Strategies for the Verification of Object-oriented Programs

J. Meyer and A. Poetzsch-Heffter

May 18, 2000

1 Introduction

Verification of typical object-oriented programs is more complex than verification of imperative programs with non-recursive procedures. It has to cope with dynamic binding of methods, recursive method invocations, subtyping, abstract types, and aliasing through object references. The use of these features and the combination of data and procedures into classes lead to a different programming style which also influences verification techniques. The goal of our research is to overcome the resulting challenges by interactive verification techniques that can be formulated as proof strategies similar to the classical approach for tactical theorem proving.

In this extended abstract, we analyze typical verification steps within proofs of object-oriented programs. We focus on the differences to verification of imperative programs with non-recursive procedures. In particular, we investigate which information about programs is needed in order to support the verification steps by strategies in a tactical theorem prover. The rest of this abstract is structured as follows. Section 2 sketches the used verification framework. In particular, it describes a simple overall technique to prove that an OO-program satisfies its specification. Section 3 analyses the steps of this simple proof technique w.r.t. formalizing them as strategies. Section 4 concludes by sketching the realization of such strategies within tactical theorem provers.

2 Verification of Object-oriented Programs

The results of the following investigation hold for most existing object-oriented programming languages. However, to have a concrete setting, we consider the kernel of Java as a representative for this language class (cf. [PHM99]). Similarly, we refer here only to a specific programming logic, namely a Hoare-style partial correctness logic, although the results of the investigation apply as well to more powerful programming logics (e.g. dynamic logic or temporal logic).

Program verification can be done under different perspectives. The classical perspective assumes a complete program with exactly one main procedure. The goal is to verify that the main procedure has a certain property. This perspective does not reflect the reuse of program modules, as it underlies modular programming in general and object-oriented programming in particular. Consequently, we assume a perspective where the goal is to verify interface specifications of a set[1] of types ST. As the implementation of these types may use other, already verified types STU, we assume that the used types are equipped with interface specifications that can be applied during verification of ST. This means that we have to relate types and specifications in such a way that the theorem proving environment can access *the* specification of a type. In particular, the verification of ST can only use properties about STU that are formulated in STU's interface specifications.

A user-defined type in Java is either declared by a class or a so-called interface. To keep things simple here, we assume that the specification of a type T consists of the specifications of T's methods (cf. [PH97] for more elaborated specification constructs). The specification of a method consists of a set of pre-/postpairs. From a theoretical point of view, it would be sufficient to consider only one pre-/postpair. In practical situations, this would lead to large pre- and

[1] As types can and usually do recursively depend on each other, we have to deal with sets of types instead of single types.

postconditions. A more structured approach using several pairs turns out to be easier to handle where each pair deals with a different aspect of the method behavior: E.g. one pair to describe the returned value of the method; one pair to specify the modifications to the object store; one pair to capture the invariant properties to the method etc.

Proof Task. Program verification techniques can be used for different purposes; e.g. to prove that a program never throws a NullPointerException or to derive the needed properties of an auxiliary method. Here, we only consider the following standard proof task: Given a set of specified and implemented types ST that use a set of specified types STU; prove that the implementation of ST satisfies its specification assuming that the types STU behave as specified. Before we present a simple technique to structure the proof task we have to summarize an important aspect of OO-program verification.

Dynamic Binding and Virtual Methods. In OO-programs, method invocation are dynamically bound, i.e. it is statically in general not known which method implementation is executed at an invocation site. Even worse for verification, several different implementations can be invoked from an invocation site during the execution of a program. There are essentially two techniques to handle dynamic method binding in a programming logic. If we know the whole program, we can figure out which methods are possibly called at the site using type and subtype information. Then, we show that the needed property at the site is guaranteed by all method implementations possibly bound to the site. The disadvantages of this technique are that we usually cannot access the implementation of the whole program and that we have to show behavioral subtyping properties (cf. [LW94]) for each invocation site again.

That is why we use so-called *virtual methods* to capture the behavior of all subtype methods; the properties of the virtual method m of type T are then used to verify an invocation site of m with *static* type T. If T is an interface type and m is declared in T, T:m denotes the virtual method that abstracts the common behavior of all implementations of m in subtypes of T. To put it the other way round, all subtype methods have to satisfy the specification of T:m. If T is a class type, T:m is again used to denote the virtual method capturing the implementation of m in T and in all subtypes of T. By T@m we denote the implementation of m in T. Notice that this implementation can be overridden in subtypes of T so that the properties of T:m are in general more abstract than those of T@m (for details of this technique we refer to[PHM99]). The specification of a method refers to the properties of T:m.

A Simple Proof Technique. In general, the verification of a set of types ST can be divided into the following steps where the specification of the used types STU serve as axioms. In the first step, the properties of virtual methods are reduced to the required properties of given implementations and and to proof obligations for types in ST that are subtypes of types in STU (for brevity, the latter aspect will be neglected in the following). In a second step, the method implementations are verified w.r.t. the required properties. To capture recursion, the properties of the virtual methods may be assumed in this step. In a third step, these assumptions have to be eliminated.

3 Automating Program Proofs for OO-Programs

In this section, we explain some of the verification steps outlined above in more detail and discuss further aspects relevant to the verification of OO-programs. The described proof techniques serve as examples for strategies which are needed for the proof of OO-programs. Program proofs are constructed using the rules and axioms of the underlying Hoare logic. Rules and axioms are provided as operations (cf. tactics in the LCF approach [GMW79]) with forward and backward direction use. Using an operation leads to an extension at the root (forward proof) or at a leaf (backward proof). Operations can be combined to form strategies (cf. tacticals in LCF).

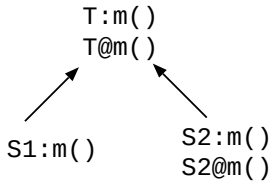
3.1 Verifying behavioral Subtyping

As we consider object-oriented programs, we are directly confronted with subtyping. This means that specified properties of a virtual method T:m have to be shown for the methods S:m in subtypes S of T. As an example we consider the following program fragment:

```
class T {
 int m() { ... }
}

class S1 extends T {
}

class S2 extends T {
 int m() { ... }
}
```



The program consists of three classes where S1 and S2 are subtypes of class T. T has a virtual method m, therefore subtypes S1 and S2 have virtual methods S1:m resp. S2:m. T:m is implemented in class T. Class S1 inherits the implementation T@m for S1:m and S2 reimplements T@m with S2@m. Suppose now you want to show the program proof obligation $\vdash \{\ \mathbf{P}\ \}$ T:m() $\{\ \mathbf{Q}\ \}$[2], which arises from the program specification. The proof of the given goal can be divided into two phases. 1) Take proof obligations for virtual methods in supertypes back to virtual methods in subtypes. 2) Show that the implementations of virtual methods in subtypes have the properties of the methods in supertypes. In the following we sketch the first phase, which is mainly performed by the operations of the following rules (where τ denotes *typeof(this)*):

subtype-rule:

$$\frac{S \preceq T \qquad \mathcal{A} \vdash \{\ \tau \preceq S \wedge \mathbf{P}\ \}\ \text{S:m()}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \tau \preceq S \wedge \mathbf{P}\ \}\ \text{T:m()}\ \{\ \mathbf{Q}\ \}}$$

class-rule:

$$\frac{\mathcal{A} \vdash \{\ \tau = T \wedge \mathbf{P}\ \} \mathit{impl}(T,m())\{\ \mathbf{Q}\ \} \qquad \mathcal{A} \vdash \{\ \tau \prec T \wedge \mathbf{P}\ \}\ \text{T:m()}\ \{\ \mathbf{Q}\ \}}{\mathcal{A} \vdash \{\ \tau \preceq T \wedge \mathbf{P}\ \}\ \text{T:m()}\ \{\ \mathbf{Q}\ \}}$$

disjunct-rule:

$$\frac{\mathcal{A} \vdash \{\ \mathbf{P}_1\ \}\ \text{comp}\ \{\ \mathbf{Q}_1\ \} \qquad \mathcal{A} \vdash \{\ \mathbf{P}_2\ \}\ \text{comp}\ \{\ \mathbf{Q}_2\ \}}{\mathcal{A} \vdash \{\ \mathbf{P}_1 \vee \mathbf{P}_2\ \}\ \text{comp}\ \{\ \mathbf{Q}_1 \vee \mathbf{Q}_2\ \}}$$

The following algorithm builds a proof tree for the behavioral subtyping proof in backward direction. It uses the subtype-, class-, strength-, and disjunct-rule of the underlying Hoare logic until only program proof obligations for method implementations are left:

Input: A proof goal g
Result: The algorithm constructs a proof tree with g as root and open leafs (slots) for the proofs of method implementations.

$\mathbb{G} \leftarrow \{g\}$
while $\mathbb{G} \neq \{\ \}$ **do**
  select any g from $\mathbb{G}$
  if g matches $\mathcal{X} \vdash \{\ \tau \preceq T \wedge \mathbf{A}\ \}$ T@m() $\{\ \mathbf{B}\ \}$ **then**
    /* show specification for all subtypes of T and T */
    use class-rule backward for g
    $\mathbb{G} \leftarrow \mathbb{G} \cup \{\mathcal{X} \vdash \{\ \tau = T \wedge \mathbf{A}\ \}$ T@m() $\{\ \mathbf{B}\ \}\}$
    $\mathbb{G} \leftarrow \mathbb{G} \cup \{\mathcal{X} \vdash \{\ \tau \prec T \wedge \mathbf{A}\ \}$ T:m() $\{\ \mathbf{B}\ \}\}$
  else if g matches $\mathcal{X} \vdash \{\ \tau \prec T \wedge \mathbf{A}\ \}$ T:m() $\{\ \mathbf{B}\ \}$ **then**
    /* show specification for all subtypes of T */
    if T has no subtypes **then**
      /* as the precondition evaluates to *false*, g can be derived from the false-axiom $\vdash \{$ FALSE $\}$ comp $\{$ FALSE $\}$ (not shown here) */
    else
      let $S_{1 \le i \le n}$ be all direct subtypes of T /*thus $\tau \not\preceq T \Rightarrow \bigvee_{i=1\ldots n} \tau \preceq S_i$*/
      use strength-rule backward to reduce g to $\mathcal{X} \vdash \{\ \bigvee_{i=1\ldots n} \tau \preceq S_i \wedge \mathbf{A}\ \}$ T:m() $\{\ \mathbf{B}\ \}$
      for $j = n \ldots 2$ **do**
        use disjunct-rule backward on g
        /* split $\bigvee_{i=1\ldots j} \tau \preceq S_i$ up to $\bigvee_{i=1\ldots j-1} \tau \preceq S_i \vee \tau \preceq S_j$*/
        g$\leftarrow \mathcal{X} \vdash \{\ \bigvee_{i=1\ldots j-1} \tau \preceq S_i \wedge \mathbf{A}\ \}$ T:m() $\{\ \mathbf{B}\ \}$
        $\mathbb{G} \leftarrow \mathbb{G} \cup \{\mathcal{X} \vdash \{\ \tau \preceq S_j \wedge \mathbf{A}\ \}$ T:m() $\{\ \mathbf{B}\ \}\}$
      end for
      $\mathbb{G} \leftarrow \mathbb{G} \cup \{g\}$

[2] We use the sequent notation of Hoare logic triples where an optional set of method assumptions is noted before the turnstyle, **P** and **Q** are pre,- resp. postcondition and the middle part denotes a program part.

end if
else if g matches $\mathcal{X} \vdash \{\ \tau \preceq \mathrm{S} \wedge \mathbf{A}\ \}$ T:m() $\{\ \mathbf{B}\ \}$ **then**
/* show specification for subtype S (which is direct subtype of T) */
use subtype-rule backward on g
$\mathbb{G} \leftarrow \mathbb{G} \cup \{\mathcal{X} \vdash \{\ \tau = \mathrm{S} \wedge \mathbf{A}\ \}$ S:m() $\{\ \mathbf{B}\ \}\}$
else if g matches $\_ \vdash \{\ \tau = \mathrm{T} \wedge \_\ \}$ T@m() $\{\ \_\ \}$ **then**
/* implementations are left open in proof tree and not considered further. A strategy to prove method implementations could be attached here */
...
end if
$\mathbb{G} \leftarrow \mathbb{G} \setminus \{g\}$ /* remove g from $\mathbb{G}$ */
end while

Running the algorithm with the goal $\vdash \{\ \mathbf{P}\ \}$ T:m() $\{\ \mathbf{Q}\ \}$ as input leads to a proof tree with three open slots: 1) $\vdash \{\ \tau = T \wedge \mathbf{P}\ \}$ T@m() $\{\ \mathbf{Q}\ \}$ 2) $\vdash \{\ \tau = S1 \wedge \mathbf{P}\ \}$ T@m() $\{\ \mathbf{Q}\ \}$ and 3) $\vdash \{\ \tau = S2 \wedge \mathbf{P}\ \}$ S2@m() $\{\ \mathbf{Q}\ \}$ which represent proof obligations for the different implementations of the virtual method T:m. Furthermore it can be seen that 1) and 2) only differ in the type of *this*. If the method implementations T@m and S2@m do not depend on the type of *this* it would be sufficient to prove the triple $\vdash \{\ \tau \preceq T \wedge \mathbf{P}\ \}$ T@m() $\{\ \mathbf{Q}\ \}$, because 1) and 2) are directly derivable from it. This could be done by a strategy too. The strategy coded within the algorithm shows that properties of a virtual method can be automatically reduced to properties of implementations. The proof construction, which is performed automatically, is guided by the structure of the subtype relation of the underlying program.

3.2 Verification of Method Implementations

The strategy described in the preceding section leads to proof obligations for method implementations. Thus at this point it is useful to have strategy support for this task. The proof of method implementations is ultimately based on the proof its bodys statement sequence. To automate this one can use techniques similar to weak precondition generation with extensions for method invocations. For all statements except method invocations, where an appropriate method specification has to be used, a proof can be constructed automatically. For all simple statements like assignment-, field-read- and field-write-, cast-, and empty-statement exists a forward operation which instantiates the appropriate axiom and thus constructs a forward proof fragment for that statement. For composed statements like block-, if-, sequence- and loop-statements there exist forward proof operations which construct a proof tree for that statement from proofs for its components. Combining these operations in an appropriate algorithm results in a strategy which performs a weak precondition generation proof and constructs a proof tree for the given statement sequence. As we consider object-oriented programs, the difference of this algorithm to described techniques in the literature [Gri81] is that we have to heavily care about method invocations.[3] If the strategy reaches an invocation statement, interaction with the user is needed. The user obtains the information at which method invocation (e.g. of method S:n) the strategy has stopped and which postcondition $\mathbf{Q}$ is generated for that invocation statement up to that point. He now can 1) select one of the given method specifications for that method and interactively adapt it to the local needs or 2) provide a precondition $\mathbf{P}$ and continue the strategy leaving the slot $\vdash \{\ \mathbf{P}\ \}$ S:n() $\{\ \mathbf{Q}\ \}$ open in the current proof and care later about it. The following algorithm sketch summarizes the technique described above:

Algorithm *wp*
Input: A statement *stmt* and a postcondition $\mathbf{Q}$.
Result: A proof tree for $\vdash \{\ \mathbf{P}\ \}$ stmt $\{\ \mathbf{Q}\ \}$ with a generated precondition $\mathbf{P}$ and postcondition $\mathbf{Q}$

if *stmt* matches **; then**

[3] Remember that it is in general not possible to generate a sufficiently weak precondition if the underlying programming language allows recursion, dynamic binding and aliasing.

return use inst_empty(stmt,Q) /* empty statement */
else if *stmt* matches **return _ ; then**
return use inst_field-read(stmt,Q) /* field-read statement */
else if *stmt* matches **stmt;stmts then**
$g1 \leftarrow$ wp(stmts,Q), $g2 \leftarrow$ wp(stmt,precondition(g1)) /* statement-list */
return seq_forward($g1$,$g2$)
else if *stmt* matches **if(expr) stmt1 else stmt2 then**
$g1 \leftarrow$ wp(stmt1,Q), $g2 \leftarrow$ wp(stmt2,Q) /* if-statement */
$f \leftarrow$ (precondition($g1$) $\wedge$ *expr*) $\vee$ precondition($g2$) $\wedge \neg$*expr*)
$g1 \leftarrow$ use strength_forward(g1, $f \wedge$ *expr*), $g2 \leftarrow$ use strength_forward(g2, $f \wedge \neg$*expr*)
return if_forward($g1$,$g2$)
else if *stmt* matches **_ = _ (_,...,_) then**
user interaction at this point results in a proof fragment for the /* method invocation */
method invocation
else if ... **then**
/* similar for while-, field-write-, assign statement, cast-, and return-statement */
/* while-loops could additionally enforce interaction e.g. for loop-invariants */
end if

The above sketched algorithm shows how traditional weak precondition generation can be enhanced to be useful in the area of object-oriented programs. User interaction and thus the proof for a method invocation statement can be considered as a strategy of its own. The overall algorithm divides the given program part *stmt* recursively into program fragments and constructs a proof with a weak precondition in forward direction while unrolling the recursion. Therefore in this example a program proof for *stmt* and postcondition Q with a computed precondition is constructed automatically guided by the program structure.

3.3 Handling Recursion

In the preceding section we did not care about the handling of recursion and assumptions. As OO-programs make direct or indirect (e.g. via dynamic dispatch) of recursion, an important task which can be supported by automatically working strategies is the elimination of assumptions, which are used to unroll recursion. In the following we demonstrate an assumption elimination strategy for method implementations to give an impression of this technique. Suppose you want to prove A_i[4], $1 \leq i \leq n$. 1. Prove $B_j := \bigcup_{i=1\ldots n} A_i \vdash \mathit{body}(A_j)$, $j = 1 \ldots n$. 2. Derive $C_j := \bigcup_{i=1\ldots n, i\neq j} A_i \vdash A_j$ from B_j, for $j = 1 \ldots n$ using the implementation rule. 3. Eliminate step by step all assumptions in $C_{1\ldots n}$, for example use $C_1 = \bigcup_{i=2\ldots n} A_i \vdash A_1$ to eliminate A_i in $C_2, \ldots, C_n$ etc. Strategies like this can be optimized with a preceding syntactical program analysis. One can for example reduce the assumptions for the proof of each $\mathit{body}(A_i)$ if one previously computes the methods which are called by its implementation.

implementation-rule:

$$\frac{\mathcal{A}\ ,\ \{\mathbf{P}\}\ \mathrm{T@m}(\mathrm{T}_{p_1}\ \mathrm{p}_1, \ldots, \mathrm{T}_{p_n}\ \mathrm{p}_n)\ \{\mathbf{Q}\} \vdash \{\mathbf{P} \wedge \mathrm{this} \neq \mathit{null} \wedge \textstyle\bigwedge_i (\mathrm{v}_i = \mathit{init}(\mathrm{TV}_i))\}\ \mathrm{BODY}(\mathrm{T@m}(\mathrm{T}_{p_1}\ \mathrm{p}_1, \ldots, \mathrm{T}_{p_n}\ \mathrm{p}_n))\ \{\mathbf{Q}\}}{\mathcal{A} \vdash \{\mathbf{P}\}\ \ \mathrm{T@m}(\mathrm{T}_{p_1}\ \mathrm{p}_1, \ldots, \mathrm{T}_{p_n}\ \mathrm{p}_n)\ \ \{\mathbf{Q}\}}$$

3.4 Combining Syntactic Checks and Verification

If full information about the program structure and the results of the analysis of static program properties like variable binding, type analysis, and invocation call graph is available, it is possible to combine that information. An example for this is to prove the property that a method does not change the object store. A method does not change the object store, if its implementation does not change the object store. This means especially that invoked methods do not change

[4]Where A_i abbreviates $\vdash \{\ \mathbf{P}\ \}\ \mathrm{T@m}\ \{\ \mathbf{Q}\ \}_i$.

the object store. A conservative approximation of this is that a method has no writing attribute access (conservative, because an implementation could undo an object store modification). The absence of writable attribute access is a static program property which can be computed by a static program analysis, e.g. during a syntactical program analysis. This information can be used to automatically compute program proofs for the invariance of the object store, because the proofs for methods with this property are simple and can be automated. Notice that the syntactic checks can be as complex as any static program analysis.

As object-oriented programs usually contain lots of methods with special syntactic properties (think of the `get`-methods of `get-set` pairs), such techniques combining static analysis and tactical theorem proving can reduce the amount of interactive proof work a lot. To exploit such benefits, the verification environment has to be sufficiently powerful to perform static program analyses.

3.5 Proof Guidance as Strategy

The proof that a specified object-oriented program fulfills its specification is ultimately based on the proof of several different properties, whose proofs can depend on each other. Because manual proof work and automized proof work are mixed, there may be proof states, where several completed and uncompleted proofs can exist simultaneously. To provide an overall proof guidance, a main strategy can be formulated which examines the current proof state, combines existing proof fragments, and continues the overall proofs by delegating proof obligations to substrategies. Thus these main strategy is a strategy, which is based on the state of all current proofs.

4 System Requirements

Within this extended abstract we presented overall techniques for proving OO-programs. From the demonstrated strategies the following properties of a tactical theorem prover can be derived: 1. Tactical program provers have to provide possibilities to formulate strategies as sketched within the examples. 2. It must be possible to manage proof obligations and proof parts. Furthermore it must be possible to inspect the current proof state and proofs. 3. Proof construction can be guided by several syntactical program properties, e.g. by the program structure, by the subtype relation, or by the call graph. Strategies and operations must have access to this information, which is needed during the whole proof process. 4. Interaction in strategies is needed to overcome problems of proof complexity. The described techniques are part of a research project, which currently led to the development of an interactive proof tool (cf. Jive[MPH00]) for a subset of Java. Jive currently supports the above described techniques.

References

[GMW79] M. Gordon, A. Milner, and C. Wadsworth. Edinburgh lcf. 1979.

[Gri81] David Gries. *The Science of Programming.* Springer-Verlag, 1981.

[LW94] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6), 1994.

[MPH00] J. Meyer and A. Poetzsch-Heffter. An architecture for interactive program provers. In S. Graf and M. Schwartzbach, editors, *TACAS00, Tools ans Algorithms for the Construction and Analysis of Software*, volume 276 of *Lecture Notes in Computer Science*, pages 63–77, 2000.

[PH97] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, January 1997.

[PHM99] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In D. Swierstra, editor, *ESOP '99*, LNCS 1576. Springer-Verlag, 1999.