On the Role of Invariants in Reasoning about Object-Oriented Languages

Joachim van den Berg, Cees-Bart Breunesse, Bart Jacobs, Erik Poll

Computing Science Institute, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen, The Netherlands
{joachim,ceesb,bart,erikpoll}@cs.kun.nl

Abstract The aim of this paper is to clarify the different roles that class invariants play in the verification of object-oriented programs, namely in method specifications as *proof obligations for method implementations* (assume the precondition and then prove the postcondition) and in specifications as *assumptions on method invocations* (prove the precondition, and then assume the postcondition), in order to prove the correctness of other methods. The standard proof obligation is that individual methods preserve the invariant of their class, as part of the method's specification. When trying to prove such an obligation one may have to be careful about the invariants that are assumed to be part of the precondition, at least if one wishes to use the same specification in other proofs. This is what we call the *conservative* approach. There is an associated more *liberal* approach which really makes a distinction between specifications as obligations and as assumptions. In the latter case the requirement to prove all invariants in the precondition is dropped. This considerably simplifies the verification work, but relies on a suitable meta-result about non-disturbance of invariants.

1 Introduction

Preconditions, postconditions and invariants are universally accepted as the basic ingredients for specification of OO programs. Invariants for classes play an important role in object-oriented languages ([11,14]) because they often express key data integrity properties, such as: this integer field is always non-negative, or this reference is never null. The idea is that invariants should always hold. But things are not so simple. Consider for example the invariant `i+j==0`. It does not hold in between the two statements `i++;j--`, even though it will hold after the composition, assuming it holds before. It is inevitable that some invariants are temporarily invalidated in a method body, but they should be re-established at some point.

Our perspective here—unlike for example [4]—is tool-assisted verification of object-oriented programs. More specifically, proving specifications for method implementations. The specifications we consider are written in the behavioural interface specification language JML, developed by Leavens *et al.* [6]. The implementations are written in Java. We have developed a special compiler, called

the LOOP tool (see [1]) which translates these JML specifications plus Java implementations into their semantics in the language of the proof tool PVS. Specifications become special predicates, which we try to prove for the (translated) implementations. In this context, we are forced to be very systematic and precise about the meaning of invariants—which is a delicate matter. The purpose of this short note is to make the main problems explicit, and to present the options that one has.

The verification of a specification for a method proceeds by stepping through the methods body via applying appropriate Hoare-like rules for JML, see [5], for each individual statement in this body. These statements may consist of method calls, like in `void m() { ... o.k(); ... }`. What we then do is *use* the specification given for the method `k`—which involves in particular showing that the precondition of this specification of `k` holds. The first point we wish to emphasise is that there are thus two different roles for method specifications in verification, namely as a proof obligation for the method implementation, and as an assumption about a method invocation. Ideally, specifications have the same meaning in these two roles, but that is possibly not the most convenient. Invariants play a key part in making such a distinction.

Method specifications in JML contain several clauses for the various modes of termination in Java, like divergence, normal termination, and exceptional termination (see [5] for details). These differences are not so relevant in this paper, and so we shall simplify this matter and use the standard triple notation from Hoare logic,

$$\{\ \mathtt{pre}\ \}\ \mathtt{m}\ \{\ \mathtt{post}\ \} \tag{1}$$

for partial correctness. Such a triple expresses that if the precondition `pre` holds in the pre-state, and if the method `m` terminates normally, then the postcondition `post` holds in the post-state. However, the triple (1) is too simple, because it does not involve the invariant $\mathtt{I_A}$ of the class `A` in which the method `m` is defined. Thus the triple (1) should be:

$$\{\ \mathtt{I_A}\ \&\&\ \mathtt{pre}\ \}\ \mathtt{m}\ \{\ \mathtt{I_A}\ \&\&\ \mathtt{post}\ \} \tag{2}$$

This is standardly taken as the meaning of a method specification, and is what should be proved for the method.

2 Invariants and inheritance

The two main requirements[1] of the behavioural approach to subtyping [10] are the following. For a subclass `B` of a class `A`,

1. The invariant $\mathtt{I_B}$ of `B` is stronger than the invariant $\mathtt{I_A}$ of `A`, *i.e.* $\mathtt{I_B} \Rightarrow \mathtt{I_A}$.
2. For each method `m` in class `A` that is overridden in the subclass `B`, the specification of `m` in `B` is stronger than the one in `A`. Usually this is expressed by the pair of implications:

$$\mathtt{pre_A} \implies \mathtt{pre_B} \qquad \text{and} \qquad \mathtt{post_B} \implies \mathtt{post_A}. \tag{3}$$

[1] Omitting constraints.

Notice that the second condition does not involve invariants. Simply adding them in the straightforward way, namely as:

$$\mathtt{I_A} \mathbin{\&\&} \mathtt{pre_A} \implies \mathtt{I_B} \mathbin{\&\&} \mathtt{pre_B} \qquad \text{and} \qquad \mathtt{I_B} \mathbin{\&\&} \mathtt{post_B} \implies \mathtt{I_A} \mathbin{\&\&} \mathtt{post_A} \tag{4}$$

is problematic, because the first implication does not hold in general—because $\mathtt{I_B}$ can be really stronger than $\mathtt{I_A}$.

The practice of actual verification [3,2] has taught us that a slightly different approach is more appropriate (and effective), in which the above implications (3) and (4) are not required. In order to explain this alternative we distinguish notationally between the version $\mathtt{m_B}$ of `m` in `B`, and the (overridden) version $\mathtt{m_A}$ in `A`. The meaning of the specification of $\mathtt{m_A}$ is as in (2) above. For $\mathtt{m_B}$ we use a conjunction of two requirements:

$$\begin{array}{l} \{\ \mathtt{I_B} \mathbin{\&\&} \mathtt{pre_B}\ \}\ \mathtt{m_B}\ \{\ \mathtt{I_B} \mathbin{\&\&} \mathtt{post_B}\ \} \\ \{\ \mathtt{I_B} \mathbin{\&\&} \mathtt{pre_A}\ \}\ \mathtt{m_B}\ \{\ \mathtt{I_B} \mathbin{\&\&} \mathtt{post_A}\ \} \end{array} \tag{5}$$

Notice that the first triple is the analogue of (2) for $\mathtt{m_B}$. The second one expresses that $\mathtt{m_B}$ should also satisfy the specification of the superclass `A`, *but with the invariant of its own class*. Indeed, this is what is often needed in practice, typically when the invariant $\mathtt{I_B}$ expresses certain safety conditions that are essential for the correct behaviour of $\mathtt{m_B}$. We shall use the second triple in (5) as interpretation of the requirement 2. of behavioural subtyping in the beginning of this section, namely that an overriding method should also satisfy the specification of the overridden method.

Several further remarks should be made at this point.

1. The formulations (5) are convenient in verification because they are in a form that can directly be used in proofs, when specifications are used as assumptions.
2. The first requirement in (5) follows from the second if we can establish the following adaptations of (4).

   $$\mathtt{I_B} \mathbin{\&\&} \mathtt{pre_A} \implies \mathtt{pre_B} \qquad \text{and} \qquad \mathtt{I_B} \mathbin{\&\&} \mathtt{post_B} \implies \mathtt{post_A}.$$

   And this is what we of course do, if possible, in order to prevent going through the method body in another lengthy proof.
3. One distinguishing feature of object-oriented languages (with inheritance) is that objects have both a static type and a dynamic (run-time) type—where the latter is a subtype of the former. This raises the question whether the invariant of an object refers to the invariant of the static or of the dynamic type of an object. This difference is relevant if Java's `instanceof` is used to get more information about an objects dynamic type. In order not to make matters more complicated than they already are, we shall ignore the difference between static and dynamic invariants and shall simply write `Inv(o)` for the invariant of an object `o`.

4. Another question is whether late binding should be used to interpret the (pure, side-effect free) methods that may occur in invariants. For example, in

```
class A {
      int i;
      //@ invariant i > min();

      //@ ensures \result >= 0;
      /*@ pure */ int min() { .. }
}

class B extends A {
      //@ ensures \result >= 10;
      /*@ pure */ int min() { .. }
}
```

   do we know, for an object `b` of class `B`, that `b.i > 0` or—by late binding—`b.i > 10` ?

3 Invariants for objects (other than `this`)

Besides the invariant of the class in which the method is defined (the invariant of `this`, as it is sometimes called), the correctness of a method's implementation may also rely on invariants of other objects used in the method body, such as parameters and (possibly static) fields. Therefore, these invariants are also important to prove the correctness of the implementation.

Like the class invariant (of `this`) that is assumed to hold in the pre-state, in most cases also the invariants of the parameters and relevant fields are needed in this state. The proof obligation for a method `m` with parameters $\overrightarrow{a}$ and defined in class `A` will then become

$$\{\ \texttt{Inv(this) \&\& Inv(}\overrightarrow{a}\texttt{) \&\& Inv(}\overrightarrow{f}\texttt{) \&\& pre}\ \}\ \texttt{m(}\overrightarrow{a}\texttt{)}\ \{\ \texttt{Inv(this) \&\& post}\ \},$$

where `Inv(this)` is what we have written as $\texttt{I}_{\texttt{A}}$ before (with `A` the class of `m`), `Inv(`$\overrightarrow{a}$`)` are the invariants of the reference parameters in $\overrightarrow{a}$, and `Inv(`$\overrightarrow{f}$`)` are the invariants of all the relevant reference fields. How it is determined which objects are relevant is not so important at this stage.

It may also be the case that a method returns a reference to an object[2]. This result object should also satisfy its invariant. Therefore, the proof obligation is further strengthened to:

$$\begin{array}{l}\{\ \texttt{Inv(this) \&\& Inv(}\overrightarrow{a}\texttt{) \&\& Inv(}\overrightarrow{f}\texttt{) \&\& pre}\ \}\\ \quad\ \texttt{m(}\overrightarrow{a}\texttt{)}\\ \{\ \texttt{Inv(this) \&\& Inv(\textbackslash result) \&\& post}\ \},\end{array}$$

[2] Also, the method may produce an exception object, which should also satisfy its invariant. But this case does not occur in our simplified Hoare logic dealing only with normal termination.

where `Inv(\result)` denotes the invariant for the return value.

In an object-oriented context, modular verification ([7,12]) is important, because one wishes correctness results to be robust with respect to addition of subclasses. Consider the following example in which a method `m` has a reference parameter `o` of class `A`:

```
void m (A o) { o.i=1; o.k(); }
```

One would like the specification of `m` to be correct for all possible future subclasses of class `A`—objects of which may be passed as actual parameter. In order to achieve such correctness, it should be assured that each subclass of `A` behaves as `A`. The assignment `o.i=1`, however, may cause trouble: it might break a possibly stronger invariant of a subclass of `A`.

In general terms, assignments may disturb the invariants of other objects via exposure of the state of an object, *e.g.* via access to public fields or aliasing. In order to control this, a meta-result about non-disturbance of other invariants is needed. Müller and Poetzsch-Heffter [13,12] propose to use suitable "universes" of objects that put restrictions on leaking references to the outside world. This will be needed to make our verifications modular.

4 Invariants in verification

It is common in the literature on object-oriented specification and verification to assume invariants, whenever needed. Explicitly, in the words of Liskov & Wing [10, p.13]: "We omit adding the invariant, because if it is needed in doing a proof it can always be assumed, since it is known to be true for all objects of its type." Such a hand-waving approach is not possible in a formalised semantics for theorem proving, and must be described explicitly. This will be done by explicitly distinguishing between proving and using method specifications.

For expository reasons we first consider some naive rules for invariants, mainly to illustrate the subtleties involved.

4.1 The black box approach

In an ideal world, every object would be a black box, that only interacts with its environment by method invocations. In such a setting, every object would take care of maintaining its own invariant only. The proof obligation for a method implementation would be as expected:

```
{Inv(callee) && pre}
  m()
{Inv(callee) && post}
```

At first sight, one might expect that users of a method may make the following assumption for method invocations:

```
{pre}
  callee.m()
{post}
```

Given that the object `callee` takes care of maintaining its own invariant, the caller can assume that the invariant of `callee` will hold.

However, the rule above is not correct, because the caller may have temporarily invalidated its invariant at the moment of the invocation `callee.m`. This is a problem because of possible *call-backs*: the execution of `callee.m()` may lead to other method invocations, including method invocations on the object `caller`, in which case execution of a method of `caller` would start in a state in which its invariant does not hold. The simplest scenario where this problem occurs is the case that `caller` and `callee` are the same object, *i.e.* if `callee.m()` is an invocation of the objects own methods. To prevent such problems, the `caller` should ensure that its own invariant holds whenever it invokes other methods:

$$\begin{array}{l}\{\texttt{Inv(caller) \&\& pre}\}\\ \quad \texttt{callee.m()}\\ \{\texttt{Inv(caller) \&\& post}\}\end{array}$$

This may be seen as formalisation of the (standard) solution for call-backs, see for example [14, Section 5.8]: "A simple solution would be to require all invariants of an object to be established before calling any method."

The big problem with this approach is that the underlying assumption—that objects are black boxes—is not true for typical object-oriented languages such as Java. For instance, invariants can be disturbed by assignments to public fields. More importantly, the invariant of an object typically depends not just on the states of that object (*i.e.* its fields), but often also depends on the states of other objects (in its "universe").

Another problem with this approach is that, in spite of the drastic simplifying assumption, insisting that the invariant is maintained whenever another method is invoked may be too strong a requirement in practice. As noted for instance in [14, Section 5.8], (re)establishing an invariant might require a sequence of method invocations. One place where this requirement is often unworkably strong is in constructors; typically the invariant is not established until the end of a constructor body, which means that we cannot invoke any methods (or super constructors) in constructor bodies [9].

4.2 The white box approach

Diametrically opposed to the black box approach discussed above, where every object is given the responsibility of maintaining only its own invariant, is what we call the white box approach: here every individual object is given the responsibility of maintaining all the invariants of all existing objects.

For this approach the proof obligation for a method implementation would be

$$\begin{array}{l}\{\forall i.\ \texttt{Inv(o}_i\texttt{) \&\& pre}\}\\ \quad \texttt{m()}\\ \{\forall i.\ \texttt{Inv(o}_i\texttt{) \&\& post}\}\end{array}$$

and the assumption on method invocations would be

$$\begin{array}{l}\{\forall i.\ \texttt{Inv(o}_i\texttt{) \&\& pre}\}\\ \quad \texttt{callee.m()}\\ \{\forall i.\ \texttt{Inv(o}_i\texttt{) \&\& pre}\}\end{array}$$

In this approach we know that at all visible states—all routine borders—all invariants of all objects hold. This approach has the advantage of being sound, as the proof obligation for method implementation and the assumption on method invocation are identical. However, the problem with this approach is that the proof obligation for method implementations is unworkably strong. In general there is no way of knowing whether a given assignment to a field does not invalidate the invariant of some other object. Additionally, we also have the problem mentioned in the last paragraph of the Subsection 4.1.

4.3 The liberal and conservative approach

In light of the fundamental problems with the two approaches above, we now consider two more pragmatic approaches to invariants, see Figure 1, called the liberal and conservative approach. In the conservative approach there is no distinction between obligation and assumption—although we use a slightly different formulation to emphasise that there are two objects involved when a specification is used as assumption. In the liberal approach one adds all invariants to the precondition in a proof obligation, but one does not need to establish these invariants when the specification is used as an assumption. The justification for this omission is like in Liskov & Wing [10], as cited above. More formally, it should be justified by a meta-result about non-disturbance of invariants, in the universes setting of Poetzsch-Heffter & Müller [13,12]. This is needed for the soundness of the liberal approach. But as we saw towards the end of Section 3, the meta-result is needed anyway to be able to work modularly.

	liberal	**conservative**

specification as obligation	$\{\forall i.\ \texttt{Inv(}o_i\texttt{) \&\& pre}\}$
	`m()`
	`{Inv(callee) && post}`
		{∀relevant i. `Inv(`o_i`) && pre`}
		`m()`
		`{Inv(callee) && post}`

specification as assumption	`{Inv(caller) &&`
	`Inv(callee) && pre}`
	`callee.m()`
	`{Inv(caller) &&`
	`Inv(callee) && post}`
		{∀relevant i. `Inv(`o_i`) && pre`}
		`callee.m()`
		`{Inv(callee) && post}`

Figure1. The proof obligations and assumptions for method specifications in the liberal and the conservative approach.

Some motivations for the rules in Figure 1:

- The reason for including `Inv(caller)` in the precondition of the liberal assumption rule is to allow for call-backs, as discussed in Subsection 4.1. In the conservative assumption rule, `Inv(caller)` should be included as one of the relevant objects if there is a call-back to it.
- The reason for including `Inv(caller)` in the postcondition of the liberal assumption rule is that it is needed to prove that a method implementation containing `callee.m()` preserves the caller's invariant. However, one could also rely on the modifiable clause (*i.e.* frame property) of the called method `m` to prove this; this is what has to be done in the conservative approach.

If one wishes to use a specification in the conservative approach one needs to prove explicitly that the invariants added to the precondition hold. Adding the invariants of *all* objects makes proofs unnecessarily difficult—and often impossible, when certain, irrelevant, invariants do not hold. This is the reason for restricting ourselves to what we call the invariants of all relevant objects. Still this puts a considerable burden on the verifier: in our experience this requirement gives a lot of work, which is in essence unnecessary. But the great advantage of this conservative approach is of course its soundness.

By having a weaker specification as obligation than as assumption in the liberal approach it is not hard to introduce logical inconsistencies in the back-end theorem prover, namely in cases where an invariant of an other object (than `this`) is necessary for a certain result value. Consider for example:

```
class A {
  int i;
  //@ invariant i > 0;

  //@ requires a != null;
  //@ ensures \result == true;
  boolean m(A a) { return a.i > 0; }

  //@ requires a != null && b != null;
  //@ ensures \result == true;
  boolean k(A a, A b) { b.i = 0; return a.m(b); }
}
```

Notice that `k` does not meet its specification. But it can be proven "correct" with the liberal approach: the specification of `m` as assumption does not require that the invariant for the parameters hold.

In the universe approach [13] certain constraints will have to be imposed, in order to prevent that the integer field `i` can become non-positive, *e.g.* by making it private or read-only. This disables invariant-disturbing assignments to `i`.

As mentioned, in the conservative approach only invariants of relevant objects are added to the precondition, whereas all invariants are added when proving a specification in the liberal approach. During such a "liberal" proof one quickly

restricts the invariants added to the precondition to only the relevant objects, because otherwise one pushes too strong a formula through the method body (via the rules for Hoare logic).

5 LOOP project

As stated in the introduction, the LOOP compiler translates Java classes plus JML annotations into PVS. The translation covers almost all of sequential Java, and this part is reasonably stable. The translation of JML is still under construction, but already covers a core part of JML: class invariants and constraints, method specifications including modifiable clauses, but, for instance, not yet model variables. The JML translation is being used for several case studies (notably for JavaCard), and is optimised on the basis of the resulting experiences.

Originally, the approach we used was to explicitly include invariants of objects other than `this` in pre- and postconditions, which is possible in JML using the `\invariant_for` keyword. For example, the method `m` in the example above could be specified as follows

```
//@ requires a != null && \invariant_for(a);
//@ ensures \result == true;
boolean m(A a) { return a.i > 0; }
```

to make it explicit that `m` relies on the invariant of its argument. The problem with this approach is that it quickly becomes cumbersome to have to explicitly include all these invariants in pre- and postconditions.

We then considered the conservative approach, mainly because this seemed to be the safest. It has given rise to several difficulties.

- How to determine which objects are relevant? Adding invariants for reference parameters is not problematic, but finding all other relevant objects in a method body is beyond static analysis. This is a problem in generating the semantics of a specification. ESC/Java uses some heuristic to choose the set of relevant objects [8, Sect. 2.4.1]. One way to tackle this problem might be to include a model variable `relevantObjects`,

  ```
  //@  public model JMLObjectSet relevantObjects;
  ```

  for every object, that keeps track of the set of relevant objects, *i.e.* `o.relevantObjects` is the set of objects on whose invariants the methods of the object `o` rely.
- Proving the invariants of all relevant objects when a method specification is used is very time consuming, and does not yield very much in terms of confidence because these invariants typically hold anyway.

On the basis of these experiences we are now moving to the liberal approach[3], also because there are now concrete plans to extend JML with the universe type system of Poetzsch-Heffter & Müller. The liberal approach makes verification easier, but it may lead to inconsistencies because the required meta-result about non-disturbance of invariants is not formalised in PVS. This possibly unsound approach goes very much against the very idea of formal verification, but seems to be the most pragmatic: we have assumptions which are strong enough for verifying non-trivial program properties, but which should not be abused.

6 Conclusion

Our work on tool-assisted verification of object-oriented programs forces us to be very explicit and precise about the meaning of all the constructs involved. This short note focuses on the role of invariants in this setting, and tries to clarify several issues that occur in various places in the literature, but are still confusing when it comes down to detailed formalisation.

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 ans Algorithms for the Construction and Analysis of Software*, number 2031 in Lect. Notes Comp. Sci., pages 299–312. Springer, Berlin, 2001.
2. J. van den Berg, B. Jacobs, and E. Poll. Formal Specification and Verification of JavaCard's Application Identifier Class. In I. Attali and T. Jensen, editors, *Proceeding of the first JavaCard Workshop (JCW'2000)*, Lect. Notes Comp. Sci. Springer, Berlin, 2001. To appear.
3. M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's Vector class. Techn. Rep. CSI-R0007, Comput. Sci. Inst., Univ. of Nijmegen. To appear in *Software Tools for Technology Transfer*, 2001.
4. K. Huizing, R. Kuiper, and SOOP. Verification of object oriented programs using class invariants. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering*, number 1783 in Lect. Notes Comp. Sci., pages 208–221. Springer, Berlin, 2000.
5. B. Jacobs and E. Poll. A logic for the Java Modeling Language JML. In H. Hussmann, editor, *Fundamental Approaches to Software Engineering (FASE)*, number 2029 in Lect. Notes Comp. Sci., pages 284–299. Springer, Berlin, 2001.
6. G.T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Techn. Rep. 98-06, Dep. of Comp. Sci., Iowa State Univ. (`http://www.cs.iastate.edu/~leavens/JML.html`), 1998, revised April 2001.

[3] We are also considering an equally unsound variation of the liberal approach which simply says (as an assumption in PVS) that all objects different from the caller (`this`) satisfy their invariant. This is even easier to use, because it avoids that the invariant assumptions at the beginning of proofs have to be carried along through the method body until the point where they are needed.

7. K.R.M. Leino. *Toward Reliable Modular Programs.* PhD thesis, California Inst. of Techn., 1995.
8. K.R.M. Leino, G. Nelson, and J.B. Saxe. Esc/java user's manual. Technical Report (2000/002), Compaq SRC, 2000.
9. K.R.M. Leino and R. Stata. Checking object invariants. Technical Report 97/007, Digital SRC, 1997.
10. B. Liskov and J. Wing. A behavioral notion of subtyping. *TOPLAS*, 16(6):1811–1841, November 1994.
11. B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, second edition, 1997.
12. P. Müller. *Modular Specification and Verification of Object-Oriented Programs.* PhD thesis, FernUniversität Hagen, 2001. Available at `http://www.informatik.fernuni-hagen.de/pi5/publications.html`.
13. P. Müller and A. Poetzsch-Heffter. Universes: a type system for alias and dependency control. Tech. Rep. 279-1/2001, Fernuniversität Hagen, 2001.
14. C. Szyperski. *Component Software.* Addison-Wesley, 1998.