Predicate Transformation as a Proof Strategy

Nicole Rauch[1] and Arnd Poetzsch-Heffter[2]

[1] `rauch@informatik.uni-kl.de`
Universität Kaiserslautern, Germany
[2] `poetzsch@informatik.uni-kl.de`
Universität Kaiserslautern, Germany

Abstract. A verification strategy implementing precondition generation is presented. It automatically constructs a weak precondition for the statements of a Java subset. The strategy uses the rules of an underlying Hoare logic.

1 Introduction

The verification of object-oriented programs is a complex task. Subtyping, abstraction, frequently used method calls, and implementation encapsulation make it more difficult than verification of simple imperative programs. As fully automatic proving is in general not feasible, we are looking for techniques that automate the easier proof steps and enable the user to solve the others interactively.

A classical technique to combine user-guided proofs with automated verification are so-called strategies. Strategies[3] can be considered programs that construct parts of a proof tree by applying axioms and rules of the underlying logic. Interactive theorem provers like PVS or Isabelle/HOL use strategies that operate on higher-order logic (e.g. in PVS, a strategy is a Lisp program that attempts to construct a proof tree for the higher-order logic underlying PVS). For a Java subset, we built an interactive program prover, called Jive ([10]), that is based on a Hoare logic ([18]). Jive supports a strategy mechanism for the automated verification of program proofs.

Position In contrast to a widely accepted assumption that weakest precondition generation is *the* technique a program prover has to be built around, we argue in this paper that a Hoare logic combined with a strategy mechanism is a better design choice when it comes to the construction of provers for more complex programming languages. The main arguments are as follows:

1. wp-transformation can be implemented as a strategy, i.e. Hoare logic combined with a sufficiently powerful strategy mechanism can do everything that a "hardwired" wp-transformer can do.
2. Strategies are more flexible. In particular, wp-transformations can be mixed with interactive proof steps.

[3] Strategies are sometimes called tacticals.

3. A correct Hoare logic is easier to design than a correct wp-calculus. In particular, sharp wp-transformations for dynamic method invocations are a difficult problem (see [2, 13] for the meaning of *sharp* here).
4. The relations between method specifications in super- and subclasses can be handled in a Hoare logic, but are not part of a wp-calculus.

In the rest of the paper, we concentrate on the first argument. We present a strategy that implements a weak precondition transformer. The produced precondition is weakest for all statements except for the while-loop and the method invocation statement. Along with the presentation, the flexibility gained by strategies will be explained, i.e. the second argument will be illustrated.

A detailed discussion of the last two arguments is beyond the scope of this paper. Here, we can only sketch the ideas underlying these issues. There are some interesting results about sharp rules (see in particular [13]). However, they have to be complemented with invariant information for the heap, in order to yield the weakest precondition for Java statements. E.g. an object allocated in a prestate of a method is as well allocated in the poststate of the method. The second issue is related to behavioral subtyping. Since a wp-calculus is only concerned with statements, a prover built around such a calculus has to enforce behavioral subtyping by extra means. As shown in [17], a Hoare logic can capture this aspect directly. I.e. behavioral subtyping becomes part of the object logic and has not to be enforced by meta logical mechanisms. In some sense, this gives more flexibility. In particular, it allows to *derive* properties of superclass methods from the properties of the subclass methods.

Related Work In the limited scope of this paper, we can only discuss alternative solutions to the construction of verification tools for complex programming languages.

The ancestor of most work on program provers is the Stanford Pascal Verifier ([9]), a wp-prover for Pascal. It had a strong influence on many other program provers. Penelope is a program prover for an Ada subset (no dynamic data types, no recursive methods, no abstraction) based on weakest precondition transformation ([5]) combined with so-called simplification directives. Simplification directives are inserted into the program. They "make the proof not only easier to complete but also more apt to replay when changes are made to the program" (p. 1069). From our point of view, simplification directives are an interesting means to include information about the proof strategy into the program.

The Extended Static Checker for Java ([7]) automatically checks Java programs against specifications. It uses a transformation of Java programs to a guarded command language and performs predicate transformation based on this language ([8]). As the declared goal of ESC/Java is to achieve full automation, sacrificing completeness and specification power, the successful use of predicate transformation in this approach does not contradict our position.

The Loop tool ([1]) is a verification tool for Java programs following the same goals that we have, but with a different approach. It translates a specified Java program into a PVS theory that captures the structural and semantical aspects of the program. Whereas we do the proofs based on the programming logic within our JIVE tool[4], the Loop approach transfers all proof obligations to PVS and uses PVS strategies for proving. To simplify the proof task, a Hoare style reasoning system is used in this semantic setting ([6]). According to personal communication, the developers of the Loop tool are interested as well in developing predicate transformation based on the Hoare rules, implemented as strategies in PVS.

There are other approaches to embed the program verification task into a general theorem prover, most notably the embedding of Java into Isabelle/HOL ([14]). The typical pattern of these embeddings is to use the strategy mechanism of the underlying theorem prover to formulate program proof strategies.

Very interesting and complex applications of strategies for a Hoare logic in the sense of this paper are given in [12]. These strategies construct proofs for composed programs from proofs of the program parts. The focus of this work is on modularity.

Overview Section 2 summarizes the language and logic supported by the JIVE system and explains what a strategy is. Section 3 presents our precondition transformation strategy and its relation to the logic.

2 Verifying Java: Logic and Strategies

The JIVE system supports the verification of programs written in a sequential Java subset called JAVA-KE. The subset encompasses basic data types, interface and class types, subtyping and inheritance, dynamic binding, most statement kinds in a simplified form, and expressions without side-effects. The implementation of exception handling is not yet finished, but is already supported by the logic and strategies described here.

Logic As logic, we use a partial correctness Hoare logic. A predecessor of the logic is published in [17]. The current version of the logic together with a soundness and completeness proof will be available soon (see [18]). Formulas of the Hoare logic are sequents of the form $\mathcal{A} \triangleright \{\ \mathbf{P}\ \}\ com\ \{\ \mathbf{Q}\ \}$ where $\mathcal{A}$ is a set of assumptions, $\mathbf{P}$ and $\mathbf{Q}$ are sorted first-order formulas and *com* is either a method or a statement. The assumptions are used to handle recursive methods. As we will not need them for this presentation, they are dropped in the following; i.e. we only consider triples of the form $\{\ \mathbf{P}\ \}\ com\ \{\ \mathbf{Q}\ \}$. To understand the following presentation, we assume that the reader is familiar with Hoare logic. Here, we only explain how the object store (or heap) is modeled and how exceptions are handled.

The object store is considered a global program variable denoted by \$. The variable \$ can be used in pre- and postconditions and holds values of type *Store*. Type *Store* is an abstract data type with the following operations:

[4] The remaining first-order logic proofs are done in PVS.

$$\begin{array}{lll}
\_\langle\_ := \_\rangle & : \mathit{Store} \times \mathit{InstVar} \times \mathit{Value} & \rightarrow \mathit{Store} \\
\_\langle\_\rangle & : \mathit{Store} \times \mathit{TypeId} & \rightarrow \mathit{Store} \\
\_(\_) & : \mathit{Store} \times \mathit{InstVar} & \rightarrow \mathit{Value} \\
\mathit{alive} & : \mathit{Value} \times \mathit{Store} & \rightarrow \mathit{Bool} \\
\mathit{new} & : \mathit{Store} \times \mathit{TypeId} & \rightarrow \mathit{Value}
\end{array}$$

InstVar denotes the set of instance variables. $OS\langle IV := V\rangle$ yields the object store that is obtained from OS by updating instance variable IV with value V. $OS\langle T\rangle$ yields the object store that is obtained from OS by allocating a new object of type T. $OS(IV)$ yields the value of instance variable IV in store OS. If V is an object reference, $alive(V, OS)$ tests whether the referenced object is alive in OS. $new(OS, TID)$ yields a reference to an object of type TID that is not alive in OS. (An axiomatization of these operations is presented in [16].)

Another non-canonical feature of the logic is the treatment of exceptions. A special program variable `exc` is assumed that is different from all other program variables. It records whether an exception has occurred and if so, of what kind. Normal termination of a statement can be expressed by the conjunct $\texttt{exc} = \texttt{null}$ in the postcondition. Abrupt termination of a statement with exception E can be expressed by a conjunct $\texttt{exc} = E$ in the postcondition. In prestates, the semantics guarantees $\texttt{exc} = \texttt{null}$. In the logic, this is expressed by the so-called *exc-rule*

$$\frac{\{\ \mathbf{P} \wedge \texttt{exc} = \texttt{null}\ \}\ com\ \{\ \mathbf{Q}\ \}}{\{\ \mathbf{P}\ \}\ com\ \{\ \mathbf{Q}\ \}}$$

The logic contains axioms for the primitive statements and rules for the compound statements and methods as well as rules for adapting and reasoning about pre- and postconditions (the above rule gives an example of the latter kind).

Proofs and Strategies A proof is a tree that is constructed using the axioms and rules of the logic. In a forward proof, the instantiation of an axiom yields a primitive tree without subtrees. The application of a rule takes proof trees for the premisses of the rule and constructs a new tree node by instantiating the rule according to the premisses. In a backward proof, the proof tree is constructed starting from the root. Given a triple as proof goal, a rule application reduces this goal to subgoals. If a subgoal is an instantiation of an axiom, it is considered to be closed.

Our system allows to combine forward and backward proofs. E.g. if we have a proof tree for $\{\ \mathbf{P}\ \}\ com\ \{\ \mathbf{Q}\ \}$ and if we have a proof tree where $\{\ \mathbf{P}\ \}\ com\ \{\ \mathbf{Q}\ \}$ is a subgoal, we can glue these two proof trees together.

In the Jive system, proof construction is fully encapsulated within a so-called proof component. The component provides operations to inspect already constructed proof parts. In particular, the triples and their statements can be inspected. Proof construction is only possible by applying the logic's rules and axioms (in addition there is a glue operation). All operations first check whether the corresponding rule application is correct. If so, the rule is applied, otherwise the failures are reported (see [10]).

A strategy is a program outside the proof component that aims to construct proofs. This can only be done by using the inspection and construction operations of the proof component. Thus, a strategy can never construct incorrect proofs (assuming that the proof component is implemented correctly). In Jive, strategies are Java programs that can be dynamically loaded into the system. Having a full-fledged programming language to formulate strategies provides a great deal of flexibility. Further details of the strategy mechanism are described in [11].

3 Predicate Transformation as Interactive Strategy

In this paper we present a strategy which can verify a large part of Java-KE constructs automatically. Given a statement s and postcondition $\mathbf{Q}$, it generates a weak precondition $pwp(s, \mathbf{Q})$ and a proof for $\{\ pwp(s, \mathbf{Q})\ \}\ s\ \{\ \mathbf{Q}\ \}$. In this context, a precondition $\mathbf{P}_1$ is weaker than $\mathbf{P}_2$ iff $\mathbf{P}_2 \Rightarrow \mathbf{P}_1$. In many cases, e.g. field-read or field-write, the strategy generates the weakest (liberal) precondition (*wp*, *wlp*) [3, 4]. But e.g. in case of the `while`-loop the *wlp* rule, which is defined recursively, cannot be applied. Thus, we do not always calculate the weakest precondition but a precondition that is weak and at the same time practically useful. We call this strategy "practical weak precondition generation" (*pwp*).

Correctness of our strategy is gained automatically as the generated preconditions are immediately verified by our logic which has been proven to be sound and complete (see [18]).

3.1 The *pwp*-Strategy

The *pwp*-strategy recursively calls substrategies. There is a substrategy for each kind of statement, e.g. the substrategy *pwp-seq* works on statement sequences, *pwp-while* treats while-loops and so on. By calling these substrategies, the abstract syntax tree is recursively descended. At the leaves of the syntax tree, the precondition is generated without further calls to substrategies. A proof tree with a single node, represented by the triple $\{\ pwp(s, \mathbf{Q})\ \}\ s\ \{\ \mathbf{Q}\ \}$, is created by applying the appropriate axiom of the logic in forward direction, which at the same time proves the generated precondition. This precondition is then used by the calling substrategy to generate the precondition for its statement. The calling substrategy also extends the generated proof tree. It adds a new root node or joins two proof trees into one by applying a rule of the programming logic in forward direction. In this way, the proof tree is built bottom-up.

Ideally, the strategy automatically builds the complete proof tree for a code sequence and glues it to the already existing proof tree where the initial strategy call started from. But, usually, user interaction is required for some of the language constructs. Details are given below when explaining the substrategies.

3.2 *pwp*-Substrategies

This section lists some of the *pwp*-substrategies. There is one substrategy for each statement. Here, we only explain the substrategies for some selected statements.

pwp-cast: This substrategy deals with one kind of terminals of the abstract syntax tree: the cast-assignments. These are assignments which at the same time perform a cast. To keep the programs to be verified more readable, the JIVE system allows to omit the cast "(T)" if the types of both sides of the assignment are identical. In these cases the cast is implicitly added by the system.

The precondition is generated mainly by performing the required syntactical substitutions, without any further invocations of substrategies. It is defined as follows, where e is a side-effect-free expression, $\preceq$ denotes the subtype relation and typeof() yields the static type of its argument:

$$\begin{aligned} pwp(\text{``}x = (\mathrm{T})e;\text{''}, \mathbf{P}) =_{def} & \ (\mathrm{typeof}(e) \preceq \mathrm{T} \ \wedge \ \mathbf{P}[e/x]) \ \vee \\ & \ (\mathrm{typeof}(e) \not\preceq \mathrm{T} \ \wedge \ \mathbf{P}[\$\langle \texttt{CastExc} \rangle / \$, \mathit{new}(\$, \texttt{CastExc})/\texttt{exc}]) \end{aligned}$$

The *pwp-cast* substrategy generates a proof tree consisting of only one node that is built from the generated precondition, the cast-assignment and the given postcondition, by instantiating the *cast-axiom* of the logic.

cast-axiom:

$$\left\{ \begin{array}{l} (\mathrm{typeof}(e) \preceq \mathrm{T} \ \wedge \ \mathbf{P}[e/x]) \vee \\ (\mathrm{typeof}(e) \not\preceq \mathrm{T} \ \wedge \ \mathbf{P}[\$\langle \texttt{CastExc} \rangle / \$, \mathit{new}(\$, \texttt{CastExc})/\texttt{exc}]) \end{array} \right\} \ x = (\mathrm{T})e; \ \{ \ \mathbf{P} \ \}$$

This immediately proves the precondition, which is trivial in this case, and the instantiated proof tree is returned to the substrategy that called *pwp-cast*.

The field-read, field-write, and object creation statements are treated in a similar way.

The substrategies that are presented in the following each take one or more proof trees and combine them into a new proof tree by placing them below a new common root node. In case of a single proof tree, this tree is simply extended by placing a new root node on top of the current one.

pwp-seq: When *pwp-seq* is called on a sequence $s_1 \ s_2$, it first calls the appropriate substrategy on s_2. This substrategy returns a proof tree t_2 with root triple $\{ \ pwp(s_2, \mathbf{Q}) \ \} \ s_2 \ \{ \ \mathbf{Q} \ \}$. *pwp-seq* extracts the precondition from the tree and uses it in calling the appropriate substrategy on the first child s_1 of the code sequence. That substrategy returns a proof tree t_1. *pwp-seq* then applies the *seq-rule* of the logic in forward direction[5] to t_1 and t_2, which generates a new common root node to the two trees. The precondition of t_1 becomes the overall

[5] For an explanation of using the rules of the logic in forward and backward direction see [10].

precondition of the new proof tree, and the postcondition is the one the *pwp-seq* substrategy was called with. Altogether, the precondition

$$pwp(\text{"}s_1\,s_2\text{"}, \mathbf{Q}) =_{def} pwp(\text{"}s_1\text{"}, (\texttt{exc} \neq \texttt{null} \ \wedge \ \mathbf{Q}) \ \vee \ (\texttt{exc} = \texttt{null} \ \wedge \ pwp(\text{"}s_2\text{"}, \mathbf{Q})))$$

is generated. *pwp-seq* uses the *seq-rule* of the logic:

seq-rule:

$$\frac{\{\ \mathbf{P}\ \}\ s_1\ \{\ (\texttt{exc} \neq \texttt{null} \ \wedge \ \mathbf{Q}) \ \vee \ (\texttt{exc} = \texttt{null} \ \wedge \ \mathbf{R})\ \} \qquad \{\ \mathbf{R}\ \}\ s_2\ \{\ \mathbf{Q}\ \}}{\{\ \mathbf{P}\ \}\ s_1\,s_2\ \{\ \mathbf{Q}\ \}}$$

The correctness of the generated precondition directly follows from the *seq-rule*.

pwp-catch: This substrategy treats `try-catch`-clauses. It calls the appropriate substrategy on the code sequence s_1 of the `catch`-statement and the overall postcondition and uses the generated precondition to form a complex expression. This expression is passed as argument to the substrategy that is then called in order to prove the code sequence s_0 of the `try`-statement. The two resulting proof trees are combined into one by applying the logic's *catch-rule* in forward direction. The substrategy *pwp-catch* generates the following precondition:

$$\begin{aligned} pwp(\text{"}\texttt{try}\{s_0\}\texttt{catch}(\mathrm{T}\,e)\{s_1\}\text{"}, \mathbf{Q}) =_{def} \ & pwp(s_0, ((\texttt{exc} = \texttt{null} \vee \mathrm{typeof}(\texttt{exc}) \npreceq \mathrm{T}) \wedge \mathbf{Q}) \\ & \vee (\texttt{exc} \neq \texttt{null} \ \wedge \ \mathrm{typeof}(\texttt{exc}) \preceq \mathrm{T} \ \wedge \ pwp(s_1, \mathbf{Q})[\texttt{null}/\texttt{exc}, \texttt{exc}/e])) \end{aligned}$$

We show the steps the strategy performs in the proof component of Jive. Apart from two implications that are not dealt with in Jive, these steps are implicitly known to be correct because they are performed by axioms and rules of our logic. Thus, the following can be regarded as a proof of correctness of the generated precondition. This proof mainly uses the *catch-rule*.

catch-rule:

$$\frac{\{\ \mathbf{P}\ \}\ s_0 \left\{ \begin{array}{l} ((\texttt{exc} = \texttt{null} \ \vee \ \mathrm{typeof}(\texttt{exc}) \npreceq \mathrm{T}) \ \wedge \ \mathbf{Q}) \\ \vee\ (\texttt{exc} \neq \texttt{null} \ \wedge \ \mathrm{typeof}(\texttt{exc}) \preceq \mathrm{T} \ \wedge \ \mathbf{R}) \end{array} \right\} \qquad \{\ \mathbf{R}[e/\texttt{exc}]\ \}\ s_1\ \{\ \mathbf{Q}\ \}}{\{\ \mathbf{P}\ \}\ \texttt{try}\{s_0\}\texttt{catch}(\mathrm{T}\ e)\{s_1\}\ \{\ \mathbf{Q}\ \}}$$

First, we determine the shape of the formula $\mathbf{R}$ used in the *catch-rule*:

$$
\begin{array}{ll}
\{\ pwp(s_1,\mathbf{Q})\ \}\ s_1\ \{\ \mathbf{Q}\ \} & \\
*\quad pwp(s_1,\mathbf{Q})\ \wedge\ \texttt{exc}=\texttt{null} \Rightarrow pwp(s_1,\mathbf{Q}) & \\
\hline
\{\ pwp(s_1,\mathbf{Q})\ \wedge\ \texttt{exc}=\texttt{null}\ \}\ s_1\ \{\ \mathbf{Q}\ \} & \text{strengthening} \\
*\quad pwp(s_1,\mathbf{Q})[\texttt{null}/\texttt{exc}]\ \wedge\ \texttt{exc}=\texttt{null} \Rightarrow pwp(s_1,\mathbf{Q})\ \wedge\ \texttt{exc}=\texttt{null} & \\
\hline
\{\ pwp(s_1,\mathbf{Q})[\texttt{null}/\texttt{exc}]\ \wedge\ \texttt{exc}=\texttt{null}\ \}\ s_1\ \{\ \mathbf{Q}\ \} & \text{strengthening} \\
\hline
\{\ pwp(s_1,\mathbf{Q})[\texttt{null}/\texttt{exc}]\ \}\ s_1\ \{\ \mathbf{Q}\ \} & \text{exc-rule} \\
\hline
\{\ \underbrace{pwp(s_1,\mathbf{Q})[\texttt{null}/\texttt{exc}][\texttt{exc}/e]}_{\equiv\mathbf{R}}[e/\texttt{exc}]\ \}\ s_1\ \{\ \mathbf{Q}\ \} &
\end{array}
$$

The proof obligations marked with $*$ are valid for all applications of the strategy and can be discharged automatically. The next step is to apply the *catch-rule* to this result:

$$
\begin{array}{ll}
\left\{\begin{array}{l} pwp(s_0,((\texttt{exc}=\texttt{null}\ \vee \\ \text{typeof}(\texttt{exc}) \not\preceq \text{T})\ \wedge\ \mathbf{Q})\ \vee \\ (\texttt{exc}\neq\texttt{null}\ \wedge \\ \text{typeof}(\texttt{exc}) \preceq \text{T}\ \wedge \\ pwp(s_1,\mathbf{Q})[\texttt{null}/\texttt{exc},\texttt{exc}/e])) \end{array}\right\}
s_0
\left\{\begin{array}{l} ((\texttt{exc}=\texttt{null}\ \vee \\ \text{typeof}(\texttt{exc}) \not\preceq \text{T})\ \wedge\ \mathbf{Q})\ \vee \\ (\texttt{exc}\neq\texttt{null}\ \wedge \\ \text{typeof}(\texttt{exc}) \preceq \text{T}\ \wedge \\ pwp(s_1,\mathbf{Q})[\texttt{null}/\texttt{exc},\texttt{exc}/e]) \end{array}\right\} & \\
\{\ pwp(s_1,\mathbf{Q})[\texttt{null}/\texttt{exc},\texttt{exc}/e][e/\texttt{exc}]\ \}\ s_1\ \{\ \mathbf{Q}\ \} & \\
\hline
\{\ pwp(\text{“}\texttt{try}\{s_0\}\texttt{catch}(\text{T}\ e)\{s_1\}\text{”},\mathbf{Q})\ \}\ \texttt{try}\{s_0\}\texttt{catch}(\text{T}\ e)\{s_1\}\ \{\ \mathbf{Q}\ \} & \text{catch-rule}
\end{array}
$$

□

This immediately yields the correctness of the generated precondition and the proof tree resulting from the application of *pwp-catch.*

pwp-while: This substrategy relies on a loop invariant **I**. First, it asks the user for this invariant such that $\mathbf{I}\wedge e \Rightarrow pwp(\text{“}s\text{”},\mathbf{I})$ and $\mathbf{I}\ \wedge\ (\texttt{exc}\neq\texttt{null}\ \vee\ \neg e) \Rightarrow \mathbf{Q}$ hold, where **Q** is the given postcondition, e is the loop condition, an expression without side-effects, and s is the loop body. (One could also imagine that the user annotates some loops of the program with loop invariants before starting the verification.) The substrategy calls the appropriate substrategy for the loop body. Then, it creates a new proof tree by applying the *while-rule* of the logic in forward direction.

The precondition generation is based on the transformer proposed by Gries [4, p. 144]. This is not the weakest liberal precondition but implies it.

$$pwp(\text{“}while(e)\{s\}\text{”},\mathbf{Q}) =_{def} \mathbf{I}$$

This is the *while-rule* of the logic:

while-rule:

$$
\frac{\{\ e\wedge\mathbf{I}\ \}\ s\ \{\ \mathbf{I}\ \}}{\{\ \mathbf{I}\ \}\ \texttt{while}(e)\{s\}\ \{\ (\texttt{exc}\neq\texttt{null}\vee\neg e)\wedge\mathbf{I}\ \}}
$$

We show the validity of the generated precondition:

$$\begin{array}{ll}
\{\ pwp(``s", \mathbf{I})\ \}\ s\ \{\ \mathbf{I}\ \} & \\
*\quad \mathbf{I} \wedge e \Rightarrow pwp(``s", \mathbf{I}) & \\
\hline
\{\ e \wedge \mathbf{I}\ \}\ s\ \{\ \mathbf{I}\ \} & \text{strengthening} \\
\hline
\{\ \mathbf{I}\ \}\ while(e)\{s\}\ \{\ (\texttt{exc} \neq \texttt{null} \vee \neg e) \wedge \mathbf{I}\ \} & \text{while-rule} \\
*\quad \mathbf{I}\ \wedge\ (\texttt{exc} \neq \texttt{null}\ \vee\ \neg e) \Rightarrow \mathbf{Q} & \\
\hline
\{\ \mathbf{I}\ \}\ while(e)\{s\}\ \{\ \mathbf{Q}\ \} & \text{weakening}
\end{array}$$

$\square$

The user has to verify the implications marked with $*$ (in Jive, PVS is used for this purpose).

pwp-invocation is derived from Gries [4, Chap. 12]. The substrategy uses the specification $\{\ \mathbf{P}\ \}$ T:m(`par`) $\{\ \mathbf{R}\ \}$ of the invoked method m where T represents the static type of the target expression. This triple denotes the specification of the virtual method T:m expressing the properties of all methods that can possibly be executed by an invocation $y.m(e)$. (For a detailed discussion of virtual methods and Hoare logic see [16].)

The *pwp-invocation* substrategy basically takes the precondition of the method specification, with some slight modifications, as precondition of the method invocation site. To be able to do this, it has to guarantee that the postcondition of the method specification implies the specification at the invocation site. This implication must hold for a suitable set of results and object stores. It is safe to assume the implication for all results and object stores, but this precondition can usually be weakened by constraining their choice. Details of such constraints are given below. If, however, the method is invoked on an object that equals `null`, the precondition can directly be derived from the postcondition, without regarding any method specifications.

$$\begin{array}{c}
pwp(``x = y.m(e);", \mathbf{Q}) =_{def} (y \neq \texttt{null} \wedge \mathbf{P}[y/\texttt{this}, e/\texttt{par}] \wedge \\
(\forall E, H : \rho(y, e, \$, H, E) \wedge \mathbf{R}[E/\texttt{res}, H/\$] \Rightarrow \mathbf{Q}[E/x, H/\$])) \\
\vee(y = \texttt{null} \wedge \mathbf{Q}[\$\langle\texttt{NullPExc}\rangle/\$, new(\$, \texttt{NullPExc})/\texttt{exc}])
\end{array}$$

The actual parameter expression e is side-effect-free. H and E must be fresh variables that do not appear in $\mathbf{Q}$ nor $\mathbf{R}$. The term $\rho(y, e, \$, H, E)$ represents some general constraints on the objects on which the method can be invoked, on the arguments, on the current object store, on the possible results and on the possible object stores after execution. A good ρ makes the precondition weaker. A simple solution is to take ρ to be true everywhere. More ambitious solutions constrain object stores of poststates in such a way that they can be obtained by a finite number of modifications of the object store in the prestate. This would especially express that all objects that are allocated in the prestate are as well allocated in the object store of the poststate because object destruction cannot be performed in JAVA-KE. In addition to this, a method body can only alter those objects that can be reached from y or e. For further examples of constraints see [15].

The *pwp-invocation* substrategy extends the proof tree of the virtual method by applying the rules for the invocation statement in forward direction.

To prove the correctness of the precondition generated by *pwp-invocation*, we need the *invoc-rule* which relates the invocation site of a method to its specification (in this rule, T denotes the static type of the target expression y), and the *invoc-exc* rule which deals with method invocations on a `null` reference:

invoc-rule:

$$\frac{\{\ \mathbf{P}\ \}\ \mathrm{T{:}m}(\mathtt{par})\ \{\ \mathbf{Q}\ \}}{\{\ y \neq \mathtt{null}\ \wedge\ \mathbf{P}[y/\mathtt{this}, e/\mathtt{par}]\ \}\ x = y.m(e);\ \{\ \mathbf{Q}[x/\mathtt{res}]\ \}}$$

invoc-exc:

$$\{\ y = \mathtt{null}\ \wedge\ \mathbf{Q}[\$\langle\mathtt{NullPExc}\rangle/\$, \mathit{new}(\$, \mathtt{NullPExc})/\mathtt{exc}]\ \}\ x = y.m(e);\ \{\ \mathbf{Q}\ \}$$

The proof uses several other rules of our logic which we cannot present within the scope of this paper. These are the *invoc-var* rule which states that method invocations do not modify local variables except the one the return value is assigned to, the *swis-rule* which states (among others) that a conjunct $\mathbf{U}$ that does not depend on program variables can simultaneously be added to the pre- and postcondition of a Hoare triple, and the *disjunct-rule*. These rules can be found in [18].

We prove $\{\ pwp(\text{“}x = y.m(e);\text{”}, \mathbf{Q})\ \}\ x = y.m(e);\ \{\ \mathbf{Q}\ \}$ under the assumption that $\{\ \mathbf{P}\ \}\ \mathrm{T{:}m}(\mathtt{par})\ \{\ \mathbf{R}\ \}$ holds. In the proof, we assume $\rho(y, e, \$, H, E) \equiv$ true. $\overline{\mathbf{Q}}$ is derived from $\mathbf{Q}$ by replacing all program variables except x with fresh logical variables.

$$\dfrac{\dfrac{\dfrac{\{\ \mathbf{P}\ \}\ \mathrm{T{:}m}(\mathtt{par})\ \{\ \mathbf{R}\ \}}{\{\ y \neq \mathtt{null} \wedge \mathbf{P}[y/\mathtt{this}, e/\mathtt{par}]\ \}\ x = y.m(e);\ \{\ \mathbf{R}[x/\mathtt{res}]\ \}}\ \text{invoc-rule}}{\left\{\begin{array}{l} y \neq \mathtt{null}\ \wedge \\ \mathbf{P}[y/\mathtt{this}, e/\mathtt{par}]\ \wedge \\ (\forall E, H : \mathbf{R}[E/\mathtt{res}, H/\$] \\ \Rightarrow \overline{\mathbf{Q}}[E/x, H/\$]) \end{array}\right\}\ x = y.m(e);\ \left\{\begin{array}{l} \mathbf{R}[x/\mathtt{res}]\ \wedge \\ (\forall E, H : \mathbf{R}[E/\mathtt{res}, H/\$] \\ \Rightarrow \overline{\mathbf{Q}}[E/x, H/\$]) \end{array}\right\}}\ \text{swis-rule} \qquad (\forall E, H : \mathbf{R}[E/\mathtt{res}, H/\$] \Rightarrow \overline{\mathbf{Q}}[E/x, H/\$]) \Rightarrow (\mathbf{R}[x/\mathtt{res}, \$/\$] \Rightarrow \overline{\mathbf{Q}}[x/x, \$/\$])}{\left\{\begin{array}{l} y \neq \mathtt{null}\ \wedge \\ \mathbf{P}[y/\mathtt{this}, e/\mathtt{par}]\ \wedge \\ (\forall E, H : \mathbf{R}[E/\mathtt{res}, H/\$] \\ \Rightarrow \overline{\mathbf{Q}}[E/x, H/\$]) \end{array}\right\}\ x = y.m(e);\ \left\{\begin{array}{l} \mathbf{R}[x/\mathtt{res}]\ \wedge \\ (\mathbf{R}[x/\mathtt{res}, \$/\$] \\ \Rightarrow \overline{\mathbf{Q}}[x/x, \$/\$]) \end{array}\right\}}\ \text{weakening}$$

$$
\begin{array}{ll}
\left\{\begin{array}{l} y \neq \mathtt{null} \wedge \\ \mathbf{P}[y/\mathtt{this}, e/\mathtt{par}] \wedge \\ (\forall E, H : \mathbf{R}[E/\mathtt{res}, H/\$] \\ \Rightarrow \overline{\mathbf{Q}}[E/x, H/\$]) \end{array}\right\} \; x = y.m(e); \; \left\{\begin{array}{l} \mathbf{R}[x/\mathtt{res}] \wedge \\ (\mathbf{R}[x/\mathtt{res}, \$/\$] \\ \Rightarrow \overline{\mathbf{Q}}[x/x, \$/\$]) \end{array}\right\} & \\
\mathbf{R}[x/\mathtt{res}] \;\wedge\; (\mathbf{R}[x/\mathtt{res}, \$/\$] \Rightarrow \overline{\mathbf{Q}}[x/x, \$/\$]) \Rightarrow \overline{\mathbf{Q}} & \\
\hline
\left\{\begin{array}{l} y \neq \mathtt{null} \;\wedge\; \mathbf{P}[y/\mathtt{this}, e/\mathtt{par}] \wedge \\ (\forall E, H : \mathbf{R}[E/\mathtt{res}, H/\$] \Rightarrow \overline{\mathbf{Q}}[E/x, H/\$]) \end{array}\right\} \; x = y.m(e); \; \{\, \overline{\mathbf{Q}} \,\} & \text{weakening} \\
\hline
\left\{\begin{array}{l} y \neq \mathtt{null} \;\wedge\; \mathbf{P}[y/\mathtt{this}, e/\mathtt{par}] \wedge \\ (\forall E, H : \mathbf{R}[E/\mathtt{res}, H/\$] \Rightarrow \mathbf{Q}[E/x, H/\$]) \end{array}\right\} \; x = y.m(e); \; \{\, \mathbf{Q} \,\} & \text{invoc-var} \\
\{\, y = \mathtt{null} \wedge \mathbf{Q}[\$\langle\mathtt{NullPExc}\rangle/\$, \mathit{new}(\$, \mathtt{NullPExc})/\mathtt{exc}] \,\} \; x = y.m(e); \; \{\, \mathbf{Q} \,\} & \text{invoc-exc} \\
\hline
\{\, \mathit{pwp}(\text{“}x = y.m(e);\text{”}, \mathbf{Q}) \,\} \;\; x = y.m(e); \;\; \{\, \mathbf{Q} \;\vee\; \mathbf{Q} \,\} & \text{disjunct-r.} \\
\mathbf{Q} \;\vee\; \mathbf{Q} \Rightarrow \mathbf{Q} & \\
\hline
\{\, \mathit{pwp}(\text{“}x = y.m(e);\text{”}, \mathbf{Q}) \,\} \;\; x = y.m(e); \;\; \{\, \mathbf{Q} \,\} & \text{weakening}
\end{array}
$$

4 Conclusions

In this paper we presented a verification strategy which implements precondition generation. It is based on a Hoare logic for a Java subset. The generated preconditions are shown to be correct w.r.t. the Hoare logic. Using a Hoare logic and supplementing it with strategies provides more flexibility than a fixed wp-calculus. Extensions of the described strategy can perform intermediate simplification steps based on the logic. Different strategies can be used to verify different aspects of a program. They can also work on the same proof side-by-side. This paper presents a substrategy that assumes annotated methods. However, one can also imagine a strategy that tries to derive method annotations from implementations. Altogether, this paper points out several advantages of strategies compared to a *wp*-calculus, and suggests starting points for the development of more refined strategies.

Acknowledgements. We thank the anonymous reviewers and Peter Müller for their careful considerations and constructive remarks.

References

1. J. van den Berg and B. Jacobs. The LOOP compiler for Java and JML. In T. Margaria and W. Yi, editors, *TACAS01, Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031 of *Lecture Notes in Computer Science*, pages 299–312. Springer Verlag, 2001.
2. A. Bijlsma, P. A. Matthews, and J. G. Wiltink. A sharp proof rule for procedures in wp semantics. *Acta Informatica*, 26:409–419, 1989.
3. E. W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

4. David Gries. *The Science of Programming*. Springer-Verlag, New York, 1981.
5. David Guaspari, Carla Marceau, and Wolfgang Polak. Formal verification of Ada programs. *IEEE Transactions on Software Engineering*, 16(9):1058–1075, Sept. 1990.
6. M. Huisman. *Reasoning About Java Programs in Higher Order Logic Using PVS and Isabelle*. PhD thesis, University of Nijmegen, 2000.
7. K. Leino, G. Nelson, and J. Saxe. *ESC/Java user's manual*. Compaq Systems Research Center, Palo Alto, CA, October 2000. #2000-002. Available from `http://gatekeeper.dec.com/pub/DEC/SRC/technical-notes/SRC-2000-002.html`.
8. K. R. M. Leino, J. B. Saxe, and R. Stata. Checking Java programs via guarded commands. In B. Jacobs, G. T. Leavens, P. Müller, and A. Poetzsch-Heffter, editors, *Formal Techniques for Java Programs*. Technical Report 251, Fernuniversität Hagen, 1999. Available from `http://www.informatik.fernuni-hagen.de/pi5/publications.html`.
9. D. Luckham et al. *Stanford PASCAL Verifier - User Manual*. Stanford University, Departement of Computer Science, Stanford, California, 1979. STAN-CS-79-731; also: Stanford Verification Group, Rep. No. 11, Edition 1, 2nd Printing March 1980.
10. J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *TACAS00, Tools and Algorithms for the Construction and Analysis of Systems*, volume 276 of *Lecture Notes in Computer Science*, pages 63–77, 2000.
11. J. Meyer and A. Poetzsch-Heffter. Strategies for the Verification of Object-oriented Programs. In G. Schellhorn and W. Reif, editors, *FM-Tools 2000: The 4th Workshop on Tools for System Design and Verification, Reisensburg, Germany*, Ulmer Informatik Berichte Nr. 2000-07. Universität Ulm, Fakultät für Informatik, 2000.
12. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.
13. David A. Naumann. Calculating sharp adaptation rules. *Information Processing Letters*, 77(2-4):201–208, 2001.
14. David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001. `http://www4.in.tum.de/~oheimb/diss/`.
15. A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997.
16. A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In D. Gries and W. De Roever, editors, *Programming Concepts and Methods (PROCOMET)*, 1998.
17. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In D. Swierstra, editor, *ESOP '99*, volume 1576 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.
18. A. Poetzsch-Heffter and N. Rauch. A Hoare Logic for a Java Subset and its Proof of Soundness and Completeness. Internal Report 324/03, Universität Kaiserslautern, Germany, 2003.