# The Need for Flexible Object Invariants

Alexander J. Summers
Imperial College London
alexander.j.summers
@imperial.ac.uk

Sophia Drossopoulou
Imperial College London
s.drossopoulou
@imperial.ac.uk

Peter Müller
ETH Zürich
peter.mueller
@inf.ethz.ch

## ABSTRACT

Specification and verification of object oriented programs usually features in some capacity the concept of an *object invariant*, used to describe the consistent states of an object. Unavoidably, an object's invariant will be broken at some points in its lifetime, and as a result, *invariant protocols* have been suggested, which prescribe the times at which object invariants may be broken, and the points at which they have to be re-established.

The fact that currently available invariant protocols do not handle well some known examples, together with the fact that object invariants and invariant protocols can largely be encoded through methods' pre- and post- conditions has recently raised the question of whether they still have a role to play, or should be replaced by more explicit pre- and post-conditions for methods.

In this paper we argue that invariant protocols express programmers' intuitions, lead to better design, allow more succinct specifications and proofs, and allow the expression of properties which involve many objects in a localised manner. In particular, the resulting verification conditions can be made simpler and more modular through the use of invariant-based reasoning.

We also argue that even though encoding invariant protocols through methods' pre- and post-conditions is possible, such an encoding loses important information, and as a result makes specifications less explicit and program evolution (whereby the program evolves after the encoding has taken place) more error-prone. Finally, we show that such encodings often cannot express properties over inaccessible objects, whereas an appropriate invariant protocol can handle them simply.

## 1. INTRODUCTION

### Object Invariants

The coupling of functionality and state is fundamental to the object-oriented programming paradigm. An object's state may change during program execution, and as a result the object's behaviour may change. Objects behave properly if they are in a *consistent state*, and are *implicitly* expected by programmers to be in such a consistent state at most times of execution.

The pioneering work of Bertrand Meyer [10] suggested that the specification and verification of object-oriented programs should be based on the concept of *object invariant*, which describes the consistent states of an object. Object invariants provide a means of specification at the natural level of the object itself, typically in terms of its fields. Method pre- and post-conditions complement these object-level specifications by describing method behaviours in the traditional procedural programming style.

### Mixed Logics

An object's invariant is usually expected to hold immediately before and after a method call to that object, but may need to be temporarily broken. Typically, an object's invariant may be broken while it is in some sense active (e.g., executing a method). We call *mixed logic* a logic which implicitly enforces that certain objects' invariants hold at particular points in an object's lifetime, and incorporates these facts into verification proofs. In a mixed logic, classes typically have explicitly declared invariants, and methods have an implicit obligation to preserve relevant object invariants. For example, using object invariants, an ordered list could be specified as follows:

```
class OList{   // specified  in  a  mixed  logic
  OList next;
  Item contents;

  // invariant : Ordered(this)

  // pre:   none
  // post:  this . contains( it )
  void insert (Item it ) {  ...  }

   ...  }
```

### Pure Logics

On the other hand, if we conjoin the intended invariants to the pre- and post-states of all methods of the particular class, we explicitly require that an object's invariant holds immediately before and after a method call to that object. We call *pure logic* a logic which does not support invariants and reasons only based on method specifications. Thus, an ordered list would be specified in a pure logic as follows:

```
class OList{   // specified  in a pure  logic
  OList next;
  Item contents;

  // pre:   Ordered(this)
  // post:  Contains(this, it) ∧ Ordered(this)
  void  insert (Item  it )   {  ...  }

   ...  }
```

## *Invariant Protocols*

Invariants often describe not only constraints on the fields of a single object, but relationships between the states of collaborating objects. However, if an invariant depends on the state of some other objects as well as its own state, then the invariant may be broken while *other* objects are active. In order to handle this complexity, an *invariant protocol* is required, which determines at which points in a program the invariant of an object may be assumed to hold, and at which points it has to be established.

Müller et. al. suggested an invariant protocol, whereby an object *owns* all objects on whose state its invariant depends, the ownership hierarchy is a tree, and whereby the invariant of an object can only be affected by its owner and its owned objects [12]. This protocol has subsequently been refined to deal with callbacks, visibility, dependency across objects with a common owner, and subclasses [12, 6], dynamic ownership [1], or ownership types [9].

However, there are cases where the invariant dependency is not a tree, e.g. the Subject and Observer pattern, and where state modifying methods do not navigate the tree from the top, e.g. the Composite. The Composite pattern describes a tree data structure, which maintains some property which relates each node to the nodes in its subtrees.[1]

We discuss here a variant of the Composite pattern, in which an integer value is stored at each node, and the property to maintain is that the value at each node is the maximum of the values stored in subtrees. To simplify matters, we consider the addition of new children to a node, but not the removal of children nodes.

```
class Node {
 int  initVal ;
 int  value;
 Node parent;
 Set<Node> children;

 Node(int  initial  ) {
    value  =  initVal  =  initial  ;
    parent  = null;
    children   = new Set<Node>;
 }

 void add(Node o) {   ...    }
}
```

We write the desired property as:

$$\mathit{node}(o) \Leftrightarrow \mathsf{value} = \mathit{max}(\{\mathsf{initVal}\} \cup \{\mathsf{o}'.\mathsf{value} | \mathsf{o}' \in \mathsf{o.children}\})$$

[1]This pattern appears in various applications, e.g., in graphics, when resizing a subcomponent could result in resizing all enclosing components), or in file systems where increasing the size of a file implicitly increases the sizes of the enclosing directories.

From our perspective, the most interesting features of the pattern are that it is designed to cope with modifications being made at arbitrary points in the data structure, and not necessarily via the "parent" nodes, and that the invariant *node* depends on more than one object. To deal with these problems it has been suggested that an object should *notify* the objects whose invariant is affected by any modification to its own state [3, 11].

## *Object invariants - the end of the road?*

On the other hand, conjoining invariants pre- and post-conditions in method specifications can encode most of the invariant protocols, and allows for more flexibility: if it is intended that for some methods (e.g., helper methods) an invariant need not hold, then the predicate expressing the invariant need not be conjoined to the specification of that method. More generally, properties across several objects can be decomposed into weaker properties (via other predicate definitions), and so it is easy to express that an "invariant" partially holds. For these reasons, it has been suggested by Parkinson [13] that the object invariant is, as a fundamental principle for object-oriented verification, superfluous.

## *The need for flexible object invariants*

In this paper we argue that the object invariant can play an important role in the design, specification and verification of object-oriented programs, even though more work in mixed logics is needed.

At a philosophical level, we believe that the concept of object invariant is natural for object-oriented programmers, because it expresses implicit information. Namely, in contrast to the functional programming paradigm, where all relevant information is made explicit through parameter passing, in the object-oriented paradigm a large amount of the relevant information is implicit in the state of the objects. When reasoning about OO programs, explicit information is expressed through methods' pre- and post-conditions, while implicit information is expressed through object invariants.

Another philosophical point is that an object invariant can in some cases be understood to be part of the *definition* of an intended type in a program. For example, having defined a class to represent integers, a natural way to define natural numbers is to write a subclass with the additional invariant that the value stored is non-negative. Like types, such an invariant provides implicit guarantees which hold most of the time, and which may require special treatment in some cases (as e.g., for assignment to arrays in Java). In such cases, we believe it to be more natural to assume the intended property by default, and to be aware of the exceptions, rather than to assert explicitly the information when we require it.

In this paper, we make three claims for the usefulness (and in some cases, necessity) of object invariants. We substantiate these claims by comparing relevant questions of design, specification and verification in the contexts of pure logics and mixed logics. Our three points are general, and at a high-level: we do not consider any particular pure logic or mixed logic for our arguments.

**Invariant-based reasoning aids and improves software design** (Section 2). We believe that thinking in terms of invariants can help to come up with good software designs, and that a design can be made more robust by following an invariant-based philosophy. Furthermore, we argue that if invariants are encoded into pre- and post-conditions, essential design information is lost, and its absence may lead to specification/design errors at the level of programmer specifications, and less practical specifications and proof obligations for automatic verification.

**Software design can be exploited by invariant-based reasoning** (Section 3). We believe that for a verification approach to be both natural to a programmer, and scalable to large programming projects, it is essential that the principles guiding the design of the code are reflected and exploited in specifying and verifying the code. Therefore, if code is designed with invariants in mind, then it is natural and useful to reflect this in the corresponding reasoning. In particular, invariants allow a verification analogue to the concept of delegation of responsibility, which is a fundamental aspect of object-oriented design philosophy.

**Object invariants allow local reasoning about global properties** (Section 4). We argue that invariant protocols express naturally the code patterns where objects notify other objects whose properties they may have broken, and that object invariants allow the expression of these specification requirements in a local way. Furthermore, we argue that object invariants are the most natural way of maintaining properties which depend on objects which are not accessible via field accesses, and can permit a simpler and more practical verification of such properties.

## 2. INVARIANT-BASED REASONING AIDS AND IMPROVES SOFTWARE DESIGN

We believe that reasoning using invariants can help to come up with good software designs, and that a design can be made more robust by following an invariant-based philosophy. Furthermore, we argue that if invariants are encoded into pre- and post-conditions, essential design information is lost, and its absence may lead to specification/design errors. We illustrate these ideas in this section with three particular points of interest.

### 2.1 Invariants impose consistency on future code

Invariants do not express constraints only on a fixed set of methods which currently exist (or are known about) in a program, but also on future extensions of a class, both when upgrading the definition of an existing class, and when adding new subclasses. If one chooses to map invariants down to the pre- and post-conditions of existing methods, the restriction on future code is lost.

Consider the ordered list (OList) example from the introduction. We consider upgrading the class definition by adding a method set allowing the value stored at a specific index in the list to be updated (standard for the usual lists in Java). Suppose that the method does not preserve *Ordered*-ness (indeed, there is no natural implementation of the intended operation which could guarantee to do so). In the mixed logic approach such a method could not usually be added to the class, since it would not adhere to the implicit specification of preserving the invariant. In the pure logic approach the method set could be specified and verified, since we are at liberty to omit the *Ordered* predicate from the specification of the method.

```
// pre: this.length ≤ i
// post: ...
void set(Item it, int i) { ... }
```

However, we argue that method set philosophically should not belong to such a class, since it does not make sense for ordered lists. The intended design of OList is not enforced in the pure logic, but in the mixed logic, the invariant can impose structure on all future versions of the class. Furthermore, the design intention is made explicit to the programmer in the declaration of the invariant.

One could argue that this potential anomaly is easily detected: after calling rogue methods such as set which do not preserve the *Ordered* property, the receiver object will become largely unusable, since it will be impossible to satisfy the pre-conditions of most other methods of that object. For example, in verified code, after calling set on an object it is no longer possible to call insert. However, the problem can still only be detected at the level of client code when this scenario actually occurs, and this compromises modularity.

Furthermore, despite the fact that set goes against the intentions of OList, this will only be detected if client code happens to call set. This means that the problem will only be unreliably detected, possibly long after the class OList is written. Instead, the violation of our intended design should be detected at the point of defining and verifying the new version of the OList class, since this is where the problem occurs.

This difference between pure logics and mixed logics becomes even more apparent when considering subclasses. A declared invariant specifies not just an implicit contract for the method definitions in the same class, but for all subclasses also. In the case of overridden methods, this is not a big issue, since behavioural subtyping [8] enforces that the subclass version of the method still adheres to the superclass specification. However, when a subclass defines new methods, we can observe a difference, in the same way as for upgrading a class. While an invariant protocol would automatically impose a restriction on the method bodies of any newly-defined methods, if the invariant had already been refactored into the pre- and post-conditions of the existing methods, no such restriction would be imposed.

Even though not all existing code follows behavioural subtyping, and even though recent work has shown how to reason about programs where subclass methods do not satisfy the contract of overridden methods [14, 5], it is generally accepted that behavioural subtyping leads to more robust design.

### 2.2 Invariant protocols enforce robust client interfaces

We believe that an invariant declares not only a property which is intended to hold for an object, but also a responsibility for maintaining it. In particular, the implementation of a class (and any other tightly-coupled classes) should take responsibility for maintaining the invariant and (ideally) clients should not need to be concerned with how it is maintained. This not only provides a philosophical division of responsibility, but can aid modular reasoning, as we will explain here.

Consider a variation of ordered lists which guarantees additionally the property that a list contains no duplicates. We consider this a new class with an intended invariant of the form: *Ordered*(this) $\wedge$ *NoDups*(this). However, for the sake of the argument, we will write the specifications in a pure logic style here. We consider the form of a class UniqueOList along with two client classes:

```
class UniqueOList{
   UniqueOList next;
   Item contents;

   // pre:  Ordered(list) ∧ NoDups(list)
   // post: Ordered(list) ∧ NoDups(list)∧ this.contains(it)
   void insert (Item it ) { ... }

    ...
}

class ClientOne {
  ClientTwo collaborator ;

  // pre: Ordered(list) ∧ NoDups(list)
  // post: NoDups(list)
  void doStuff(UniqueOList list) {
    //
    list . insert (new Item());
    // etc ..

    // properties of list ?
    collaborator .doMoreStuff( list );
  }
}

class ClientTwo {

  // pre: NoDups(this)
  // post: NoDups(this)
  void doMoreStuff(UniqueOList list ) {
    // do more stuff .. but don't call methods on list
  }
}
```

In the example code above, we have made use of the flexibility permitted in a pure logic: we suppose that we are in a situation where doMoreStuff does not depend on the ordered-ness of the passed list, and so chooses not to require this property in its specification. This is acceptable so long as the method doStuff doesn't require ordered-ness to be guaranteed in its post-condition. However, it would not usually be allowed in a mixed logic setting, since the intended invariant is not guaranteed to be preserved by the specifications.

In principle, this set-up should verify in a pure logic. The fact that some methods require properties of list weaker than the notional invariant does not cause any difficulties, so long as doMoreStuff doesn't call any methods on list which require the full invariant. However, consider what happens if the implementation of doMoreStuff in class ClientTwo is upgraded. Suppose that in a new version, we wish to depend on the property that list is sorted after all, perhaps because we wish to call methods on list. Now, we need to strengthen the pre-condition of doMoreStuff, in order to expect the full invariant. But this implies that we must reverify doStuff in class ClientOne also. Of course, this happens in general when one wishes to vary the specification of a method - the call-sites of the method must in general be reverified. But the point here is that, the change in specification is due to the weakness of requirements made on list in the first place. Intuitively, since list is a UniqueOList it should be required to satisfy the invariant of that class. Furthermore, we argue that it should not be the responsibility of the client classes to negotiate how much of this invariant they expect about list: by declaring an invariant in the UniqueOList class, a common expectation is drawn up for all uses of such objects, and responsibility for maintaining this property should be enforced. In other words, allowing too much flexibility regarding the encoding of invariants breaks the philosophical meaning of the invariant, and may potentially weaken modular reasoning by leaving contracts more vulnerable to change during software evolution.

## 2.3 Invariant-based reasoning reflects cooperation between objects

The concept of an object invariant can naturally lead to programming patterns which allow a local and flexible implementation of a constrained data structure, in which the elements of the structure must maintain properties with respect to one another. A good example is the Composite pattern.

Now consider a naïve specification and implementation of the add method, which does not consider how to preserve the intended invariant *node* for the data structure as a whole:

```
// pre: o.parent == null ∧ node(this)
// post: children . contains( this ) ∧ node(this)
void add(Node o) {
  o.parent = this;
  children .add(o);
  value = // max{this.value, o.value};
}
```

The implementation satisfies its specification [2] and also preserves the invariant of the receiver. But this misses the point: the idea of the Composite pattern is that modifications can be sound *so long as* the invariants of nodes depending on the locations are also maintained. The underlying intention is for an implementation which is *considerate* of the invariants of the nodes above the modified point in the data structure. More explicitly, the implementation of a method on an object $o$ in the data structure needs to be concerned not only with the invariant of $o$, but the invariants of other objects in the data structure (regardless of whether those objects are involved directly in the $o$'s invariant. This can be achieved (for example), by the following alternative implementation:

```
// pre: o.parent == null
// post: children . contains( this )
void add(Node o) {
  o.parent = this;
  children .add(o);
  this.update();
}

void update() {
  value =
     // max {value}∪{o.value | children.contains(o)};
```

[2]Note that we do not need to require *node*($o$) in the precondition, since the *node* predicate is only concerned with the direct children.

```
    if (parent != null) parent.update();
  }
```

The new implementation ensures that the invariant is preserved for all objects in the data structure. It is not necessary to explore the data structure arbitrarily to guarantee this; instead, the form of the desired invariant implies that it can only be violated for an object $o$ if either the set $o$.children is modified, or $o$.value changes for some object $o'$ contained in $o$.children. Turning this information on its head, it becomes clear that whenever an object $o'$ obtains a new parent, or changes its value while it has a parent, the update() method should be called on that parent (which will fix a broken invariant). The two calls to update() in the code above correspond exactly to these cases.

The point we wish to illustrate with this example is that it is the consideration of invariants of other objects which allows flexible patterns such as the Composite, and the correct identification of objects whose invariants are affected which avoids being unnecessarily concerned with properties of arbitrary objects in the data structure. Furthermore, since this pattern is, we believe, designed with an invariant-based reasoning in mind, it makes sense for the verification of the pattern to follow a similar line of reasoning.

## 3. INVARIANT-BASED REASONING CAN EXPLOIT SOFTWARE DESIGN

We believe that for a verification approach to be both natural to a programmer, and scalable to large programming projects, it is essential that the principles guiding the design of the code are reflected and exploited in specifying and verifying the code. Having argued in the previous section that invariant-based reasoning can aid software design, we wish to consider the dual perspective: we believe that, if code is designed with invariants in mind, then it is natural and useful to reflect this in the corresponding reasoning.

Where an invariant is guaranteed by provider code in an encapsulated fashion (client code can neither observe the invariant in a broken state, nor directly break it), it is inherent in the software design that the client code will implicitly depend on the invariant holding whenever it is useful. This delegation of responsibility for a property is, we believe, a fundamental aspect of object-oriented design philosophy. Therefore, when reasoning, we would like to reflect this philosophy where it applies.

To illustrate this point, consider client code manipulating several OList instances. Considering the pure logic specification of OList, when we verify a client, we will need to establish the *Ordered* predicate on the receivers before each method call.

```
Item i, j;
OList l1=new OList(); // 1:  establishes  Ordered(l1)
OList l2=new OList(); // 2:  establishes  Ordered(l2)
OList l3=new OList(); // 3:  establishes  Ordered(l3)
// some code          // 4:  to show Ordered(l1)
l1.insert(i);         // 5:  establishes  Ordered(l1)
                      // 6:  to show Ordered(l2)
l2.insert(j);
```

In particular, before the call l1.insert(i) the client has to establish *Ordered*(l1), and thus will have to prove that the "some code" preserves *Ordered*(l1). Similar arguments apply to the call l2.insert(j).

At a practical level, even if these proof steps could be discharged by a tool, they clutter the program proof, impose more work on the verifier, and require proof steps which are essentially superfluous. More importantly, these superfluous steps ignore the promise made by the invariant, which is to declare to clients that a property is guaranteed to be preserved, and need not concern them. In a sense the argument for verification becomes non-modular, since the client must instead be concerned with whether this property of the object is preserved.

On the other hand, in a mixed logic, and provided the class OList was written so as to preserve the invariant, and provided the class OList does not leak any part of its internal representation, all verification steps above are essentially superfluous. Namely, the invariant protocol can enforce that *all* objects of class OList when seen by the client, satisfy their invariant.

Using the mixed logic specification of OList, and assuming proper encapsulation, we can safely assume the invariant to hold:

```
Item i, j;
OList l1=new OList(); // 1:  implicitly  Ordered(l1)
OList l2=new OList(); // 2:  implicitly  Ordered(l2)
OList l3=new OList(); // 3:  implicitly  Ordered(l3)
// some code          // 4:
l1.insert(i);         // 5:
                      // 6:
l2.insert(j);
```

This does depend on an invariant-based approach flexible enough to handle complex examples, while guaranteeing encapsulation where appropriate. This can be challenging: although many current technologies exist which incorporate invariant-based reasoning (*e.g.,* Boogie [1], OT, VT [12]), the mechanisms which enforce encapsulation (typically based on some form of *ownership*) are not flexible enough to handle all natural examples. We believe that the development of a more refined ability to describe patterns of encapsulation and invariant protocols is important future work.

## 4. OBJECT INVARIANTS ALLOW LOCAL VERIFICATION OF GLOBAL PROPERTIES

Object invariants and invariant protocols allow some requirements to be global and implicit, and therefore allow the specifications to be local, *i.e.,* concerned with just one object, rather than with a collection of objects.

### 4.1 Global properties of accessible objects

In section 2.3, we argued that the implementation of the Composite pattern implicitly encodes an invariant-based argument. Since we believe that the design of the code should be reflected in the specification and verification, we would like to return to the question of how to specify and verify this example.

#### *4.1.1 Global properties of accessible objects - pure logics*

If one attempts to specify the implementation in a pure logic, through predicates in the pre- and post-conditions, then recursive predicates can be employed to capture the requirements. Since there is no longer a protocol to guarantee invariant preservation by default, it is typical to explicitly express consistency recursively through the data structure. For example, the following predicate expresses that a whole tree is consistent:

$$\mathsf{tree}(o) \equiv \mathsf{node}(o) \wedge \forall\, o' \in o.\mathsf{children},\ \mathsf{tree}(o')$$

However, in order to specify the behaviour of this example, given that parent fields get notified of updates, one needs to recurse upwards through the structure, as well as downwards:

$$\begin{aligned}&\mathsf{above\_tree}(o) \equiv\\ &\quad \mathsf{tree}(o) \wedge ((o.\mathsf{parent} == \mathbf{null}) \vee\\ &\qquad\qquad (\mathsf{above\_tree}(o.\mathsf{parent})))\end{aligned}$$

Furthermore, to specify the consistency properties which are not affected by a call to update, one needs a predicate to describe the areas of the tree which form the other branches:

$$\begin{aligned}&\mathsf{other\_branches}(o) \equiv\\ &\quad ((o.\mathsf{parent} == \mathbf{null}) \vee (\mathsf{other\_branches}(o.\mathsf{parent}) \wedge\\ &\qquad \forall\, o' \in o.\mathsf{parent.children},\ o' \neq o \Rightarrow \mathsf{tree}(o')))\end{aligned}$$

We can then specify the example as follows:

```
// pre: o.parent == null ∧ above_tree(this)
// post: children.contains(this) ∧ above_tree(this)
void add(Node o) {
  o.parent = this;
  children.add(o);
  this.update();
}

// pre: other_branches(this) ∧ ∀ o' ∈ this.children,
     tree(o')
// post: above_tree(this)
void update() {
  value =
    // max {value}∪{o.value | children.contains(o)};
  if (parent != null) parent.update();
}
```

The resulting specification is adequate, but it is overly conservative (since the predicate definition above_tree is concerned with the whole tree at a time, whereas the natural argument is only concerned with at most one broken object, along with the current receiver), and the choice of specification is error-prone (since the requirement to preserve the invariants of other objects is not inherent in the verification effort). One might pick a more streamlined predicate definition, for example avoiding the redundancy in the predicate above. Jacobs et al. [4] developed a cleverer definition of the predicate for this example, splitting the tree into a subtree and a *context*, in order to facilitate their separation-logic-based verification. However, these predicates still ultimately cover the *whole tree* at a time; it is less cumbersome to reason about which invariants are broken, than to assert which ones hold, as the predicates do.

#### *4.1.2 Global properties of accessible objects - mixed logics*

For comparison, we consider how the example might be specified in a mixed logic. In order to deal with update, we need a construct to explicitly declare that the invariant (tree(this)) which might by default be expected to hold is actually allowed to be broken. We will use a construct broken(o) to declare that the invariant of object o may be broken when the method is called. However, in the post-state, we still require that the corresponding invariant is guaranteed to hold - in this way we reflect the role of the update method, which is to *fix* a broken invariant.

```
// pre: o.parent == null
// post: children.contains(this)
void add(Node o) {
  o.parent = this;
  children.add(o);
  this.update();
}

// pre: broken(this)
// post: true
void update() {
  value =
    // max {value}∪{o.value | children.contains(o)};
  if (parent != null) parent.update();
}
```

Note that we do not require a post-condition for update, but there is an implicit obligation to fix the invariant of the receiver (and preserve all others), due to the declaration broken in the pre-state. Note also that the implicit protocol regarding invariants allows the specifications to be shorter, and to concentrate on the specifics of the method behaviour. In particular, there is no need or motivation to express properties of objects not concerned with the method implementations, in contrast to the situation with a pure logic.

There are two subtleties which have been glossed over in the argument above. One is that, in order to make this kind of invariant-based reasoning adequate, one needs to have some restriction of *which* invariants may depend on which objects' fields. [3]

The other subtlety to the reasoning about invariants above is the following: we are tacitly assuming that the concepts expressed by the field children and parent are inverse to one another. Although this seems intuitive, there is nothing in our specifications which actually enforces this fact. One could express this through a further invariant however, and the approach to verifying this kind of structural invariant is discussed in the next subsection.

## 4.2 Global properties depending on inaccessible objects

There is an area where invariant-based reasoning seems almost indispensable: in the verification of properties concerning potentially inaccessible objects.

Such an example arises in the PIP[4], which - in a simplified view - is based on a graph structure, represented by Node objects, in which a Node has at most one parent and where cycles in the parent relation are possible. A Node's field

[3]This problem can be tackled in several ways: one could have a methodology to track which objects depend on the field for their invariants (e.g., the *friends* work of Barnett and Naumann [3], and the work of Middelcoop [11]), or one could in certain cases apply the ideas of visibility-based invariant reasoning (i.e., *visibility-based invariants* in Spec♯ [2], and to the *visibility technique* of Müller et al. [12]). One could also hope to encapsulate the fields of the data structure, in such a way that client code would be unable to depend on them for invariants (this is difficult if the client invariants can depend on state indirectly, via e.g., pure method calls in specifications).

[4]the Priority Inheritance Protocol [15]

value is expected to be the maximum of its initVal value, and the initVal value of all its (transitive) descendants:

$P1(\mathsf{n}) \equiv$
$\mathsf{n.value}{=}max(\{\mathsf{n.initVal}\} \cup \{\mathsf{n'.value} | \mathsf{n}{\in}\mathsf{n'.parent}^*\})$

The problem with property $P1$ is that it is concerned with arbitrary other nodes in the graph, and so is expensive to check and awkward to verify. Instead, it is advantageous for the implementation to be concerned with a localised version of this invariant: nodes should be concerned only with their immediate neighbours in the graph structure. Intuitively, the following alternative *local* invariant can be aimed for: the current value of a node is the maximum of its initVal field and the value field of all direct children:

$P2(\mathsf{n}) \equiv \mathsf{n.value}{=}max(\{\mathsf{n.initVal}\}{\cup}\{\mathsf{n'.value}|\mathsf{n}{=}\mathsf{n'.parent}\})$

If all nodes maintain this property, then an inductive argument can be made to show that the invariant originally intended is guaranteed, i.e, we can show that

$\forall \mathsf{n} : \mathsf{Node}.P1(\mathsf{n}) \leftrightarrow \forall \mathsf{n} : \mathsf{Node}.P2(\mathsf{n})$.

The PIP example is very similar to the Composite. The key difference is that the "parent" relationship no longer philosophically reflects a "part of" relationship between the objects, but a rather looser association between essentially independent entities. In particular, the implementation does not naturally require references from a Node to its children. This presents difficulties in verifying the intended invariant. From a practical perspective, it is possible to have cycles in the data structure, which is forbidden for the Composite.

In the interest of brevity, in the current paper we have considerably simplified the problem, and in particular we do not allow the removal of parent edges in the implied graph (but deal only with addition of new edges). Below we give an implementation of class Node, intended to maintain $P2$ for all Node objects:

```
class Node {
  Node parent;
  int  initVal , value;

  Node(int val) {
       parent = null;     initVal = value = val;
  }

  // pre: nd.parent == null
  void acquire(Node nd) {
       nd.parent = this; this.update(nd.value);
  }

  protected update(int newValue) {
     if (newValue > value)
     { value = newValue;
        if (parent != null){ parent.update(value); }  }
  }
}
```

At an informal level the partial correctness of the class Node as above is obvious (in fact, with some work one can also argue termination, but this is not directly of interest to us here). Nevertheless, the specification and verification of class Node pose some problems, because $P2(\mathsf{n})$ depends on *all* nodes pointing to it; which, in particular are not accessible from n.

We envisage four approaches to the problem: 1. Add ghost state to track the objects in question, 2. Extra parameters to the predicates to express the base of reference, 3. Explicit use of universal quantification, 4. Considerate Programming. We briefly discuss the four approaches.

### *4.2.1 Ghost state to track the objects in question*

An initial reaction is to add ghost state which tracks the objects in question. Then we could transform our requirement to

$P2a(\mathsf{n}) \equiv$
$\mathsf{n.value}{=}max(\{\mathsf{n.initVal}\}{\cup}\{\mathsf{n'.value}|\mathsf{n'}{\in}\mathsf{n.children}\})$

and specify our code as follows:

```
class Node {
  Node parent;
  int  initVal , value;
  ghost Set<Node> children;

  // pre:  P2a(this) ∧ P2a(nd) ∧ ...
  // post: P2a(this) ∧ P2a(nd) ∧ ...
  void acquire(Node nd) { ...  }

  ...
}
```

However, this reduces the problem to one of consistency of the ghost state: how can it be known that *all* nodes whose parent points to the current object, are actually stored in the ghost state of the object? In other words, how can we establish $P2b$, where

$P2b(\mathsf{n}) \equiv \mathsf{n.children} = \{\mathsf{n'} \mid \mathsf{n} = \mathsf{n'.parent}\}$

Again, this is a property concerned with potentially inaccessible objects on the heap. This approach has brought *no* progress: we still do not have a way to handle such an invariant.

### *4.2.2 Further parameters as frames of reference*

We consider attempting to track the state of the object graph in the specification, by introducing a set of nodes as as a frame of reference. We require such a set to be closed under parent field references, and its elements to satisfy a variant of property $P2$ *restricted* to the children nodes from that set:

$P3(\mathsf{NS}) \equiv \forall \mathsf{n} \in \mathsf{NS} : \; P4(\mathsf{n}, \mathsf{NS}) \;\wedge\; \mathsf{n.parent} \in \mathsf{NS}$
$P4(\mathsf{n}, \mathsf{NS}) \equiv \mathsf{n.value} =$
$max(\{\mathsf{n.initVal}\} \cup \{\mathsf{n'.value} \mid \mathsf{n'}{\in}\mathsf{NS} \;\wedge\; \mathsf{n} = \mathsf{n'.parent}\})$

We now introduce a static field NodeSet to track the set of nodes[5]. We sketch the specification of the class Node as follows:

```
class Node {
  Node parent;
  int  initVal , value;
  static Set<Node> NodeSet;

  // pre: P3(NodeSet)
  // post: P3(NodeSet) ∧ this ∈ NodeSet
  Node(int value) { ... NodeSet.add(this); }

  // pre: P3(NodeSet) ∧ this, nd ∈ NodeSet
  //                   ∧ nd.parent = null
  // post: P3(NodeSet) ∧ .... ∧ nd.parent = this
  void acquire(Node nd) { ...  }

  // pre:
       P3(NodeSet \ {this}) ∧ this.parent ∈ NodeSet
  //           this.value == max({...} \ {newValue})
```

[5]We could have also kept the set in some other globally accessible way, or even passed it as a parameter.

```
  // post: $P3$(NodeSet)
  protected update(int newValue) { ... }
}
```

In the approach presented above the set NodeSet may contain unrelated nodes; as such, the specification of the behaviour of one Node is concerned with essentially arbitrary other Nodes in the graph of objects – one might consider this "paranoid" reasoning! This abandons any kind of locality in the specification, and means that a representation of the graph is (implicitly or explicitly) carried around to perform the reasoning. From the point of view of the actual implementation, which manages to be extremely local (each Node only needs access to its parent), such a global approach to the verification would be disappointing. [6]

Furthermore, the introduction of a new entity NodeSet is tantamount to the introduction of ghost state.

Most importantly, the specification given above is weaker than the intended specification where *all* Nodes satisfy property $P2$. In particular, there is *no* guarantee that

$o.\mathsf{parent} \in \mathsf{Node.NodeSet} \implies o \in \mathsf{Node.NodeSet}$

And thus, there is still *no* guarantee that

$\forall o \in \mathsf{Node.NodeSet}: \; P2(o)$

Nor is there a guarantee that

$\forall o \in \mathsf{Node.NodeSet}: \; P2(o.\mathsf{parent})$

Therefore, this approach has brought no further progress than the ghost state introduced in section 4.2.1.

### 4.2.3 Explicit use of universal quantification

We can encode the intended invariant $\forall \mathsf{n} : \mathsf{Node}.P2(\mathsf{n})$ as part of the pre- and post-conditions for all methods:

```
class Node {
  Node parent;
  int initVal , value;

  // pre: ∀n : Node.$P2$(n)
  // post: ∀n : Node.$P2$(n)
  Node(int value) { ... }

  // pre: ∀n : Node.$P2$(n) ∧ nd.parent == null
  // post: ∀n : Node.$P2$(n) ∧ nd.parent == this
  void acquire(Node nd) { ... }

  // pre: ∀n : Node \ {this}.$P2$(n) ∧
  //          this.value == max({...} \ {newValue})
  // post: ∀n : Node.$P2$(n)
  protected update(int newValue) { ... }
}
```

The specification above correctly expresses the behaviour of the class Node, but has the disadvantage that it requires universal quantification. Universal quantification in specifications is not supported by all approaches (for example, separation logics do not admit quantification over the heap). However, even when they are, they have both philosophical and practical disadvantages. The specification using universal quantification abandons any kind of local characterisation of the behaviour of the method, despite its rather local implementation. By contrast, an invariant protocol typically provides a way of maintaining global program invariants that reflect the quantification over objects through fairly simple proof obligations about the invariants of particular objects. Furthermore, the more-specific proof obligations about the invariants of particular objects are much more practical for theorem provers than to show that the quantified formulas are preserved across a method execution.

### 4.2.4 Considerate Programming

From the point of view of the implementation, the intuitive reason why the intended invariant is guaranteed can be seen by inspection of the methods. Here, as soon as any parent or value field modification is made, which might potentially break the invariant of another object, the appropriate object is notified (via the method update), and so is able to account for the change.

This pattern of programming is typical of invariant-style reasoning - rather than declaring explicitly which objects are in which states, one instead ensures that the invariants of all objects will be preserved by every method call. In particular, we like to call this "considerate" reasoning - a method body must take into account the possible effects it has on the properties of other objects, regardless of whether those objects are relevant for the properties of the current receiver. The knowledge that all objects behave "considerately" (according to a specified invariant protocol) allows implicit knowledge that these invariants are preserved, without requiring access to them directly.

Although this reasoning style seems natural with invariants it is not directly supported in the mainstream verification methodologies. Nonetheless, existing research shows that such a reasoning style is feasible and sound [3, 11]. We believe that such considerate programming/reasoning should be expressible in specifications, and supported by verification tools.

We employ the *broken* construct again to indicate that, contrary to the default behaviour, the invariant of an object is not required to hold in the pre-state of a method:

```
class Node {
  Node parent;
  int initVal , value;
  // INV $P2(this)$

  Node(int value) { ... }

  void acquire(Node nd) { ... }

  // pre: broken(this)∧
  //          this.value = max({...} \ {newValue})
  protected update(int newValue) { ... }
}
```

We believe that the style advocated above directly expresses the intuition guiding the thinking behind the code. It hides all unnecessary details from the client. Furthermore, verification of the class will reflect the strategy in the design of the code - namely "consideration", and the fact that modification of field newValue affects the validity of the invariant $P2$ for the parent of the current object, but no other objects.

## 5. CONCLUSIONS

We have argued that the concept of invariant is both natural and useful for many aspects of object-oriented verification. Invariants can express the design intentions of a pro-

[6]This approach may look more natural in the functional programming context (where all information is passed explicitly), but looks less natural in object-oriented (where we expect to deal with implicit information).

grammer, and provide an implicit contract for future subclasses which cannot be captured directly by method specifications. Furthermore, the ability to separate the concerns of the invariant from the method specifications themselves makes clear the division of responsibility in the verification, and avoids unnecessarily cluttering the client specifications with properties guaranteed by code they cannot see. For these reasons, pure logics, while flexible, require more verification steps than mixed logics, and are not always able to express the intended structure of an argument, as it corresponds with the particular program design.

In particular, in any style of specification and verification, the problem of establishing whether a particular property (which may be expressed by a predicate, invariant or otherwise) is preserved by execution of code arises frequently (the *frame problem*). While there are techniques for reasoning directly about framing (using footprints, modifies clauses, framing inference rules etc.), mixed logics can exploit properties such as encapsulation and information hiding to drastically reduce the number of proof obligations which need to be discharged by any of these methods. The invariant protocol inherent in a mixed logic can provide guarantees that these simplified proof obligations are sufficient, which are justified once and for all by a soundness proof for the technique, rather than requiring justification each time the situation arises during verification. At the end of section 3, and in section 4.2.4, we discussed examples of the uses of such invariant protocols, at the level of sketches. Fully worked out, and implemented protocols exist, *e.g.,* invariant protocols for private invariants [6], where restrictions on the accessibility of the fields mentioned in an invariant are used to simplify proof obligations, and the Boogie methodology [7], where proof obligations for individual objects are sufficient to maintain a global property that quantifies over all objects, and specifies which invariants are known to hold.

We have also argued that invariants can help reflect properties inherent in a software design, and that this can help keep the verification argument closer to the original intentions of the implementation. Perhaps the best example of this is the Composite pattern, whose design appears to strongly reflect an invariant-based thinking, and whose verification is made significantly simpler in a verification approach which does the same. More generally, while invariant protocols can be restrictive, they also add structure to the verification effort, and to the proofs themselves. Since the patterns that a protocol describes (should) naturally occur in software designs, the ability to re-use the structured arguments which a mixed logic provide saves unnecessary repetition in the verification effort.

As a separate point, we have observed that mixed logics allow the verification of "global" properties over potentially inaccessible objects, and in a local way. We have argued that the key to the flexible verification of these examples is the "considerate reasoning" which comes naturally with invariants.

While we have shown many important uses for invariants, and particularly for the "considerate programming" philosophy which comes with the reasoning, we do not believe that existing technologies for invariant-based reasoning support a rich enough variety of verification patterns. In particular, the pattern of reasoning present in the Composite and PIP examples is not naturally supported by mainstream verification technologies. Identifying the prevalent and useful patterns which arise in verification, and how they can be mixed within a large program will be interesting future work.

As future work we hope to develop further techniques to support verification efforts which can be flexible, natural, and appealing to programmers. To this end, we believe that a verification approach which can closely reflect the design decisions of programmers is essential, and that as we have explained, invariants will play a vital role.

## Acknowledgements

We would like to thank Frank Piessens for many detailed discussions and suggestions on the content of this paper. We would also like to thank the anonymous referees for their helpful feedback.

This work was funded in part by the IST-2005-015905 MOBIUS project.

## 6. REFERENCES

[1] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *JOT*, 3(6):27–56, 2004.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, LNCS, pages 49–69. Springer-Verlag, 2005.

[3] M. Barnett and D. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC*, volume 3125 of *LNCS*, pages 54–84. Springer, 2004.

[4] Frank Piessens Bart Jacobs, Jan Smans. Verifying the composite pattern using separation logic. In *SAVCBS*, 2008.

[5] W.-N. Chin, C. David, H. Nguyen, and S. Qin. Enhancing modular oo verification with separation logic. In *POPL*. ACM Press, 2008.

[6] G. T. Leavens and P. Müller. Information hiding and visibility in interface specifications. In *ICSE*, pages 385–395. IEEE, 2007.

[7] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.

[8] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM ToPLAS*, 16(6):1811–1841, 1994.

[9] Y. Lu, J. Potter, and J. Xue. Object Invariants and Effects. In *ECOOP*, volume 4609 of *LNCS*, pages 202–226. Springer-Verlag, 2007.

[10] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[11] R. Middelkoop, C. Huizing, R. Kuiper, and E. J. Luit. Invariants for non-hierarchical object structures. *Electr. Notes Theor. Comput. Sci.*, 195:211–229, 2008.

[12] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.

[13] M. Parkinson. Class invariants: the end of the road? In *International Workshop on Aliasing, Confinement and Ownership*, 2007.

[14] M. Parkinson and G. Bierman. Separation logic, abstraction and inheritance. In *POPL*, pages 75–86. ACM Press, 2008.

[15] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.