Preserving the Correctness of Object-Oriented Programs under Extension

Peter Müller and Arnd Poetzsch-Heffter*
Fachbereich Informatik
Fernuniversität Hagen

Abstract

In object-oriented programming, software is mainly constructed by composition and specialization of types. Due to dynamic binding, program correctness may be invalidated by adding new types to existing programs. Essentially, two problems can occur: 1. Adding a new subtype S may violate the specification of its supertype T; thus components using T may be invalidated. 2. In certain cases, type invariants can be violated by adding new types to existing components. This paper sketches solutions to these problems. It claims that behavioral subtyping is a solution to the first problem. As a possible solution to the second problem, it proposes techniques to make interface specifications more expressive, to restrict the form of invariants by semantical constraints (similar to behavioral subtyping), and to refine existing module concepts.

1 Introduction

This paper summarizes a talk held at the "Kolloquium Programmiersprachen und Grundlagen der Programmierung" in Fehmarn. The presented work is based on the formal framework presented in Arnd Poetzsch-Heffter's talk "A Logic for the Verification of Object-Oriented Programs" in Fehmarn. Thus, it is assumed that the reader is familiar with the formal foundations which are not repeated here.

The objective of this paper is to describe the problems that can occur when object-oriented programs exploiting side-effects and destructive updates are extended. We focus on object-oriented programs for three reasons: (1) Object-oriented languages provide support for component-based program development (see below). (2) The role of object-oriented languages in industrial software development becomes more and more important. (3) Encapsulation and explicit interfaces ease the specification and verification of programs. We do not exclude side-effects, because they are at least important for the specification and verification of low level program components and occur in most of the existing class libraries.

Object-oriented programming languages provide means to support component-based program construction: Specialization through subtyping/subclassing, code inheritance, and encapsulation mechanisms to guarantee integrity constraints on data representations. In today's software construction, this support is unfortunately limited to syntactic checks.

*[Peter.Mueller, Arnd.Poetzsch-Heffter]@Fernuni-Hagen.de

Programmers are able to invalidate supertype specifications by adding subtypes or invariants of existing types by adding new types. As an example for the first problem consider a subtype LS of a list type where method `insert` of LS does nothing. Such misbehavior can in general not be detected by syntactic checking only. As an example for the second problem consider objects of a type C referencing objects of a type D. Assume that the invariant for C states that there is never a chain of references from C objects to C objects. By adding a new subtype of D that has a C attribute this invariant can be invalidated.

The use of semantic-based specification and verification techniques can overcome the above problems and lead to a systematic construction of correct programs from correct components.

Overview In the rest of this section, we sketch the formal background needed to make later explanations sufficiently detailed. In section 2, we describe the problems involved with the extension of object-oriented programs. In particular, we summarize the proof obligations for imported and declared code. Sections 3 and 4 discuss how these obligations can be solved or reduced. Our conclusions are contained in section 5.

Formal Framework Formal verification requires a formal semantics of the programming language and of the specifications. In this subsection, we give a quick overview of our framework. The reader may refer to [PH97, MPH97] or to Poetzsch-Heffter's paper in this report for a detailed description of the formal basis.

Our specification technique is based upon the two-tiered Larch approach (cf. [GH93]). Program specifications consist of two major parts: (a) A program-independent specification which provides the mathematical vocabulary (e.g., definitions of abstract data types) and (b) a program dependent part that relates the implementation to universal specifications. An interface specification of a type C consists of (a) a specification for each public method of C, and (b) a type invariant. Method specifications are given by pre- and postconditions. Type invariants describe properties that have to hold for each object of a type in any state where the object is accessible from outside.

In the implementation, values of abstract data types are in general represented by several linked objects. We consider such object structures as a whole. Going beyond Larch, the relation between object structures and values of abstract data types is made explicit by so-called abstraction functions. Modification of an object X possibly changes the abstractions of all object structures referencing X. As a consequence, the treatment of side-effects becomes very important. The definition of abstraction functions and formal specification of side-effects require a formalization of the data and state model of the programming language. The data model of a programming language defines the objects and values that may be used in programs. The state of an object tells whether the object is allocated, and it assigns an object to each of its attributes. The collection of all object states at a point of program execution is called the current *object environment*, denoted by the global variable \$.

For verification, we assume a Hoare-style programming logic as presented in [PH97]. This logic is a formalization of the axiomatic semantics of the underlying programming language. Thus, correctness of a program is showed by translating its specification into Hoare triples and proving these triples in the logic. This translation is described in section 2.1.

2 Program Extensions

In this section, we present an example program and discuss the problems caused by composition of modules. In particular, we show that behavioral subtyping is a solution to some of those and summarize the proof obligations resulting from the remaining problems.

2.1 Discussion of Problems

As a beginning of our discussion, we sketch the problems caused by program extensions.

Example Program As an example, we consider an abstract type ALIST which describes the interface of a integer list. Furthermore, we have a type DLIST which implements ALIST via doubly linked lists. We assume a type DLELEM to store the nodes of this list.

```
abstract type ALIST is
  public empty():         ALIST
  public append(i: INT): ALIST
  public first():         INT
  public rest():          ALIST
end
```

```
type DLIST subtype of ALIST is
  att head: DLELEM
  public empty():         DLIST
  public append(i: INT): DLIST
  public first():         INT
  public rest():          DLIST
end
```

We assume, that ALIST is formally specified. E.g., the type invariant of ALIST states that the list contains only positive integer values: $inv_{ALIST}(X, E) \Leftrightarrow_{def} positive\_entries(X, E)$ where X is an ALIST object and E denotes an object environment. The specification of method ALIST:append states that whenever the parameter is positive, the method will append the actual parameter to the implicit parameter and return the resulting list. This is expressed by use of an abstraction function aL (cf. section 1) which maps objects of type ALIST to values of an abstract data type $List$ (app is a constructor of $List$):

```
ALIST:append(i:  INT): ALIST
  pre   aL(this, $) = L ∧ i > 0
  post  aL(result, $) = app(L, i)
```

Behavioral Subtyping To get a first idea of the problems involved with program extensions, we assume a program $\mathcal{P}$ that consists of type ALIST and a type C which makes use of ALIST. $\mathcal{P}$ is assumed to be verified. Now we extend $\mathcal{P}$ by type DLIST. Which conditions have to be fulfilled by DLIST to preserve correctness of $\mathcal{P}$? This question has two aspects: (1) What is the relation between the method specifications of DLIST and ALIST? (2) What is the relation between the invariants of DLIST and ALIST?

Relation of Method Specifications To answer the first questions, we consider a program part of type C which makes use of properties of method ALIST:append:

```
(1)   var v: ALIST;
(2)   ...
(3)   v := v.append(5);
(4)   ...
```

As this fragment is assumed to be verified, it is guaranteed that the precondition of ALIST:append holds before execution of statement (3), and that the postcondition holds after that. When $\mathcal{P}$ is extended by type DLIST, `v` may hold objects of type DLIST as DLIST is a subtype of ALIST. Due to dynamic binding, DLIST:append may now be called in line (3). Thus, preserving correctness requires DLIST:append to work correctly in contexts where calls to ALIST:append appear. I.e., DLIST:append has to show the behavior specified for ALIST:append: Whenever the precondition of ALIST:append holds, and DLIST:append is called, the postcondition of ALIST:append has to hold after termination of DLIST:append. In general, this can be formulated as follows: Let S be a subtype of T. For each method m associated with S and T with pre-post-pairs (R_S, Q_S) and (R_T, Q_T), R_T implies R_S and Q_S implies Q_T in case the `this` object is of type S[1]. I.e., S:m shows the behavior specified for T:m.

Relation of Invariants To understand the relation between the invariants of the super- and the subtype, we consider the following program fragment from type C:

```
(1)   var v: ALIST;
(2)   ...
(3)   i := v.first().sqrt();
(4)   ...
```

From the invariant of ALIST, we can deduce that first returns a positive integer. Thus, the computation of the square root is defined. To preserve correctness, this argument has to hold for DLIST objects as well. I.e., each DLIST object has to fulfill the type invariant of ALIST. In general, $inv_S(X, E)$ has to imply $inv_T(X, E)$ for all X of type S, where S is a subtype of T.

Subtypes that fulfill the requirements sketched in the last two paragraphs are usually called *behavioral subtypes* (cf. [Ame87] for more details). As shown by the two examples above, behavioral subtyping is a prerequisite to preserve program correctness under extension. Unfortunately, behavioral subtyping is not sufficient as type invariants can still be violated by additional types. We discuss this topic in the following.

Semantics of Type Invariants To motivate the semantics of type invariants, we consider a program $\mathcal{P}$ with types ALIST, SLIST, and C. ALIST is defined as shown above. SLIST implements ALIST via acyclic singly linked lists. Type C is assumed to contain an attribute `l` of type ALIST. The invariant of C states that the list stored in `l` has to be acyclic. As SLIST is the only implementation of ALIST, this invariant certainly holds. Now we extend $\mathcal{P}$ by type DLIST. In addition to the definition above, DLIST is assumed to contain a method `make_cycle` that forms a cyclic list. Although DLIST is a behavioral subtype of ALIST, correctness of $\mathcal{P}$ is violated by adding DLIST as the invariant of C may no longer hold. To cope with such effects, we have to assign a very strong meaning to type invariants.

We want the type invariant of a type C to hold for all objects of type C in all states where these objects are accessible from outside. Thus, we require invariants to be invariant under execution of public methods. I.e., if the invariants hold for all objects in the

[1] This requirement is stronger than necessary, but sufficient for the purposes of this paper.

precondition of *any public method*, they should hold in the postcondition. In particular, they have to hold for objects created during method execution.

Certainly, invariants need not hold in all intermediate states during execution of C's methods; in particular during the construction of linked object structures, invariants are usually violated. We require invariance only for public methods, because this guarantees that the invariant of a type C holds outside the execution of methods of C and because we want to allow private methods to perform auxiliary operations violating, e.g., well-formedness properties. To use type invariants as invariants in the proof technical sense, they have (a) to be true in possible initial program states and they have (b) to be invariant under *all* public methods. Requirement (a) is trivially satisfied because no user-defined objects are allocated in initial program states. Requirement (b) is the proof obligation resulting from invariants. Although requirement (b) is as well justified from an operational point of view — a method m_C of type C can call a method m_D of type D and thus manipulate D-objects —, the literature often assigns a weaker meaning to invariants which makes verification much more difficult and leads to unintuitive situations.[2]

We define the formal semantics of specifications by interpreting them as triples in a Hoare-style logic. Thus, the connection between specifications and programs is precisely defined. Let us assume a program $\mathcal{P}$ with types $C_1 \ldots C_n$. The specification of $\mathcal{P}$ consists of the type invariants INV_{C_i} and of a pre-postcondition-pair (P_m, Q_m) for each method m. To verify $\mathcal{P}$, we have to prove a triple of the following form for each public method m, where \$ denotes the current object environment:

$$\{\, P_m \wedge \bigwedge_{i=1}^{n} INV_{C_i}(\$) \,\} \text{ meth m } \{\, Q_m \wedge \bigwedge_{i=1}^{n} INV_{C_i}(\$) \,\}$$

To prove these properties for an extended program, enforcing behavioral subtyping is not sufficient. Further proof obligations have to be fulfilled to guarantee that all invariants hold in the extended program as well. These obligations are summarized in the next subsection.

2.2 Summary of Proof Obligations

Recall, that specifying a type invariant means that this invariant has to be preserved by all public methods of a program. Assume a module M with types C_1 to C_m. M does not import any modules. Verifying M means to prove the triple above for each public method of M.

Consider a module N which imports M. N contains the types C_{m+1} to C_n. I.e., the resulting program $\mathcal{P}$ is a composition of the modules M and N containing the types C_1 to C_n. What triples have to be shown to verify $\mathcal{P}$? Again, we have to prove that every public method of $\mathcal{P}$ preserves each invariant of any C_i. For most practical applications, this is equivalent to showing that (1) C_M:m preserves INV_M, (2) C_M:m preserves INV_N, (3) C_N:m preserves INV_M, and (4) C_N:m preserves INV_N, where C_M:m and C_N:m denote a public method m of a type declared in module M or N, respectively and INV_M and INV_N denote the conjunction of all type invariants in module M or N.

[2]The weaker meaning requires each method to preserve the invariant of the type it belongs to. The example above sketched a situation where the invariant of type C was violated by a method of type DLIST. Thus, we cannot make use of inv_C during the verification of C because it may not hold.

Obligation 1 is fulfilled as M is verified. Obligation 4 causes no further problems as it can be solved locally in module N (maybe using some specifications of M). Obligation 2 means to show properties of an imported module. This is not desirable as the imported code may come from a library and, thus, be not accessible to the verifier. Obligation 3 results in a huge amount of proof obligations for complex components as it contains one triple for each type in the import path of the module. So, in large systems, hundreds or thousands of triples have to be proved. We will discuss possible solutions to the problems caused by obligations 2 and 3 in the next two sections.

3 Proving Obligations for Imported Code

If imported code comes from a software library, clients of that code may only have access to interfaces and specifications, not to the implementation. Thus, we have to develop techniques that allow one to prove obligations of the second kind (see above) without revisiting imported code. This can be achieved by two approaches: Preservation of the invariants can be proved using specifications of imported methods, or can be achieved by requiring the invariants of new types to fulfill certain semantical constraints.

3.1 Expressiveness of Type Invariants

Not every property that is expressible in a specification framework is desirable as type invariant. Consider the following invariant of a type T: $inv_T(X,E) \Leftrightarrow_{def} \forall Y : alive(Y,E) \Rightarrow typ(Y) = T$. This invariant states that every allocated object Y of a program is of type T. Of course, invariants of that kind are not desired because they cannot be proved in any reasonable program. Thus, we'd like to constrain the expressiveness of invariants to forbid such ill-formed specifications.

Type invariants of a type T are meant to express properties of objects of type T or objects referenced from such objects. This can be enforced by putting semantical constraints on type invariants. To express reachability, we use a predicate $reach(X,L,E)$ which holds iff a location L can be reached by an object X in an environment E. A location is reachable if there exists a chain of references from X to L. Based on *reach*, we can define an equivalence on object environments. Environments E and E' are equivalent from the point of view of an object X, if X is alive (i.e. allocated) in E iff it is alive in E' and if all locations reachable from X hold the same objects in E and E':

$$E \equiv_X E' \Leftrightarrow_{def} \ (alive(X,E) \Leftrightarrow alive(X,E')) \wedge \forall L : reach(X,L,E) \Rightarrow E(L) = E'(L)$$

This definition allows us to constrain invariants:

$$E \equiv_X E' \Rightarrow (inv_T(X,E) \Leftrightarrow inv_T(X,E'))$$

Invariants that fulfill this constraint can only express properties of X and objects reachable from X. This property is used to prove the preservation of invariants in the following.

3.2 Method Specifications

To deduce the preservation of invariants from method specifications, these have to be sufficiently strong and detailed. E.g., if the specification of method m states that m only

modifies objects of type C, and if objects of a type D cannot reach C objects in memory, the invariant of D cannot be violated by m. This example shows two important aspects of method behavior: sharing properties and invariance properties.

Specifying Sharing and Invariance Properties Sharing properties describe how objects in memory are shared by different object structures. E.g., the last object of a singly linked list is shared by all other list objects of the list structure. Specification of sharing properties is crucial for verification because, in general, modification of one object changes the abstract value of all structures that share this object. Basic constructs to express sharing properties are reachability of objects (see above) or disjointness of object structures.

Invariance properties are used to describe side-effects of methods by specifying which parts of the object environment remain unchanged under method execution. This can be done by relating the pre- and poststate of the method. Again, this relation can be expressed by defining an equivalence on object environments. Two environments E and E' are considered equal for a type T, if all objects that are *not* of type T are alive in E iff they are alive in E' and if all locations that are *not* part of a T object hold the same objects in E and E'. Thus, E and E' are equal except that T objects may be created or modified:

$$E \equiv_T E' \Leftrightarrow_{def} \begin{array}{ll} typ(X) \not\trianglelefteq T & \Rightarrow (alive(X,E) \Leftrightarrow alive(X,E')) \wedge \\ typ(obj(L)) \not\trianglelefteq T & \Rightarrow E(L) = E'(L) \end{array}$$

T-equivalence can be used to specify invariance properties of methods: If the pre- and poststate of a method m are T-equivalent, m leaves all objects that are not of type T unchanged. We demonstrate this by method DLIST:append from above:

```
DLIST:append(i:  INT): DLIST
  pre   $ = E
  post  $ ≡DLIST E ∨ $ ≡DLELEM E
```

DLIST:append may only modify (or create) DLIST and DLELEM objects.

Exploiting Sharing and Invariance Properties To demonstrate the application of the specification above, we go back to our example program $\mathcal{P}$. $\mathcal{P}$ consists of the types ALIST, DLIST, and DLELEM. We want to extend $\mathcal{P}$ by type STACK which implements an integer stack with the usual operations. The height of the stack is stored explicitly:

```
type STACK is
  att s:      STACK
  att elem:   INT
  att height: INT
  ...
end
```

STACK contains any invariant inv_{STACK} which meets the constraint described in section 3.1. Now we want to prove that DLIST:append preserves the invariant of STACK. This can be done by the following lemma:

$$(E \equiv_T E' \wedge (reach(X,L,E) \Rightarrow typ(obj(L)) \not\trianglelefteq T)) \Rightarrow E \equiv_X E'$$

If two environments only differ in locations of T objects and if an object X does not reach such a location, the environments are X-equivalent (the proof of this lemma is omitted for brevity). If we can prove that any STACK object X does not reach a DLIST or DLELEM object, we can derive that the pre- and postcondition of DLIST:append are X-equivalent (cf. specification of DLIST:append). Because of the constraint for invariants, this implies that $inv_{STACK}(X, \$)$ holds in the poststate of DLIST:append if it held in the prestate; it is preserved by DLIST:append.

The fact that STACK objects do not reach DLIST or DLELEM objects can be showed by a syntactic analysis of type STACK. As STACK only contains attributes of types STACK and INT, STACK objects can only reach STACK and INT objects. In particular, they cannot reach DLIST or DLELEM objects. Syntactic analysis of that kind often ease the effort of showing proof obligations.

3.3 Semantical Constraints

The argument applied in the last paragraph does no longer hold if we generalize STACK to an OBJECT stack instead of an integer stack, because STACK object can now reach any object of a program[3]:

```
type STACK is
  att s:      STACK
  att elem:   OBJECT
  att height: INT
  ...
end
```

Thus, method specifications as used above are not sufficient to show the preservation of invariants. Therefore, we have to constrain the invariants of types that are added to a program.

A typical semantical constraint was presented in section 3.1. By putting stronger constraints on invariants, we can achieve that the invariants of new types cannot be violated by imported methods. E.g., if an invariant only states properties of objects of types that are declared in the current module (i.e., not imported), this invariant can only be violated by methods of these types or their subtypes, i.e., not by imported methods. We demonstrate this by our example program $\mathcal{P}$.

The idea is as follows: If the invariant of STACK does not make any statements about objects reachable through `elem`, is cannot be violated by an imported method. To express this constraint, we introduce X-M-equivalence where X is an object an M is a module[4]:

$$E \equiv_X^M E' \Leftrightarrow_{def} \begin{array}{l} alive(X, E) \Leftrightarrow alive(X, E') \wedge \\ reach(X, L, E) \wedge \neg M \text{ imports } mod(typ(obj(L))) \Rightarrow E(L) = E'(L) \end{array}$$

X-M-equivalence is weaker than X-equivalence because only those locations have to hold the same in location in E and E' that belong to an object whose type is not imported by

[3] Every type is a subtype of OBJECT.

[4] $mod(T)$ denotes the module a type T is declared in.

M. Thus, the following semantical constraint is stronger than the constraint presented in section 3.1:

$$E \equiv_X^{mod(T)} E' \Rightarrow (inv_T(X, E) \Leftrightarrow inv_T(X, E'))$$

This constraint guarantees that the invariant cannot be violated by any imported method. Despite this strong constraint, many of the desired properties of STACK can still be specified, e.g. the correct use of attribute `height`:

$$inv_{STACK}(X, \$) \Leftrightarrow_{def} \ \$(X.height) = 0 \ \vee \ \$(X.height) = \$(\$(X.s).height + 1)$$

4 Reducing Obligations for New Code

Although it is theoretically possible to prove that every new method preserves the invariant of each imported type (as the code is accessible), this is not desired because of two reasons: (1) In large systems the proof obligations become too big and, thus, unmanageable. (2) To support encapsulation of implementations, it is desirable to hide implementation-dependent parts of invariants from client modules. In this section, we present the idea of semantic module concepts that allow one to drop the proof obligations about *semantically private types*.

Syntactic module concepts support the encapsulation of types. E.g., type DLELEM would be declared private in the module containing type DLIST. As the list can only be accessed via DLIST, DLELEM can be hidden from clients of the module. From a semantic point of view, this is no longer true, as clients of the module might violate the invariant of DLELEM or vice versa: A private type C may contain an attribute a of a public type D. Thus, the invariant of C, denoted by inv_C, may state properties of the objects stored in a. All methods of the module M containing C and D preserve inv_C. But, a client module N may contain a subtype of D providing a method that violates inv_C. Thus, we need a module concept which provides *semantically private types*, i.e., types that can be hidden from clients of the module in the sense that their invariants are not part of the interface of the module.[5] This helps to solve the two problems sketched above: (1) Semantically private types do not contribute to the proof obligations for client modules, which eases verification. (2) It is no longer necessary to reveal details of the implementation by exporting invariants which specify properties of the data representation.

Semantical privacy can be achieved by enforcing certain syntactical restrictions of the implementation or by putting semantical constraints on invariants. To illustrate this approach, we introduce a predicate P on types. $P(T)$ holds, iff T is final (i.e., has no subtypes) and P holds for the type of each attribute of T. The invariant of a type for which P holds can not be affected by program extensions as subtyping is ruled out. This simple syntactical limitation is of course too restrictive as it forbids the use of subtyping. By combining this idea with a simple constraint on the invariant, the situation becomes better: The invariant of T may only state properties of objects which are reachable via attributes whose types fulfill P. This combination enables semantical privacy and a restricted use of subtyping.

Pushing the constraints on invariants further leads to the following notion: The invariant of a type C has only access to the invariants of reachable objects, not to their

[5]Note, that a type can be semantically private despite being syntactically public.

implementation. In combination with behavioral subtyping, this constraint assures the preservation of the invariant. The disadvantage of this approach is that it leads to an increasing number of types as showed by the following example: Assume, we have a type `fraction` with two attributes `numerator` and `denominator` of type `int`. The invariant states that the denominator is non-zero. Now we implement a type C which has an attribute a of type `fraction`. We want to specify that the fraction stored in a is non-negative. Because of the constraint sketched above, this can not be specified in the invariant of C as it is a property of `fraction`. Thus, we have to build a new type which essentially behaves like `fraction` but has a stronger invariant.[6] This approach fully supports subtyping.

5 Conclusion

Extension of object-oriented programs may violate program correctness. We showed that behavioral subtyping is a prerequisite to preserve correctness. Furthermore, certain obligations for imported and newly declared methods have to be proved to show preservation of type invariants.

Properties of imported methods can be proven without knowing their implementation. This can be achieved by exploiting sharing and invariance properties of imported methods or by putting semantical constraints on the invariants of newly declared types. We claimed that proof obligations for newly declared methods can be reduced by semantic module concepts supporting semantically private types. Further work will be concerned with elaborating the ideas sketched in this paper.

References

[Ame87] Pierre America. Inheritance and subtyping in a parallel object-oriented language. In Jean Bezivin et al., editors, *ECOOP '87, European Conference on Object-Oriented Programming, Paris, France*, pages 234–242, NY, June 1987. Springer-Verlag. Lecture Notes in Computer Science, Volume 276.

[GH93] J. V. Guttag and J. J. Horning. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag, 1993.

[MPH97] P. Müller and A. Poetzsch-Heffter. Formal specification techniques for object-oriented programs. In M. Jarke, K. Pasedach, and K. Pohl, editors, *Informatik 97: Informatik als Innovationsmotor*, Informatik Aktuell. Springer-Verlag, 1997.

[PH97] A. Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs.* PhD thesis, Technische Universität München, 1997. (Habilitationsschrift).

[6] The stronger invariant requires stronger preconditions for some methods, e.g., the multiplication method. Thus, the new type is not a behavioral subtype of `fraction`.