targets}(\mathit{for}\ x; u) &= \mathit{targets}(u)
\end{aligned}
$$

From now on we use some vector notation for abbreviation. For example, in Def. 3, the notation $\bar{t}$ stands for $t_1,\ldots,t_n$, where n is the arity of f.

We do not specify the exact programming language used to form the programs $\mathtt{p}$ here. It might be a simple theoretical "while"-language, or a minimalist object-oriented language as in [18], or a large subset of sequential Java as in the KeY system [16]. In this paper, we only require that its state transitions can be modelled in terms of changing the interpretation of the function symbols in our signature. To this end, a local program variable `x` can be represented logically as a constant symbol $\mathtt{x} \in \mathcal{F}$, while an object field (or struct member) `f` is a function symbol $\mathtt{f} \in \mathcal{F}$ with arity 1, which maps an object to a value. In order to resemble typical programming language notation, we often denote a term `f(o)` as `o.f` for such function symbols `f`. Arrays can be modelled via a single binary function `[]`, where we typically pretty-print a term `[](a,i)` as `a[i]`. We identify side-effect free program expressions with logical terms.

3.2. Semantics

The semantics of DL formulas is based on *Kripke structures*:

Definition 4 (Kripke structures)**.** *A* Kripke structure *is a triple* $(\mathcal{D}, \mathcal{S}, \rho)$*, where* $\mathcal{D}$ *is a universe of semantical values; where* $\mathcal{S}$ *is the set of all program* states*, which are functions* $s \in \mathcal{S}$ *that map every function symbol* $f \in \mathcal{F}$ *to a function* $s(f) : \mathcal{D}^n \to \mathcal{D}$ *and every predicate symbol* $p \in \mathcal{P}$ *to a relation* $s(p) \subseteq \mathcal{D}^n$ *(where* n *is the arity of* f *and* p*, respectively); and where* ρ *is a function that associates with every program* $\mathtt{p} \in \mathit{P}$ *a* transition relation $\rho(\mathtt{p}) \subseteq \mathcal{S}^2$*.*

The function ρ represents the semantics of the programming language used to form the programs in $\mathtt{P}$: for two states $s_1, s_2 \in \mathcal{S}$, having $(s_1, s_2) \in \rho(\mathtt{p})$ means that if we execute $\mathtt{p}$ in state s_1, the execution may terminate in s_2. If $\mathtt{p}$ is deterministic, then for every starting state s_1 there is at most one such state s_2.

In the following, we assume to be given a fixed Kripke structure. Before we can define the semantics of terms, formulas and updates, we need to introduce the concept of *semantic updates*, which represent state changes on the semantic level.

Definition 5 (Semantic updates)**.** *A* semantic update *is a set* U *of tuples* $(f, \bar{v}, v)$*, where* $f \in \mathcal{F}$ *is a function symbol with arity* n*,* $\bar{v} \in \mathcal{D}^n$ *is a tuple of values, and where* v *is a value. Furthermore, a semantic update never contains* $(f, \bar{v}, v)$ *and* $(f, \bar{v}, v')$ *for values* $v, v' \in \mathcal{D}$ *with* $v \neq v'$*. This absence of "conflicts" makes it possible to use such a semantic update* U *as a state transforming function, where for each state* s *the output state* $U(s)$ *is defined by:*

$$U(s)(f)(\bar{v}) = \begin{cases} v & \text{if } (f, \bar{v}, v) \in U \\ s(f)(\bar{v}) & \text{otherwise} \end{cases}$$

for all function symbols $f \in \mathcal{F}$ *and all* $\bar{v} \in \mathcal{D}^n$ *(where* n *is the arity of* f*), and by* $U(s)(p) = s(p)$ *for all predicate symbols* $p \in \mathcal{P}$*.*

Definition 6 (Semantics)**.** *Given a state* $s \in \mathcal{S}$ *and a variable assignment* $\beta : \mathcal{V} \to \mathcal{D}$*, a term* t *is evaluated to a value* $val_{s,\beta}(t) \in \mathcal{D}$*, a formula* φ *to a truth value* $val_{s,\beta}(\varphi) \in \{tt, ff\}$*, and an update* u *to a semantic update* $val_{s,\beta}(u)$*. The evaluation is defined in Fig. 1. A formula* φ *is called (logically)* valid*, denoted* $\models \varphi$*, iff* $val_{s,\beta}(\varphi) = tt$ *for all* $s \in \mathcal{S}$ *and all* β*.*

As usual, β_x^v denotes the variable assignment which is identical to β except that $\beta_x^v(x) = v$. As defined in the figure, a formula $[\mathtt{p}]\varphi$ holds in a state if all states reachable by executing $\mathtt{p}$ in this state satisfy the postcondition φ, or in other words, if $\mathtt{p}$ is partially correct wrt. φ. Similarly, $\{u\}\varphi$ holds in a state if φ holds in the state produced by the update u. In case of a conflict between u_1 and u_2 in a parallel update $u_1 \parallel u_2$, the rightmost update u_2 "wins". For quantified updates *for* $x; u$, we do not care about their semantics in case of conflicts here, and instead view it just as an unspecified semantic update. A more precise definition can be found in [19], but does not matter here, because the quantified updates occurring in this paper never produce conflicts.

Before moving on to the calculus, we state a few observations on the above definitions that will be needed in a proof later on. We do not prove these

$$
\begin{aligned}
val_{s,\beta}(x) &= \beta(x) \\
val_{s,\beta}(f(\bar{t})) &= s(f)(val_{s,\beta}(\bar{t})) \\
val_{s,\beta}(if(\varphi)then(t_1)else(t_2)) &= \begin{cases} val_{s,\beta}(t_1) & \text{if } val_{s,\beta}(\varphi) = tt \\ val_{s,\beta}(t_2) & \text{otherwise} \end{cases} \\
val_{s,\beta}(\{u\}t) &= val_{s',\beta}(t), \text{ where } s' = val_{s,\beta}(u)(s)
\end{aligned}
$$

$$
\begin{aligned}
val_{s,\beta}(true) &= tt \\
val_{s,\beta}(false) &= ff \\
val_{s,\beta}(p(\bar{t})) &= tt \quad \text{iff} \quad val_{s,\beta}(\bar{t}) \in s(p) \\
val_{s,\beta}(t_1 \doteq t_2) &= tt \quad \text{iff} \quad val_{s,\beta}(t_1) = val_{s,\beta}(t_2) \\
val_{s,\beta}(\neg\varphi) &= tt \quad \text{iff} \quad val_{s,\beta}(\varphi) = ff \\
val_{s,\beta}(\varphi_1 \wedge \varphi_2) &= tt \quad \text{iff} \quad ff \notin \{val_{s,\beta}(\varphi_1), val_{s,\beta}(\varphi_2)\} \\
val_{s,\beta}(\varphi_1 \vee \varphi_2) &= tt \quad \text{iff} \quad tt \in \{val_{s,\beta}(\varphi_1), val_{s,\beta}(\varphi_2)\} \\
val_{s,\beta}(\varphi_1 \rightarrow \varphi_2) &= val_{s,\beta}(\neg\varphi_1 \vee \varphi_2) \\
val_{s,\beta}(\forall x; \varphi) &= tt \quad \text{iff} \quad ff \notin \{val_{s,\beta^v_x}(\varphi) \mid v \in \mathcal{D}\} \\
val_{s,\beta}(\exists x; \varphi) &= tt \quad \text{iff} \quad tt \in \{val_{s,\beta^v_x}(\varphi) \mid v \in \mathcal{D}\} \\
val_{s,\beta}([\mathtt{p}]\varphi) &= tt \quad \text{iff} \quad ff \notin \{val_{s',\beta}(\varphi) \mid (s,s') \in \rho(\mathtt{p})\} \\
val_{s,\beta}(\{u\}\varphi) &= tt \quad \text{iff} \quad val_{s',\beta}(\varphi) = tt, \text{ where } s' = val_{s,\beta}(u)(s)
\end{aligned}
$$

$$
\begin{aligned}
val_{s,\beta}(f(\bar{t}) := t) &= \big\{(f, val_{s,\beta}(\bar{t}), val_{s,\beta}(t))\big\} \\
val_{s,\beta}(u_1 \parallel u_2) &= (val_{s,\beta}(u_1) \cup val_{s,\beta}(u_2)) \setminus C, \text{ where} \\
&\qquad C = \big\{(f, \bar{v}, v) \mid (f, \bar{v}, v') \in val_{s,\beta}(u_2) \\
&\qquad\qquad \text{and } v \neq v'\big\} \\
val_{s,\beta}(for\ x; u) &= \bigcup_{v \in \mathcal{D}} val_{s,\beta^v_x}(u) \text{ if there are no conflicts}
\end{aligned}
$$

Figure 1: Semantics of terms, formulas and updates

observations themselves, but consider them obvious. First we note that certain updates replace the interpretation of one function symbol with that of another.

Proposition 1 (Function-replacing updates)**.** *If $f, f' \in \mathcal{F}$ are function symbols, and $u = \big(\mathit{for}\, \bar{x}; f(\bar{x}) := f'(\bar{x})\big)$, then for all states s, all variable assignments β, all function and predicate symbols $op \in \mathcal{F} \cup \mathcal{P}$:*

$$\mathit{val}_{s,\beta}(u)(s)(op) = \begin{cases} s(f') & \text{if } op = f \\ s(op) & \text{otherwise} \end{cases}$$

Secondly, we state that an update changes *at most* the interpretation of its targets, i.e., of the function symbols $f \in \mathit{targets}(u)$:

Proposition 2 (Non-targeted symbols)**.** *For all states s, all variable assignments β, all updates u, and all symbols $op \in (\mathcal{F} \cup \mathcal{P}) \setminus \mathit{targets}(u)$:*

$$\mathit{val}_{s,\beta}(u)(s)(op) = s(op)$$

Our final observation is that the interpretation of symbols which do not occur in a term, formula or update does not affect the evaluation of that term, formula or update. Note however that we have to exclude from this statement all symbols which can affect the interpretation of programs, such as function symbols used to represent program variables. This is because the interpretation of such symbols may affect the semantics of formulas indirectly via modal operators $[\mathtt{p}]$. For strictly logical symbols, such as fresh symbols introduced during a proof, we know that they do not affect any programs.

Proposition 3 (Non-occurring symbols)**.** *For all states $s_1, s_2 \in \mathcal{S}$, all variable assignments β, and all terms, formulas or updates a: if for all function and predicate symbols $op \in \mathcal{F} \cup \mathcal{P}$ that occur in a or whose interpretation affects $\rho(\mathtt{p})$ for any $\mathtt{p} \in \mathit{P}$ it holds that $s_1(op) = s_2(op)$, then*

$$\mathit{val}_{s_1,\beta}(a) = \mathit{val}_{s_2,\beta}(a)$$

3.3. Calculus

For mechanical reasoning about the validity of DL formulas we use a *sequent calculus*. A *sequent* is a construct $\Gamma \vdash \Delta$, where Γ (called the *antecedent*) and Δ (called the *succedent*) are finite sets of formulas. Its semantics is defined as $\mathit{val}_{s,\beta}(\Gamma \vdash \Delta) = \mathit{val}_{s,\beta}(\bigwedge \Gamma \to \bigvee \Delta)$. 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*). The rule is called *sound* iff the validity of all the premises implies the validity of the conclusion.

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 the applied rule does not have any premises, the branch is *closed*. If all branches

of a proof tree are closed and all applied rules are sound, this implies that the root sequent is logically valid.

The classical rules of a sequent calculus for first-order logic can be found, e.g., in [16, Chapt. 2]. Here, we concentrate on how to handle formulas with programs in them. For this purpose, we use rules which operate on the *active statement*, i.e., the first basic command in the modal operator, and shorten the program step by step until only a first-order problem remains. Intuitively, this process can be understood as *symbolic execution* [20]: the program is "executed", but with symbolic instead of concrete values for its variables. It is similar to the *verification condition generation* or the *strongest postcondition computation* 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 typical rules for the three elementary programming constructs of assignments, conditional statements, and loops, in a simplified Java-like language. The basic assignment rule is

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

where Γ and Δ are sets of formulas; u is an update; `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. As a border case, any of Γ, Δ and u may be empty and disappear. The rule simply transforms the *program assignment* `x = se;` into an equivalent update $\mathtt{x} := \mathtt{se}$.

This update and the preceding update u can then be aggressively simplified and normalised using a set of update rewriting rules [19]. For example, the following rule combines two updates into a single parallel one:

$$\{u\}\{f(\bar{t}) := t\} \rightsquigarrow \{u \parallel f(\{u\}\bar{t}) := \{u\}t\}$$

It is sound because by Def. 6, the rightmost sub-update of a parallel update prevails in case of a conflict. Overall, the update rewrite system establishes a normal form of updates, and immediately drops ineffective sub-updates. For simplicity, we use it as a monolithic sequent rule simplifyUpdate here, which performs several rewriting steps at once.

During the course of symbolic execution, a complex update describing the state change of the program accumulates in this way in front of the modal operator. Once the program has been dealt with completely, the final update can be applied to the postcondition as a substitution, which is also done by simplifyUpdate. As an example, consider the following unclosed proof tree (with the root at the bottom):

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash\ \mathit{if}(\mathtt{a} \doteq \mathtt{b})\mathit{then}(2)\mathit{else}(1) \doteq 1}\,(\text{simplifyUpdate})}{\vdash\ \{\mathtt{a.f} := 1 \parallel \mathtt{b.f} := 2\}(\mathtt{a.f} \doteq 1)}\,(\text{simplifyUpdate})}{\vdash\ \{\mathtt{a.f} := 1\}\{\mathtt{b.f} := 2\}(\mathtt{a.f} \doteq 1)}\,(\text{assign})}{\vdash\ \{\mathtt{a.f} := 1\}[\mathtt{b.f = 2;}]\mathtt{a.f} \doteq 1}\,(\text{simplifyUpdate})}{\vdash\ \{\mathtt{a.f} := 0\}\{\mathtt{a.f} := 1\}[\mathtt{b.f = 2;}]\mathtt{a.f} \doteq 1}\,(\text{assign})}{\vdash\ \{\mathtt{a.f} := 0\}[\mathtt{a.f = 1; b.f = 2;}]\mathtt{a.f} \doteq 1}\,(\text{assign})}{\vdash\ [\mathtt{a.f = 0; a.f = 1; b.f = 2;}]\mathtt{a.f} \doteq 1}$$

Recall that `a.f` is just a notational variation of the term `f(a)`, used in order to resemble the usual object attribute access notation. One after the other, the three assignments are turned into updates. Since the first update is overwritten by the second, it can be simplified away. Finally, the resulting update is applied to the postcondition `a.f` $\doteq 1$ as a substitution. 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:

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

For loops, the simplest approach is to *unwind* them:

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

Note that if the programming language has features such as exceptions, `break` or `continue` statements, this rule (and others) become more complex, but the basic concept remains the same.

Using loopUnwind is sufficient only for loops which terminate after a fixed, statically known number of iterations. General loops can be handled with loopInvariant:

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

Here, *Inv* is a formula acting as a loop invariant. The first two branches correspond to the base case and the step case, respectively, of an inductive argument guaranteeing that *Inv* holds at the beginning of all loop iterations. The result of this induction is used on the third branch, where we can assume that *Inv* holds after leaving the loop and before continuing program execution behind the loop. The problem with the invariant rule is that—unlike the symbolic execution rules—it can be applied automatically only if a suitable invariant *Inv* is already known for 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 some form of 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" or "join" 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 $\texttt{i} \doteq 0 \vdash [\texttt{while(i<j) ...}]\psi$, which says that we have to consider the loop in all states where `i` has the value 0. After one iteration, we might arrive at $\texttt{i} \doteq 1 \vdash [\texttt{while(i<j) ...}]\psi$, reflecting the fact that after this iteration, `i` has been incremented by one. "Merging" these sequents will combine the antecedents disjunctively, yielding the sequent $\texttt{i} \doteq 0 \lor \texttt{i} \doteq 1 \vdash [\texttt{while(i<j) ...}]\psi$. Thus, we know that after up to one iteration through the loop, the value of `i` is either 0 or 1.

With every such iteration of unwinding, symbolically executing and merging, the set of states that are deemed possible for the loop entry point becomes larger. 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 corresponds to a computation of the *static semantics*. Obviously, the infinite number of states means that for most loops, such a computation will not terminate. To change this, we need to introduce *approximation*. A form of approximation particularly suitable in our context is that of predicate abstraction [3, 5]: 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. 3.3 with predicate abstraction based loop invariant inference as described in Sect. 4.

5.1. Merging Proof Branches

First is a rule for merging execution paths at junction points in the control flow graph, called merge:

$$\text{merge}\quad \frac{\bigwedge(\Gamma_1 \cup \neg\Delta_1) \vee \dots \vee \bigwedge(\Gamma_n \cup \neg\Delta_n) \;\vdash\; \psi}{\Gamma_1 \;\vdash\; \psi, \Delta_1 \qquad \dots \qquad \Gamma_n \;\vdash\; \psi, \Delta_n}$$

where $\neg\Delta_i$ stands for the set $\{\neg\delta \mid \delta \in \Delta_i\}$. 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

$$(\text{merge})\quad \frac{\varphi_1 \vee \varphi_2 \vdash [\texttt{while(e) p}]\psi}{\varphi_1 \vdash [\texttt{while(e) p}]\psi \qquad \varphi_2 \vdash [\texttt{while(e) p}]\psi}$$

Lemma 1 (Soundness of merge)**.**

$$\models \bigwedge(\Gamma_1 \cup \neg\Delta_1) \vee \dots \vee \bigwedge(\Gamma_n \cup \neg\Delta_n) \vdash \psi \tag{1}$$

implies

$$\models \Gamma_1 \vdash \psi, \Delta_1 \qquad \dots \qquad \Gamma_n \vdash \psi, \Delta_n$$

Proof. Assume (1) holds. Let $s \in \mathcal{S}$ be a state, β a variable assignment, and $i \in \{1, \dots, n\}$. We need to show $val_{s,\beta}(\Gamma_i \vdash \psi, \Delta_i) = tt$. If there is $\gamma \in \Gamma_i$ with $val_{s,\beta}(\gamma) = ff$ or if there is $\delta \in \Delta_i$ with $val_{s,\beta}(\delta) = tt$, this is trivially true. We therefore assume $val_{s,\beta}(\bigwedge(\Gamma_i \cup \neg\Delta_i)) = tt$, and aim to show $val_{s,\beta}(\psi) = tt$. This follows immediately from (1). □

5.2. Predicate Abstraction

The next rule is responsible for the predicate abstraction step:

$$\textsf{predicateAbstraction} \quad \frac{\alpha_P(\bigwedge(\Gamma \cup \neg\Delta)) \;\vdash\; [\texttt{while(e) p}\ \omega]\psi}{\Gamma \;\vdash\; [\texttt{while(e) p}\ \omega]\psi, \Delta}$$

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 $\varphi \to \alpha_P(\varphi)$ 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.

Note that the semantics of the `while` loop occurring in predicateAbstraction has not been formally defined. This does not matter, because the rule only uses the loop as the provider of a set P of loop predicates and is otherwise independent from the form of the program in the sequent.

Lemma 2 (Soundness of predicateAbstraction)**.**

$$\models \varphi \to \alpha_P(\varphi) \quad \textit{for all } \varphi \tag{2}$$

and

$$\models\ \alpha_P(\bigwedge(\Gamma \cup \neg\Delta)) \vdash [\texttt{while(e) p}\ \omega]\psi \tag{3}$$

together imply

$$\models\ \Gamma \vdash [\texttt{while(e) p}\ \omega]\psi, \Delta$$

Proof. Assume (2) and (3) hold. Let $s \in \mathcal{S}$ be a state and β a variable assignment. We need to show $val_{s,\beta}(\Gamma \vdash [\texttt{while(e) p}\ \omega]\psi, \Delta) = tt$. If there is $\gamma \in \Gamma$ with $val_{s,\beta}(\gamma) = ff$ or if there is $\delta \in \Delta$ with $val_{s,\beta}(\delta) = tt$, this is trivially true. We therefore assume $val_{s,\beta}(\bigwedge(\Gamma \cup \neg\Delta)) = tt$, and aim to show $val_{s,\beta}([\texttt{while(e) p}\ \omega]\psi) = tt$.
By (2), we know that $\models \bigwedge(\Gamma \cup \neg\Delta) \to \alpha_P(\bigwedge(\Gamma \cup \neg\Delta))$. Thus, we have $val_{s,\beta}(\alpha_P(\bigwedge(\Gamma \cup \neg\Delta))) = tt$. Together with (3), this yields the desired result $val_{s,\beta}([\texttt{while(e) p}\ \omega]\psi) = tt$. □

5.3. Handling Updates

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 the modality $[\mathtt{p}]$. This can be achieved with the shiftUpdate rule:

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

where:

- for each $f \in \mathit{targets}(u)$: $f' \in \mathcal{F}$ is a fresh 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'), in an arbitrary order
- $\mathit{Upd} = \bigwedge_{f \in \mathit{targets}(u)} \forall \bar{y}; f(\bar{y}) \doteq \{u'\}\{u\}f(\bar{y})$

Intuitively, the update u' substitutes for each updated function symbol f (as defined in Def. 3) 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 new antecedent $(\{u'\}\Gamma,\ \mathit{Upd})$ is the strongest postcondition of Γ under u; as a whole, the shiftUpdate rule is closely related to the classical strongest postcondition rule for assignments, which in the same way introduces a fresh name for the old instance of the assigned program variable (or which, in other words, existentially quantifies the old instance of the assigned variable). The following proof tree is an example:

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

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 applying shiftUpdate, which is that it indirectly introduces quantifications and case distinctions for the possible aliasing situations. Using updates—instead of handling assignments in a strongest postcondition style right away—allows us to delay these complications as long as possible. However, the approach of the paper is independent of the choice of using updates, and would still be valid in an update-less setting.

Lemma 3 (Soundness of shiftUpdate)**.**

$$\begin{array}{c} \models\ \{u'\}\Gamma, \mathit{Upd} \vdash [\mathtt{p}]\psi, \{u'\}\Delta \\ \mathit{implies} \\ \models\ \Gamma \vdash \{u\}[\mathtt{p}]\psi, \Delta \end{array} \tag{4}$$

Proof. Assume (4) holds. Let $s \in \mathcal{S}$ be a state and β a variable assignment. We need to show $\mathit{val}_{s,\beta}(\Gamma \vdash \{u\}[\mathtt{p}]\psi, \Delta) = \mathit{tt}$. If there is $\gamma \in \Gamma$ with $\mathit{val}_{s,\beta}(\gamma) = \mathit{ff}$ or if there is $\delta \in \Delta$ with $\mathit{val}_{s,\beta}(\delta) = \mathit{tt}$, this is trivially true. We therefore

assume $val_{s,\beta}(\bigwedge(\Gamma \cup \neg\Delta)) = tt$, and aim to show $val_{s,\beta}(\{u\}[\mathtt{p}]\psi) = tt$.
Let $U = val_{s,\beta}(u)$, and let the state $s' \in \mathcal{S}$ be defined as follows:

$$s'(op) = \begin{cases} s(f) & \text{if } op = f' \text{ for some } f \in targets(u) \\ U(s)(op) & \text{otherwise} \end{cases}$$

That is, s' is identical to $U(s)$ except that the fresh function symbols f' are interpreted like the corresponding f are interpreted in s. We are now going to show that $val_{s',\beta}(\{u'\}\bigwedge(\Gamma\cup\neg\Delta)) = tt$, i.e., that the changes we made to obtain s' and the update u' "cancel each other out" wrt. the validity of $\bigwedge(\Gamma \cup \neg\Delta)$.
Let $U' = val_{s',\beta}(u')$, and $s'' = U'(s')$. By Prop. 1, s'' satisfies

$$s''(op) = \begin{cases} s'(f') & \text{if } op = f \in targets(u) \\ s'(op) & \text{otherwise} \end{cases}$$

By definition of s', this is the same as

$$s''(op) = \begin{cases} s(op) & \text{if } op \in targets(u) \\ s(f) & \text{if } op = f' \text{ for some } f \in targets(u) \\ U(s)(op) & \text{otherwise} \end{cases}$$

With the help of Prop. 2 we can simplify this to

$$s''(op) = \begin{cases} s(f) & \text{if } op = f' \text{ for some } f \in targets(u) \\ s(op) & \text{otherwise} \end{cases} \tag{5}$$

That is, s'' is identical to s except for the interpretation of the fresh function symbols f'.
Since these symbols do not occur in Γ, Δ or any programs, and since we know that $val_{s,\beta}(\bigwedge(\Gamma\cup\neg\Delta)) = tt$, it follows by Prop. 3 that $val_{s'',\beta}(\bigwedge(\Gamma\cup\neg\Delta)) = tt$, and consequently (by choice of s'') we get the desired property

$$val_{s',\beta}(\{u'\}\bigwedge(\Gamma \cup \neg\Delta)) = tt \tag{6}$$

Let $f \in targets(u)$. (5) tells us that $s''(f) = s(f)$. Therefore $U(s'')(f) = U(s)(f)$. Independently, the definition of s' yields $s'(f) = U(s)(f)$. Combined, we have $s'(f) = U(s'')(f)$, which by definition of s'' is the same as $s'(f) = U(U'(s'))(f)$. Since this holds for all $f \in targets(u)$ (and since by Prop. 3 $U = val_{s'',\beta}(u)$), this implies

$$val_{s',\beta}(Upd) = tt \tag{7}$$

Together, (4), (6) and (7) imply $val_{s',\beta}([\mathtt{p}]\psi) = tt$. Since s' is identical to $U(s)$ except in the f' which do not occur in $[\mathtt{p}]\psi$ or any programs, this implies by Prop. 3 that $val_{U(s),\beta}([\mathtt{p}]\psi) = tt$, or equivalently, $val_{s,\beta}(\{u\}[\mathtt{p}]\psi) = tt$. □

5.4. Setting Back Proof Branches

The symbolic execution during invariant inference sometimes creates proof branches that do not contribute to the loop invariant and which we thus do not want to follow up on. For example, such irrelevant branches occur when the loop body throws 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, we will reverted them to the loop entry with an operation setBack, that we informally define as

"replace a goal by any of its dominators in the proof graph".

As usual, a *dominator* of a node n is a node n' with the property that every path from the root to n must pass through n'. As an example for setBack, consider the proof graph below:

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

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.

The setBack operation can be seen as a non-destructive form of backtracking. It is not expressible as a sequent calculus rule in the regular sense, but it preserves the overall meaning of the proof: if all goals are valid, then the root must be valid.

Lemma 4 (Soundness of setBack)**.** *Every proof graph which is constructed by applying rules that are sound in the traditional sense and the* setBack *operation satisfies: if all goal sequents are valid, then the root sequent is valid.*

Proof sketch. For proof graphs consisting just of a root node, the proposition is trivially satisfied.
As an induction hypothesis, assume that we are given a proof graph p with root r and goals G for which all sub proof graphs (including p itself) satisfy the proposition. We need to show that the graph p' with goals G' resulting from applying setBack to one of the goals $g \in G$ again satisfies the proposition, i.e., that the validity of all G' implies the validity of r.
By definition of setBack, $G' = (G \setminus \{g\}) \cup \{g'\}$, where g' corresponds to the same sequent as some node d which dominates g. Consider the subgraph p_d resulting from cutting off in p all nodes strictly dominated by d. For the goals G_d of p_d we know: $G_d \subseteq (G \setminus \{g\}) \cup \{d\}$ (because g has been cut off, while d has become a leaf). By the induction hypothesis, we know that the validity of G_d implies the validity of r. Since the sequents corresponding to G' are a superset of the sequents corresponding to G_d, this means that also the validity of G' implies the validity of r. □

6. Proof Search Strategy

Sect. 4 has sketched the overall idea of 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 be applied 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 Figs. 2 and 3. The three functions in Fig. 2 are helpers for the main function in Fig. 3. This main 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 $(\Gamma \vdash \{u\}[\mathtt{p}]\psi, \Delta)$, where $\mathtt{p}$ is the only program occurring in the sequent. This assumption holds throughout typical Hoare-like proofs, e.g. in the KeY system.

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 rule applications 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 yet 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. In this way, we turn the entry points of loops into "synchronisation points", where different proof branches belonging to the same loop—to which rules are otherwise applied independently in an unspecified order—wait for each other. Only when all of them are ready do we continue with the waiting branches, by combining them all 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" (if any) 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. 3).

—— Pseudocode ——

```
//returns the node where symbolic execution entered a loop
Node entryNode(Node node, Loop 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 node, SetOfLoop leftLoops)
    if(activeStatement(node) is a loop)
        Loop loop := activeStatement(node)
        if(appliedRule(node) = loopUnwind and loop ∉ leftLoops)
            return loop;
        else if(appliedRule(node) = loopInvariant)
            leftLoops := leftLoops ∪ {loop};
    return innermostLoop(firstParent(node), leftLoops);

//tells whether a node has to wait for other merge parents
boolean waiting(Node node)
    if(activeStatement(node) is a loop)
        Loop loop := activeStatement(node)
        foreach(goal reachable from entryNode(node, loop))
            if(open(goal) and activeStatement(goal) ≠ loop) return true;
    return false;
```

—— Pseudocode ——

Figure 2: Proof search strategy for predicate abstraction: helper functions

If the active statement is a loop, and if an invariant is already known for this loop, this 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 is to perform predicate abstraction. Finally, we check whether the iterative unwinding process has reached a fixed point, i.e., whether the current abstraction implies the previous abstraction for this loop. The "abstraction" is the context formula of the sequent, as produced by α_P; for example, in a sequent $\varphi \vdash [\mathsf{p}]\psi$ resulting from predicateAbstraction, it is the formula φ. The initial abstraction before the first iteration is simply *false*—the strongest possible invariant, which is then weakened in each iteration (unless the loop is unreachable). If the current abstraction is indeed a fixed point, then it is used as an invariant in the loopInvariant rule. Otherwise, one more iteration is initiated with loopUnwind.

```
// chooses a goal and a rule which should be applied to the goal
(Node, Rule) chooseRuleApplication()
    Node goal := any goal with open(goal) and not waiting(goal);
    if(not occursIn(innermostLoop(goal, ∅), goal)) return (goal, setBack);
    else if(activeStatement(goal) is a loop)
        Loop loop := activeStatement(goal);
        Node entry := entryNode(goal, loop);
        Rule lastRule := appliedRule(firstParent(goal));
        if(knownInvariant(loop) ≠ none)
            return (goal, loopInvariant[inv=knownInvariant(loop)]);
        else if(lastRule = SE) return (goal, shiftUpdate);
        else if(lastRule = shiftUpdate) return (goal, merge);
        else if(lastRule = merge) return (goal, predicateAbstraction);
        else if(lastRule = predicateAbstraction)
            if(isValid(formula(goal) → formula(entry)))
                return (goal, loopInvariant[inv=formula(goal)]);
            else return (goal, loopUnwind);
    else return (goal, SE);
```

Figure 3: Proof search strategy for predicate abstraction: main function

Note that the other "direction" of implication always holds, i.e., the current abstraction is always implied by the previous one. This is because in each iteration, the new abstraction results from disjunctively combining several proof branches, including at least one which corresponds to the previous abstraction. Also note that checking whether the current abstraction implies the previous one is a comparatively simple task: since both formulas are built from the same set P of loop predicates, this check only requires propositional reasoning, not full first-order theorem proving (unlike the computation of α_P itself).

7. Implementational Details

7.1. Predicate Abstraction

So far, we have avoided looking into the details of the predicate abstraction operator α_P. This is because the contribution of this paper lies not in a way of *performing* predicate abstraction, but in the *integration* of predicate abstraction into the calculus and the overall verification methodology. The approach only requires that $\varphi \to \alpha_P(\varphi)$ is always valid, and that the image of α_P is finite. Nevertheless, computing α_P is non-trivial. Typically it is by far the most computationally expensive operation of the whole inference/verification process, because it requires many theorem prover queries of the form "does a imply b?", where a and b are first-order formulas.

Several algorithms for doing predicate abstraction are available in the literature (see for example [21, 5]). The prototypical implementation of our approach in the Java verification system KeY, which is the basis for the experiments in Sect. 9, uses a somewhat more naive scheme than what is proposed in current papers. For us, the abstraction of a formula φ is the conjunction of all predicates from P which are implied by φ, i.e., $\alpha_P(\varphi) = \bigwedge\{p \in P \mid (\varphi \rightarrow p)$ *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 us to handle a significantly higher number of predicates.

For efficiency, our implementation uses the Simplify prover [22] instead of KeY itself for checking the validity of the formulas $\varphi \rightarrow p$. This is in fact against the general spirit of our approach: we want to integrate everything into a *single* prover, avoiding the duplication of knowledge that is present in related approaches. However, this is an implementational decision, which would not be necessary if the used program logic prover was more optimised towards speed than KeY currently is.

In order to keep the number of calls to Simplify down, the implementation exploits some known implication relationships between predicates: 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 $\varphi \rightarrow p_1$. Also, predicates that were already found to be not valid in a previous iteration for a loop do not need to be checked again.

7.2. Generating Loop Predicates

Besides the computation of α_P, another aspect of practical importance is how to automatically generate a useful set P of loop predicates. Our implementation features an ad hoc set of heuristics for this purpose. They are run immediately before the first application of predicateAbstraction to a particular loop. Based on the current sequent "$\Gamma \vdash [\texttt{while(e) p}\,\omega]\psi, \Delta$" and on the loop predicates manually specified by the user (if any), they create in an exhaustive way many typical invariant components. The following paragraphs describe these heuristics in more detail. Note that, unlike the simplified logic presented in Sect. 3, the logic of KeY [16, Chapt. 3] is a *typed* logic, whose types correspond to those of the verified Java program. As the predicate generation makes use of type information, there will be some mention of types below.

As a first step, we identify those local program variables that occur both in $\Gamma \cup \Delta$ and in $[\texttt{while(e) p}\,\omega]\psi$. These are the only program variables that are interesting at the current program point, since (i) no information is available about those not in $\Gamma \cup \Delta$, and (ii) those not in $[\texttt{while(e) p}\,\omega]\psi$ are irrelevant for both the further excution of the program and for the postcondition ψ. These program variables, together with the constant symbols 0 and `null` (Java's null reference), are used to form an initial set of terms.

Next, we extend this set by applying to all terms in the set all suitably typed function symbols that represent Java fields, as well as the array access operator `[]` (Sect. 3.1). For example, if the original set contains program variables `o`

and `a`, terms like `o.f` (where `f` is a Java field defined for the type of `o`), `a[0]` and `a.length` (where the type of `a` is an array type) are added. The current implementation does exactly one such step of "heap indirection", but in general of course an arbitrary number is possible.

We then generate the following predicates for all boolean terms b in the set, for all integer terms i_1, i_2, i_3, i_4, for all reference terms o_1, o_2, for all arithmetic relations $\lhd_1, \lhd_2, \lhd_3, \lhd_4 \in \{<, \leq\}$, and for all user-specified predicates $p_1(x), p_2(x, y)$ containing one free variable x or two free variables x and y, respectively:

- $b \doteq \texttt{true}$, $b \doteq \texttt{false}$
- $i_1 \lhd_1 i_2$
- $o_1 \doteq o_2$, $\neg o_1 \doteq o_2$
- $\forall x; (i_1 \lhd_1 x \wedge x \lhd_2 i_2 \rightarrow p_1(x))$
- $\forall x; \forall y; (i_1 \lhd_1 x \wedge x \lhd_2 y \wedge y \lhd_3 i_2 \rightarrow p_2(x, y))$
- $\forall x; \forall y; (i_1 \lhd_1 x \wedge x \lhd_2 i_2 \wedge i_3 \lhd_3 y \wedge y \lhd_4 i_4 \rightarrow p_2(x, y))$

The last three cases can lead to large numbers of predicates. For example, the number of predicates created by the very last case for each user predicate $p_2(x, y)$ is $2^4 * n^4$, where n is the number of integer terms in the set. Some of these predicates imply others, which is exploited by our predicate abstraction implementation to avoid some validity checks.

In addition to the above predicates, we use each elementary conjunct of the postcondition ψ as a loop predicate. Finally, we derive a special predicate from the postcondition in the following common case: frequently, the loop guard is a binary formula such as $i < n$, while the postcondition contains a guarded quantification such as $\forall x; (\varphi_1(x) \wedge x < n \rightarrow \varphi_2(x))$, where the quantified variable x ranges up to the same boundary n as the variable i does in the loop. In this case, we add a loop predicate $\forall x; (\varphi_1(x) \wedge x < i \rightarrow \varphi_2(x))$, which expresses the likely guess that, in each loop iteration, property $\varphi_2(x)$ has already been established for all x up until i.

Extending and tuning these heuristics to cover more invariant elements is possible quite easily. This flexibility, which enables us to quickly adapt the class of inferrable invariants to a new problem domain, is one of the main advantages of predicate abstraction. However, increasing the number of predicates of course has an adverse effect on performance, so one has to strike a balance there between power and efficiency. An alternative to heuristically generating predicates, which has gained a lot of popularity in recent years, is attempting to infer the needed predicates *systematically* from failed proof attempts (see e.g. [6, 23]). Combining such an iterative "counterexample-guided abstraction refinement" (CEGAR) technique with our approach remains as future work.

8. Example

As an extended example, we walk through a proof for the Java implementation of *selection sort* shown in Fig. 4. The code is annotated with specifications written in the Java Modeling Language (JML) [24]. 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. The keyword `normal_behaviour` expresses that if the precondition holds, then the method is not allowed to terminate by throwing an exception.

—— 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 ——

Figure 4: Java implementation of selection sort

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 [5]: 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. 4 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 predicates generated by the heuristics of Sect. 7.2; for example, based on the specified predicate `a[minIndex]` $\leq$ `a[`x`]`, the essential predicate

$\forall x; (0 \leq x \land x < \texttt{i} \rightarrow \texttt{a[minIndex]} \leq \texttt{a}[x])$ is generated automatically, together with many similar quantified formulas using different guards. For arriving at the predicate $\texttt{a[minIndex]} \leq \texttt{a}[x]$, the user needs the intuition that the array is supposed to contain a value at position `minIndex` that is smaller than its values at other indices, and that this may be relevant for the verification of the loop.

The JML specification can be translated into a DL sequent of the form $\varphi \vdash [\texttt{Sorter.sort();}]\psi$, 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. 5.

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. 5) 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.

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. This is what happens with our implementation, where the inferred invariant is

$$\begin{aligned}
&\forall x; \forall y; (0 \leq x \land x < y \land y < \texttt{i} \rightarrow \texttt{a}[x] \leq \texttt{a}[y]) \\
&\land \forall x; \forall y; (0 \leq x \land x < \texttt{i} \land \texttt{i} \leq y \land y < \texttt{a.length} \rightarrow \texttt{a}[x] \leq \texttt{a}[y]) \\
&\land 0 \leq \texttt{a.length} \land \texttt{i} \leq \texttt{a.length} \land 0 \leq \texttt{i} \land \neg \texttt{a} \doteq \texttt{null} \land \texttt{exc} \doteq \texttt{null}
\end{aligned}$$

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

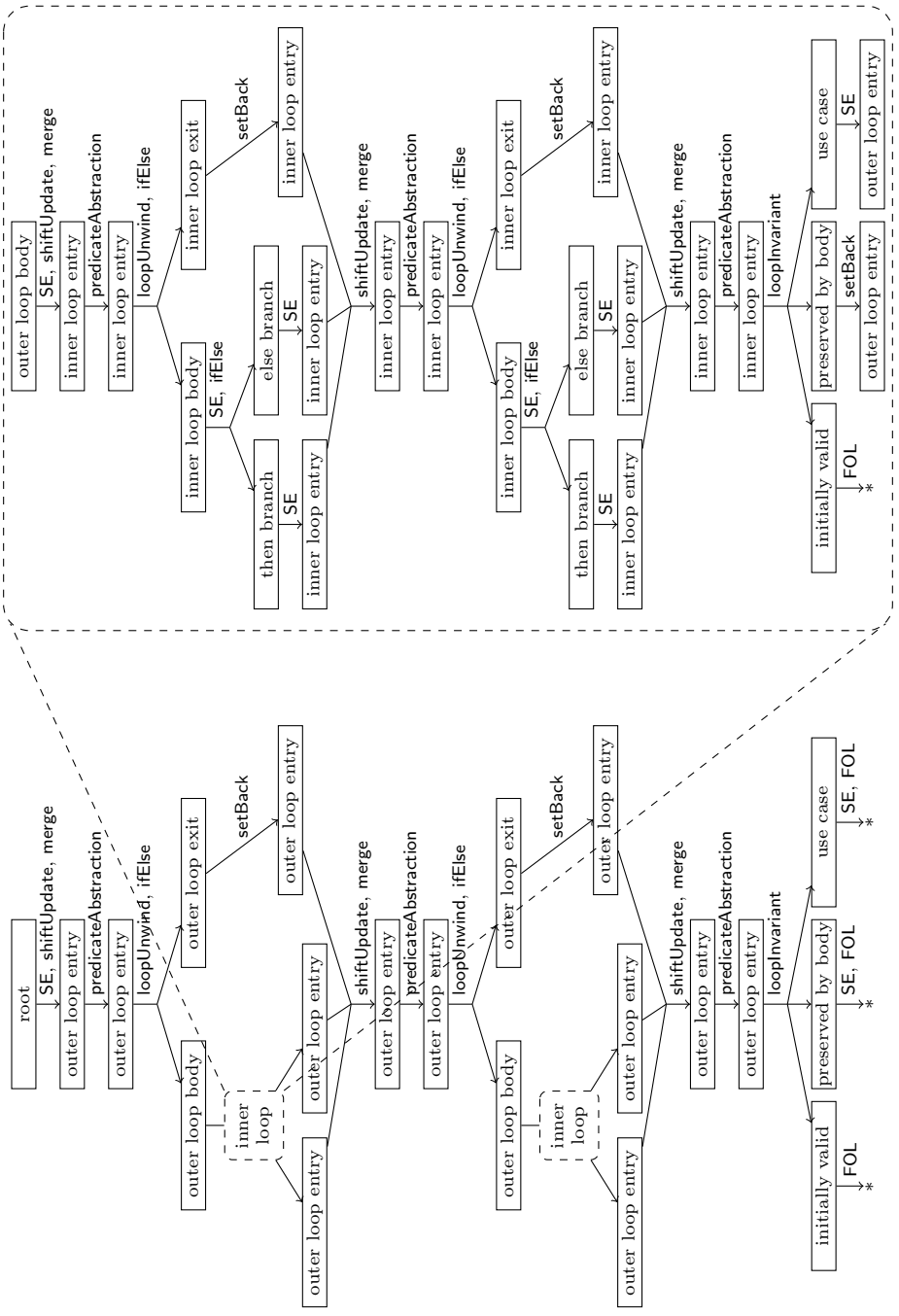

Figure 5: Proof graph for selection sort

Table 1: Experimental results

	Lines	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

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. 5 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 the number of 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 [25]. These methods are simpler than selection sort algorithmically, but technically more involved. The last two examples are the two methods requiring loop invariants in the tutorial [17].

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, in a similar way as in the selection sort example. On the negative side, there are three additional loops in [25] 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 us to take advantage of the power of a deductive framework, while selectively introducing the approximation that is 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, i.e., expressive enough to prove the desired postcondition. In this case, user intervention is required; the generated invariant, even though too weak, may be helpful in figuring out what to do. The strength of the found invariants heavily depends on the underlying set of loop predicates, whose elements are either generated heuristically or provided manually in place 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 support other abstract domains, in addition to predicate abstraction.

Acknowledgments

I would like to thank Peter H. Schmitt and Philipp Rümmer for valuable discussions, and the anonymous reviewers for their comments which helped to improve the paper.

References

[1] M. D. Ernst, J. Cockrell, W. G. Griswold, D. Notkin, Dynamically discovering likely program invariants to support program evolution, IEEE Transactions on Software Engineering 27 (2) (2001) 99–123.

[2] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints,

in: Proceedings, 4th ACM Symposium on Principles of Programming Languages (POPL 1977), ACM Press, 1977, pp. 238–252.

[3] S. Graf, H. Saïdi, Construction of abstract state graphs with PVS, in: O. Grumberg (Ed.), Proceedings, 9th International Conference on Computer Aided Verification (CAV 1997), Vol. 1254 of LNCS, Springer, 1997, pp. 72–83.

[4] P. Cousot, R. Cousot, Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, in: M. Bruynooghe, M. Wirsing (Eds.), Proceedings, 4th International Symposium on Programming Language Implementation and Logic Programming (PLILP 1992), Vol. 631 of LNCS, Springer, 1992, pp. 269–295.

[5] C. Flanagan, S. Qadeer, Predicate abstraction for software verification, in: Proceedings, 29th ACM Symposium on Principles of Programming Languages (POPL 2002), ACM Press, 2002, pp. 191–202.

[6] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement, in: E. A. Emerson, A. P. Sistla (Eds.), Proceedings, 12th International Conference on Computer Aided Verification (CAV 2000), Vol. 1855 of LNCS, Springer, 2000, pp. 154–169.

[7] B. Weiß, Predicate abstraction in a program logic calculus, in: M. Leuschel, H. Wehrheim (Eds.), Proceedings, 7th International Conference on integrated Formal Methods (iFM 2009), Vol. 5423 of LNCS, Springer, 2009, pp. 136–150.

[8] P. H. Schmitt, B. Weiß, Inferring invariants by symbolic execution, in: B. Beckert (Ed.), Proceedings, 4th International Verification Workshop (VERIFY'07), Vol. 259 of CEUR Workshop Proceedings, CEUR-WS.org, 2007, pp. 195–210.

[9] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, K. R. M. Leino, Boogie: A modular reusable verifier for object-oriented programs, in: F. S. de Boer, M. M. Bonsangue, S. Graf, W.-P. de Roever (Eds.), Revised Lectures, 4th International Symposium on Formal Methods for Components and Objects (FMCO 2005), Vol. 4111 of LNCS, Springer, 2006, pp. 364–387.

[10] K. R. M. Leino, F. Logozzo, Loop invariants on demand, in: K. Yi (Ed.), Proceedings, 3rd Asian Symposium on Programming Languages and Systems (APLAS 2005), Vol. 3780 of LNCS, Springer, 2005, pp. 119–134.

[11] K. R. M. Leino, F. Logozzo, Using widenings to infer loop invariants inside an SMT solver, or: A theorem prover as abstract domain, in: Proceedings, 1st International Workshop on Invariant Generation (WING 2007), 2007.

[12] A. Tiwari, S. Gulwani, Logical interpretation: Static program analysis using theorem proving, in: F. Pfenning (Ed.), Proceedings, 21st International Conference on Automated Deduction (CADE-21), Vol. 4603 of LNCS, Springer, 2007, pp. 147–166.

[13] D. Harel, D. Kozen, J. Tiuryn, Dynamic Logic, MIT Press, 2000.

[14] C. A. R. Hoare, An axiomatic basis for computer programming, Communications of the ACM 12 (10) (1969) 576–580.

[15] M. Balser, W. Reif, G. Schellhorn, K. Stenzel, A. Thums, Formal system development with KIV, in: T. S. E. Maibaum (Ed.), Proceedings, 3rd International Conference on Fundamental Approaches to Software Engineering (FASE 2000), Vol. 1783 of LNCS, Springer, 2000, pp. 363–366.

[16] B. Beckert, R. Hähnle, P. H. Schmitt (Eds.), Verification of Object-Oriented Software: The KeY Approach, Vol. 4334 of LNCS, Springer, 2007.

[17] W. Ahrendt, B. Beckert, R. Hähnle, P. Rümmer, P. H. Schmitt, Verifying object-oriented programs with KeY: A tutorial, in: F. S. de Boer, M. M. Bonsangue, S. Graf, W.-P. de Roever (Eds.), Revised Lectures, 5th International Symposium on Formal Methods for Components and Objects (FMCO 2006), Vol. 4709 of LNCS, Springer, 2008, pp. 70–101.

[18] B. Beckert, A. Platzer, Dynamic logic with non-rigid functions: A basis for object-oriented program verification, in: U. Furbach, N. Shankar (Eds.), Proceedings, 3rd International Joint Conference on Automated Reasoning (IJCAR 2006), Vol. 4130 of LNCS, Springer, 2006, pp. 266–280.

[19] P. Rümmer, Sequential, parallel, and quantified updates of first-order structures, in: M. Hermann, A. Voronkov (Eds.), Proceedings, 13th International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR 2006), Vol. 4246 of LNCS, Springer, 2006, pp. 422–436.

[20] J. C. King, Symbolic execution and program testing, Communications of the ACM 19 (7) (1976) 385–394.

[21] S. Das, D. L. Dill, S. Park, Experience with predicate abstraction, in: N. Halbwachs, D. Peled (Eds.), Proceedings, 11th International Conference on Computer Aided Verification (CAV 1999), Vol. 1633 of LNCS, Springer, 1999, pp. 160–171.

[22] D. Detlefs, G. Nelson, J. B. Saxe, Simplify: A theorem prover for program checking, Journal of the ACM 52 (3) (2005) 365–473.

[23] D. Beyer, T. A. Henzinger, R. Jhala, R. Majumdar, Checking memory safety with Blast, in: M. Cerioli (Ed.), Proceedings, 8th International Conference on Fundamental Approaches to Software Engineering (FASE 2005), Vol. 3442 of LNCS, Springer, 2005, pp. 2–18.

[24] G. T. Leavens, A. L. Baker, C. Ruby, Preliminary design of JML: A behavioral interface specification language for Java, ACM SIGSOFT Software Engineering Notes 31 (3) (2006) 1–38.

[25] W. Mostowski, Fully verified Java Card API reference implementation, in: B. Beckert (Ed.), Proceedings, 4th International Verification Workshop (VERIFY'07), Vol. 259 of CEUR Workshop Proceedings, CEUR-WS.org, 2007, pp. 136–151.