Verification of Object Oriented Programs Using Class Invariants

Kees Huizing, Ruurd Kuiper, and SOOP*

Eindhoven University of Technology, PO Box 513, 5600 MB Eindhoven,
The Netherlands,
keesh@win.tue.nl, wsinruur@win.tue.nl

Abstract. A proof system is presented for the verification and derivation of object oriented programs with as main features strong typing, dynamic binding, and inheritance. The proof system is inspired on Meyer's system of class invariants [12] and remedies its unsoundness, which is already recognized by Meyer. Dynamic binding is treated in a flexible way: when throughout the class hierarchy overriding methods respect the pre- and postconditions of the overridden methods, very simple proof rules for method calls suffice; more powerful proof rules are supplied for cases where one cannot or does not want to follow this restriction.
The proof system is complete relative to proofs for properties of pointers and the data domain.

1 Introduction

Although formal verification is not very common in the discipline of object oriented programming, the importance of formal specification is generally acknowledged ([12]). With the increased interest in component based development, it becomes even more important that components are specified in an unambiguous manner, since users or buyers of components often have no other knowledge about a component than its specification and at the same time rely heavily on its correct functioning in their framework. The specification of a class, sometimes called *contract*, usually contains at least pre- and postconditions for the public mehtods and a class invariant.

A class invariant expresses which states of the objects of the class are consistent, or "legal". An object that doesn't satisfy the class invariant has an uninterpretable state that should only occur during an update of the object. Therefore, whenever an object is handed over from one piece of the code to the other (and therefore possibly from one developer to another), one should be able to assume the class invariant to hold for the object. Of course, this is what the term "invariant" conveys. Nevertheless, the usual practice in verification is too

* Research group Systematic Object Oriented Programming, at the time of conception of this paper consisting of Lex Bijlsma, Rik van Geldrop, Louis van Gool, Kees Hemerik, Kees Huizing, Ruurd Kuiper, Onno van Roosmalen, Jaap van der Woude, and Gerard Zwaan

T. Maibaum (Ed.): FASE 2000, LNCS 1783, pp. 208–221, 2000.

weak to guarantee this invariance. According to this practice, one proves that every (public) method that starts in a state satisfying the invariant will also terminate in a state satisfying the invariant. As Meyer already remarks, this does not in general guarantee invariance [12] and hence, leads to unsoundness. When the call chain of methods can visit a certain object twice, the second call may find the object in an inconsistent state, i.e., not satisfying the invariant. This situation of *reentrance*, sometimes called *call-back*, occurs in many object oriented designs. Another problem may occur when a class invariant depends on the state of more than one object. In this case, changing the state of one object may break the invariant of another [15]. In this paper, this problem is called *vulnerability*.

To overcome these problems, we formulate a proof system that does guarantee true invariance of the class invariants. This leads to more proof obligations than just "assuming that the invariant holds before, prove that it holds after", but it makes problems with call-backs and fragile base classes [15] visible as early as possible in the development.

Proof theoretic approaches comparable to ours but not dealing specifically with the questions addressed in the present paper, i.e., call-backs and vulnerability, can be found in, e.g., [9,14,13,8,10]. A somewhat more different proof theoretic approach appears in [1]. A semantically oriented approach, aiming for automated object-oriented verification, is proposed in [6].

2 Framework

2.1 Programming Language

The proof system works in principle for any object oriented, strongly typed programming language. For our notation, we stay closely to Java [3]. With strongly typed, we mean that every expression has a static type and that type incompatibilities render a program illegal.

Furthermore, we expect our language to have dynamic binding, so we assume that values of expressions have a *dynamic type* that can be different from the static type of the expression. We assume that there is a subtype relation of types, which is a partial order. We assume that the language is *type safe*, i.e., the dynamic type of an expression is always a subtype of the static type. In practice, a language is seldom type safe in the sense that every legal, or compilable, program is type safe. Of course, one could design proof rules to prove that a program is indeed type safe. We will not go into this further. Inheritance follows subtyping. When C is a subclass of D, C is also a subtype of D.

2.2 Object References

An object reference is an expression that refers to an object or equals the special constant `null`. The value `null` does not refer to an object, hence it cannot be dereferenced, i.e., no method call or other member access can be applied to `null`.

We thus assume that object references are never undefined (such as pointing to garbage). This assumption can be achieved by having no explicit object deletion in the language but relying on automatic garbage collection instead. Non-`null` object references are equal if and only if they refer to the same object. Object references may occur in statements as well as in assertions.

The special variable `this` always refers to the currently active object, i.e., the object to which the currently executing method belongs. We assume that the only object that can be changed at any moment during execution is the currently active object. This can be achieved by not allowing assignments to fields of other objects, or more severely, only allowing access to objects via method calls. This latter restriction is advised in many object oriented design methodologies.

We assume that the static type of object references is always known. The proof system can derive equality and inequality of reference and type expressions. About the type system we assume:

1. the subtype relation $<:$ is transitive and reflexive;
2. we do not have subsumption, i.e., $A <: B$ does not imply that any expression of type A is of type B;
3. if $o.x$ is a type correct expression, it refers to a member that is declared in (a supertype of) the dynamic type of o; this member is uniquely determined, though not statically. If o is `null`, the value of the expression is undefined.

A new object of class C is created by the expression `new C()`. After creation, the *constructor* associated to class C, if it exists, is executed. In this paper, we allow at most one constructor per class. For the purpose of the proof system, we consider the expression `new` as a method call. The result value is a reference to the newly created object.

2.3 Assertion Language

The assertion language is first order predicate logic with local program variables as free variables. Concerning instance variables (also known as attributes), there is a slight complication, since their value depends on the object they belong to. Therefore, instance variables should always be prefixed with a reference to the object they belong to. In many cases, this reference is `this`, and this reference is silently assumed when we omit it, but it may be any other expression that yields an object reference.

Every assertion in the proof system is associated to a class. For the pre- or post-condition of a method, it is the class that the method belongs to; for class invariants it is that class, obviously; for annotations in the code it is the class the code is associated to. This class is called the type of the assertion. When an assertion is prefixed with an object reference, it is to be evaluated in the context of that object. We define this syntactically, as follows:

Definition: Let P be an assertion of type C and o an object reference of type C or a subtype thereof. Then we define $o.P \equiv P[\texttt{this}/o]$).

An unprefixed assertion is silently assumed to have `this` as a prefix.

If $o = \texttt{null}$, the value $o.P$ is undefined. Note that `this` never has value `null`.

2.4 Proof System

In an object-oriented program, the notion of a main program has more or less disappeared, leaving a set of classes for the "user" (programmer, developer) to work with. Therefore, the purpose of a proof system for object-orientation is not to prove a certain program correct, but to prove a system of classes correct. To achieve this, to every class we associate a class invariant and a pair of pre/postconditions to every method in the class. This specification can be used to infer properties about his own program. The proof system provides a method to prove that the class indeed satisfies its specification. Before we go into class invariants, we first give a sketch how pre- and postconditions are used in the proof system.

The idea is to put at certain places in the program text assertions that should be true whenever program execution arrives at that point. This is proved by fulfilling proof obligations, which are either implications in predicate logic, or Hoare-triples[7]. All methods are annotated with assertions. A legal annotation has assertions in at least the following places:

- before and after the body of the method;
- before and after every method call (including `new` statements).

Assume that an annotated method m in class C looks as follows (because the Java-style braces that surround blocks of code clash with Hoare-triple braces, we use `begin` and `end` for program blocks):

$\{pre\}$
`void m(void) begin`
  $\{Q_1\}$
  `... // code without method calls`
  $\{R_1\}$
  o_1`.method`$_1$`();`
  $\{Q_2\}$
  .
  .
  .
  $\{R_{n-1}\}$
  o_{n-1}`.method`$_{n-1}$`();`
  $\{Q_n\}$
  `... // code without method calls`
  $\{R_n\}$
`end`
$\{post\}$

So every method call o_i.method$_i$ is surrounded by a pair (R_i, Q_{i+1}), and the other pieces of code (so-called *local code segments*) by pairs (Q_i, R_i). To get consistent subscripting, it may be necessary to insert an empty statement between Q_1 and R_1 or Q_n and R_n, in case the method begins or ends with a method call.

Then we have the following proof three obligations. How to fulfill the last two, we will address later.

1. prove:

   $\texttt{this}.pre \Rightarrow Q_1$

   $R_n \Rightarrow \texttt{this}.post$

2. for every two assertions Q_i and R_i that are separated by local code P_i, prove the Hoare triple:

   $\{Q_i\}P_i\{R_i\}$

3. for every two assertions R_i and Q_{i+1}, surrounding method call $o.\text{method}_i()$, prove:

   $\{R_i\}o.\text{method}_i()\{Q_{i+1}\}$

We have simplified matters somewhat. We assumed that the method has no parameters, and we assume that the method body can indeed be written as a sequence of local code segments and method calls. It is relatively straigthforward to remove these simplifications; we will not go into this.

The second type of proof obligations, we call *local proof obligations*. How these proofs are established is not the concern of this paper. One can substitute one's favourite proof system, applied to one's favourite programming language, using one's favourite assertion language.

Regarding the third proof obligation, we recall the proof obligation for ordinary procedure calls. In that case, we would have to prove

$R_i \Rightarrow pre_{\text{method}_i}$ and $post_{\text{method}_i} \Rightarrow Q_{i+1}$

where pre_m and $post_m$ are the pre- and postcondition of the procedure, [4].

For method calls, the situation is somewhat more complicated and how to formulate the proof obligation in that case is the subject of the next section.

2.5 Proving Pre- and Postconditions for Methods

Before we discuss what has to be done for methods, we introduce some notation.

Notation

- $\mathcal{T}_d(o), \mathcal{T}_s(o)$ denote dynamic resp. static type of o
- $o.m$ denotes method m of the dynamic type of o (cf. Java method call)
- $o{:}m$ denotes method m of the *static* type of o
- m_C denotes method m in class C, so $o{:}m = m_{\mathcal{T}_s(o)}$
- $D <: C$ denotes that D is a subtype of C; note that the subtype relation is reflexive, so $D = C$implies $D <: C$.

Methods are specified by pre- and postconditions, just like procedures in ordinary sequential programming languages. Apart from a funny syntax to specify the first parameter, the important difference with procedures is that methods have *dynamic binding*. From the syntax of a method call like $o.m()$, we can not deduce which method will be executed, and, consequently, we do not know which pre/postcondition pair should be used. We propose two different solutions, which do not necessary exclude each other.

1. During class design, make sure that an overriding method (a method of a subclass that redefines a method of a superclass), can really substitute the method it overrides. I.e., the precondition of the overriding method should be weaker than the precondition of the method it overrides, and the postcondition should be stronger[1]. This leads to the proof obligations:
   For any two types $D <: C$ that both define method m, prove:
   - $pre_{m_C} \Rightarrow pre_{m_D}$
   - $post_{m_D} \Rightarrow post_{m_C}$

   Now we may use the following axiom in our proofs:

   $$\{o.pre_{am}\}o.m()\{o.post_{am}\}$$

2. Keep information about the dynamic type of an object in the assertions; then use the following axiom:

   $$\{\mathcal{T}_d(o) = C \wedge o.pre_{m_C}\}o.m()\{o.post_{m_C}\}$$

   At object creation, dynamic type information is inserted into the assertions, for instance by the following axiom:

   $$\{true\}o = \texttt{new}\, C()\{\mathcal{T}_d(o) = C\}$$

   Likewise for assignment:

   $$\{\mathcal{T}_d(o') = C\}o = o'\{\mathcal{T}_d(o) = C\}$$

These axioms take care of the type information, but of course don't capture all of the behaviour of object creation and assignment; we assume that the rest of the proof system takes care of that.

These two solutions do not exclude each other. Nevertheless, if 1 is used for a certain class C, then the associated proof obligations must be proved for C and all its subtypes. An advantage of approach 2 is that no such additional proof obligations are incurred. However, an advantage of approach 1 over approach 2 is that no dynamic type information needs to be recorded in the assertions.

[1] This is basically the methodology as elaborated in more detail in [10]

3 Class Invariants

The proof system described above can be unwieldy. One of the key issues of object oriented design is that objects represent something, possibly an object in the real world, or a more abstract entity, which is more than just the data that it contains. This means that, whenever one uses an object in a correctness proof, one may assume that it satisfies certain semantic properties. These properties are commonly shared among all objects of the same type and since object types are represented by classes in object oriented languages, they are called *class invariants* [12]. This is essentially an extension of the notion of *representation invariant*. Representation invariants are a well-known and powerful concept in the verification of data structures. In addition to representation invariants, a class invariant may talk about properties of linked objects too. Class invariants simplify the specification of methods by factoring out common properties. Furthermore, class invariants help simplifying correctness proofs: at a method call, there is in general only a proof obligation for the precondition of the method, not for the invariant of the called object, as we will see later.

We associate a class invariant with every class (when omitted, we assume the invariant to be *true*), and we write I_C for the invariant associated to class C. We use the notation $o.I$ for the class invariant of the actual class of o evaluated in the context of the object referred to by o. Note that this depends on the dynamic type of o, and without information about this dynamic type, we cannot deduce anything from $o.I$. This is different from the expression $o.P$ where P is a known predicate. The expression $o.I$, however, is merely an expression and is not an abbreviation for a predicate. Similar to the pre/postconditions for methods, we give two approaches.

1. Make sure that the invariant of a subtype is a strengthening of the invariant of the supertype. Formally,

   Proof obligation: Whenever $D <: C$, then prove $I_D \Rightarrow I_C$, $pre_{M_C} \Rightarrow pre_{M_D}$, and $post_{M_D} \Rightarrow post_{M_C}$.
   Then we can use the following

   Axioms:

   $$o.I \Rightarrow o{:}I \quad o{:}pre_m \Rightarrow o.pre_m \quad o.post_m \Rightarrow o{:}post_m$$

   This approach captures an important principle of object-oriented design: a subclass should be a specialization of its superclass, hence properties that hold for objects of a certain type C (here modelled by the class invariant I_C) should also hold for objects of subtypes of C.
2. Collect dynamic type information in assertions and use the following rules:

   $$\frac{\mathcal{T}_d(o) = C}{o.I \Leftrightarrow o.I_C} \quad \frac{\mathcal{T}_d(o) = C}{o.pre_m \Leftrightarrow o.pre_{m_C}} \quad \frac{\mathcal{T}_d(o) = C}{o.post_m \Leftrightarrow o.post_{m_C}}$$

Soundness of these rules follows from [10], where, in a different setting, the same proof obligations appear. In fact, the proof obligation of $pre_{M_C} \Rightarrow pre_{M_D}$ compromises completeness when D has more variables (attributes) than C. In that case, approach 2 can be used, or approach 1 should be refined along the lines of [10].

3.1 Where Do Class Invariants Hold?

Until now, we have not defined what it means for a class to satisfy its specification. We will do that now.

For pre/postcondition pairs, we would want that a method that is called in a state where the precondition holds should terminate in a state where the postcondition holds.

For class invariants, it is less evident what should be required. It is unrealistic to have class invariants hold all the time. When the data in an object changes, it is often not possible to keep the invariant valid during the whole process. However, when an object is handed over, or when a method is invoked on an object, this object should be in a consistent state. Otherwise, the receiver of the object can hardly do anything useful with it. Since it is the purpose of class invariants to describe that an object is in a consistent state, we would want our class invariants to hold in these states.

Definition 1. *A class invariant I_C is* valid *if it holds for all existing objects of type C during the following points of program execution:*

- *at the beginning of any method execution, except for a new object at the beginning of the execution of its constructor*
- *at the end of any method execution.*

3.2 Proof Obligations for Class Invariants

We want to design a proof system that allows us to derive validity for class invariants. For this purpose, we have to extend the system of section 2 with additional proof obligations.

3.3 Simple Case

In the simplest case it suffices to prove that the invariant of an object holds at exit of any method that changes the object, including any constructor of that object:

1. For any constructor $c()$ of class C prove $\{true\}body_c\{I_C\}$
2. for any plain method $m()$, prove $\{pre_m \wedge I\}body_m\{post_m \wedge I_C\}$

This proof system may be unsound, however, when there is *re-entrance*. Re-entrance occurs when a chain of method calls returns to an object earlier in the chain. For example, consider an object α that executes a method m. Halfway, m calls a method on object β and this methods calls back on α by method n. When n starts, m is not finished, so the proof obligation above does not guarantee that α's invariant holds. Re-entrance is in particular possible in situations where call-back mechanisms are exploited.

To accommodate this, we have to require that the invariant holds just before any method call in the body of a method. This leads to the following structure.

- **assumption** At the beginning of the method body and after every method call, *assume* that the invariant holds.
- **obligation** At the end of the method body and before every method call, *prove* that the invariant holds.

Note that we can strengthen the assumption with invariants of any other object, for instance, objects that are handed to a method via parameters can be assumed consistent.

This is not enough, however. Suppose object o refers to object p in its invariant. Then, changing p may invalidate the invariant of o. This suggests the following definition. This is the so-called *forward-backward* problem, described by Meyer. In this example, two objects keep references to one another. When this is expressed in the invariant of one of these objects, this invariant could be violated by changing the other object. This notion is captured in the following definition.

Definition 2. *When object reference o of type D occurs in invariant I_C, we say that objects of class C are* vulnerable *to (objects of) class D. o is called the* vulnerability reference.

Given an execution state, an object α is semantically vulnerable *to an object β if a change to β can invalidate the invariant of α.*

The idea now is to strengthen the proof obligation above with obligations to prove the invariants of all objects that are vulnerable to the current class. For this purpose, we need a reference to the vulnerable object. The next section deals with this problem.

3.4 Referencing Vulnerable Objects

Suppose we have a linked list of objects of class C. Every object has a field n that references the next object in the list (possibly `null`), and an integer field x. If we want to express that the list is strictly decreasing, we need as invariant

$$I_C : x \neq \texttt{null} \rightarrow x > n.x$$

Then, each object in the list is vulnerable to the next one. For instance, a method m that increases x would maintain the invariant of the current object, but would

invalidate the invariant of the previous object in the list. So we need a proof obligation that can talk about the previous object in the list, although in general objects don't have such a reference, as this example shows. This problem is not restricted to this example. In many cases, there are one-way references and when such a reference occurs in the invariant, the referenced object cannot talk about the object that is vulnerable to its changes.

For this purpose, we introduce *logical variables* (sometimes called freeze variables or specification variables, [4]) in the method body that refer to the vulnerable objects. For every method in class B and expression o of type B occurring in the invariant of class A, the assertion at the start of the method body may be strengthened by expressions of the form

$$X.o = \texttt{this} \vee X = \texttt{null}$$

where X is of type A.

3.5 Proof System

Gathering these ideas together, we come to the following scheme of proof obligations.

Let M be a method of class C, annotated as in section 2.4, let $r_1 \ldots r_k$ be the vulnerability references of C, then do the following[2] for every $1 \leq i \leq n$.

1. (pre-condition) choose object references $p_1, \ldots, p_m$, and prove
   $pre \wedge p_1.I \wedge \ldots \wedge p_m.I \wedge (X_1.r_1 = \texttt{this} \vee X_1 = \texttt{null}) \wedge \ldots \wedge (X_k.r_k = \texttt{this} \vee X_k = \texttt{null}) \Rightarrow Q_1$
   when M is a constructor, none of the p_i may equal `this`;
2. (local code segment P_i)prove $\{Q_i\}P_i\{R_i\}$;
3. (pre condition method call $o_i.M_i$) prove $R_i \Rightarrow o_i.pre_{M_i}$;
4. (local invariant) $R_i \Rightarrow I$
5. (vulnerable invariants) for every vulnerability reference r_j prove $X_j \neq \texttt{null} \wedge R_i \Rightarrow X_j : I$;
6. (post-condition method call) choose object references $q_1, \ldots, q_m$ and prove $o_i.post_{M_i} \wedge q_1.I \wedge \ldots \wedge q_m.I \Rightarrow Q_i$;
7. (post-condition) $R_n \Rightarrow post$.

Remarks

ad 1 These invariants are free to choose. Any object reference that is in scope can be used, referring via member fields of the current object or via parameters of M. Object references that are not in scope are not forbidden, but are useless in proofs.

ad 2 For this, we rely on the underlying proof system. Note that P_i does not contain any method calls.

[2] When method M is inherited from a superclass B, these proof obligations have to be redone in case the invariant of C is stronger than the invariant of B.

ad 3 Again, two approaches can be followed here. When it has been proved that $pre_{M^i_C} \Rightarrow pre_{M^i_D}$ for every class $D <: C$, it suffices to prove $o_i : pre_{M^i}$. Otherwise, some type information about o_i must be used.

ad 6 Analogous to 3, when it has been proved that $post_{M^i_D} \Rightarrow post_{M^i_C}$, then the proof obligation reduces to $o_i : post_{M_i} \wedge$ etc. Otherwise, type information about o_i must be used.

4 Example

This example shows how to deal with vulnerability. Consider two classes A and B, where objects in A hold a reference to those in B, but not the other way round. Changing the value of an object in class B may invalidate the invariant of an object in class A. Hence, objetcs in class A are vulnerable to B.

```
{I_A: x>ref.y}
class A begin
   B ref;
   int x;
end

{I_B: true}
class B begin
   int y;

   void dec() begin
      {Q_1}
      y:=y-1;
      {R_1}
   end;

   void inc() begin
      {Q_2}
      y:=y+1;
      {R_2}
   end
end
```

Obviously, *dec*() leaves I_A intact, whereas *inc*() doesnot necessarily.

To prove the correctness of *dec*(), we choose

$$Q_1 : (X.ref = this \vee X = \texttt{null}) \wedge y = N \wedge X.x > X.ref.y$$

(the last conjunct is the invariant for X).

$$R_1 : X.x > N \wedge y = N - 1$$

$X.x > N$ follows from the fact that $X.x$ is immutable to this segment and $X.ref.y = \texttt{this}.y = N$. From R_1 the required $X.x > X.ref.y$ can easily be deduced.

The proof obligation that R_2 implies $X.I$ cannot be satisfied, which is of course what we want.

5 Soundness and Completeness

This section briefly sketches the proofs of soundness and completeness.

To define these notions, we assume that there is always a main program that starts out with no objects allocated. As far as verification is concerned, we consider this main program as a method body with pre- and postconditions equivalent to *true*.

Following [11], we define an execution of the program as a maximal, in our case also terminating, sequence of transitions between program states.

Definition 3. *An* execution e *of program* P *is a terminating transition sequence* $\sigma_0 \xrightarrow{l_1} \ldots \xrightarrow{l_n} \sigma_n$ *where the* σ_i *denote program states and each label* l_i *either denotes*

1. *a local tranisition, i.e., a sequence of local steps, corresponding to a local code segment, not involving method calls or object creation; which local transitions are allowed is defined by the semantics of the programming language that is used and is not of interest here;*
2. *or a method call* $o.m()$*;*
3. *or a return transition, corresponding to the termination of a method; which method is terminated is determined by the balance of calls and returns in the previous part of the execution sequence.*

Since we are only studying partial correctness, we can restrict ourselves to terminating executions. This means in particular that dereferencing of `null`-referencing will never occur, since this would lead to abortion.

We assume that the proof system is sound for local transitions, i.e., whenever $\{Q\}P\{R\}$ has been proven for a local code segment P, it is true, i.e., every terminating execution of P starting in a state satisfying Q will end in a state satisfying R.

In the following, the phrase "all class invariants hold" (in a state σ or during an execution e), means that the invariants of all allocated objects (in σ or in the states of e) evaluate to true.

Theorem 1 (Soundness). *Let a program be given with all proof obligations satisfied and an execution sequence with all class invariants holding in* σ_0*. Then all class invariants hold for all objects at all states* σ_i *in the transition sequence.*

Proof Let a program and an execution sequence $e = \sigma_0 \xrightarrow{l_1} \ldots \xrightarrow{l_n} \sigma_n$ be given as in the theorem. Note that all states σ_i correspond to points in the code that are annotated in the correctness proof. By induction to i, we prove that all class invariants hold during e and furthermore that the assertions from the correctness proof hold in the corresponding states. For $i = 0$ it holds by assumption. For $i > 0$, there are three cases.

1. l_i denotes a local transition in class C for code segment P_j. By induction, $[\![Q_j]\!]\sigma_{i-1}$ and by soundness of the proof system for local transitions and p.o. (proof obligation) 2, $[\![R_j]\!]\sigma_i$. Now consider an object α of class D. If D is not vulnerable to C, it is obvious that $[\![\alpha.I_D]\!]\sigma_{i-1} \Leftrightarrow [\![\alpha.I_D]\!]\sigma_i$. If $[\![\alpha]\!]\sigma_{i-1} = [\![\texttt{this}]\!]\sigma_{i-1}$, and hence $[\![\alpha]\!]\sigma_i = [\![\texttt{this}]\!]\sigma_i$, then $[\![\alpha.I_C]\!]\sigma_i$ follows from $[\![R_j]\!]\sigma_i$ and p.o. 4. If D is vulnerable to C via r, we know by p.o. 5 that $R \Rightarrow (X \neq \texttt{null} \Rightarrow X\!:\!I)$. Since X is a free variable here and we have soundness for local transitions, this must hold for any valuation of X, in particular for $X = \alpha$. Since α refers to an actual object, it can't be `null` and, knowing that $[\![R_j]\!]\sigma_i$, $[\![\alpha.I]\!]\sigma_i$ must be true.
2. l_i denotes a method call $o_j.M_j$. When this is not a constructor, no object has changed state between σ_{i-1} and σ_i and hence all invariants are maintained. When the method call is a constructor call, a new object has been created in σ_i. For this object, however, the invariant is not required at σ_i. Furthermore, by induction $[\![R_j]\!]\sigma_{i-1}$ and then by p.o. 3, $[\![o_i.pre_{M_j}]\!]\sigma_{i-1}$. By the semantics of the method call, we then know that $[\![pre]\!]\sigma_i$ and because all invariants are maintained, $[\![p_k.I]\!]$ holds for arbitrary p_k. When we choose a valuation for the X_k that satisfies $X_k.r_k = \texttt{this} \vee X_k = \texttt{null}$ (which is always possible), we know that $[\![Q_1]\!]\sigma_i$ because of p.o. 1.
3. l_i denotes a return transition. Then no objects have changed their state and hence all invariants are maintained. The proof that $[\![Q_j]\!]\sigma_i$ is analogous to the previous point, now using p.o. 6 and p.o. 7.

end of proof

In the following definition, an annotation of the program refers to the annotation in section 3.5, in which postconditions may be strengthened. Why this is, is explained below.

Theorem 2 (Completeness). *Let a program be given of which all executions satisfy at method calls the corresponding preconditions and at both calls and returns all class invariants. Then there exists an annotation of the program and the classes such that all proof obligations are fulfilled.*

We have to allow that postconditions are strengthened, because otherwise it may be impossible to fulfill proof obligation 6.

Now we can prove completeness if we have a complete proof system without class invariants. Then we can strengthen the postconditions in such a way that they imply the necessary class invariants.This must be possible, since the class invariants are holding in the corresponding states, by assumption.

This proof depends on the existence of a complete proof system for the chosen object oriented programming language. In the literature various proof systems can be found ([2,5]), although they differ somewhat from our approach.

6 Conclusion

The above approach extends the practical applicability of Object Oriented verification using pre and post conditions and invariants; its soundness and completeness is argued. One could perhaps view the notion of completeness as not

fully satisfactory - we are studying other notions that do not depend on changing contracts, viz. strengthening postconditions, taking into account the extension with subclasses.

Future work considers furthering the practical use of the approach through establishing simplifying (preferably syntactical) restrictions. Correctness of the resulting method could then be argued on the basis of the framework presented here.

References

1. M. Abadi and K.R.M. Leino, *A Logic of Object-Oriented Programs*, in TAPSOFT '97, LNCS 1214, Springer, 1997.
2. P.H.M. America and J.J.M.M. Rutten, *A Parallel Object-Oriented Language: Design and semantic foundations*, PhD thesis, Free University of Amsterdam, 1989.
3. K. Arnold and J. Gosling, *The Java programming language, 2nd ed.*, Addison-Wesley, 1997.
4. K.R. Apt and E.-R. Olderog, *Verification of sequential and concurrent programs*, Springer-Verlag, 1991.
5. F.S. de Boer, *Reasoning about dynamically evolving process structures: A proof theory for the parallel object-oriented language POOL*, PhD thesis, Free University of Amsterdam, 1991.
6. U. Hensel, M. Huisman, B. Jacobs, and H. Tews, *Reasoning about Classes in Object-Oriented Languages: Logical Models Tools*, in ESOP at ETAPS 1998, Springer-Verlag, 1998.
7. C.A.R. Hoare, *An axiomatic basis for computer programming*, Communications of the ACM, 12, pp. 576–583, 1969.
8. H.B.M. Jonkers, *Upgrading the pre- and postcondition technique.* In VDM '91: Formal Software Development Methods, LNCS 551, Springer-Verlag, 1991.
9. K. Rustan M. Leino, *Toward Reliable Modular Programs*, Phd. Thesis, California Institute of Technology, Pasadena, 1995.
10. B. Liskov and J. Wing, *A behavioral notion of subtyping*, ACM TOPLAS , 16:6, pp. 1811-1841, 1994.
11. Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.
12. B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1988.
13. A. Poetzsch-Heffter and P. Müller, *Logical foundations for typed object-oriented languages*, in D. Gries and W.P. de Roever, editors, Programming Concepts and Methods (PROCOMET), 1998.
14. A. Poetzsch-Heffter, *Specification and verification of object-oriented programs*, Habilitation, TU Muenchen, 1997.
15. C. Szyperski, *Component software : Beyond object-oriented programming*, Addison-Wesley, 1998.
16. J.Warmer, A. Kleppe, *The Object Constraint Language*, Addison-Wesley, 1999.