# A Universe-Type-Based Verification Technique for Mutable Static Fields and Methods – Work in progress –

A. J. Summers[1], S. Drossopoulou[1], and P. Müller[2]
[1] Imperial College London, [2] Microsoft Research, Redmond

**Abstract.** We present a novel technique for the verification of invariants in the setting of a Java-like language including static fields and methods. The technique is a generalisation of the existing Visibility Technique of Müller et al., which employs universe types.
In order to cater for mutable static fields, we extend this topology to multiple trees (a forest), where each tree is rooted in a class. This allows classes to naturally own object instances as their static fields. We describe how to extend the Visibility Technique to this topology, incorporating extra flexibility for the treatment of static methods.
We encounter a potential source of callbacks not present in the original technique, and show how to overcome this using an effects system. To allow flexible and modular verification, we refine our topology with a hierarchy of 'levels'.

## 1 Introduction

In this paper, we extend the Visibility Technique (VT for short) [10], a known visible states verification technique based on universe types, to cater for static fields and methods. When adding statics to verification, one needs to address the following questions:

1. Where in the topology do static fields appear?
2. May instance methods update static fields?
3. May static invariants mention the fields of objects of their class?
4. May instance invariants mention static fields of their class, or of other classes?
5. Can static methods break invariants of objects, and if so, of which objects?
6. Can instance methods break static invariants, and if so, of which classes?
7. What proof obligations are necessary before a call to a static method?
8. What proof obligations are necessary before a call to an instance method?

In this paper, we explore these questions in the context of VT, and extend the technique and heap topology to handle static fields. In the process, we encounter a potential source of callbacks not present in VT, and devote much of this paper to solving this problem. We develop an approach involving a combination of effect annotations and refinements to the heap topology using levels. We then extend the technique to allow more expressive invariants.

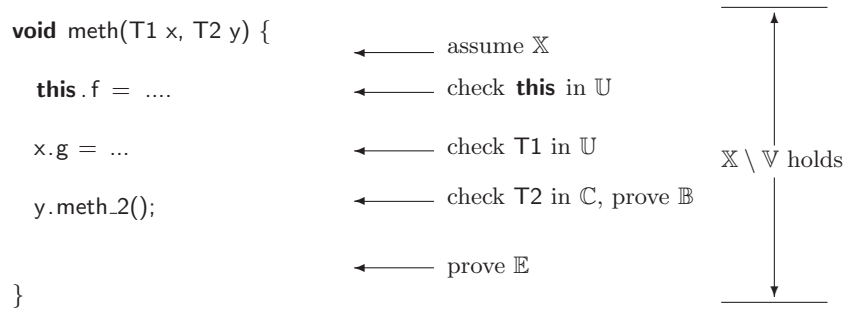


**Fig. 1.** Illustration of the use of the seven components.

In Sec. 2 we give the background to visible states verification techniques, universe types, and VT. In Sec. 3 we discuss the first two questions from above. In Sec. 4 we address the others, give a first attempt to an extension of VT, and argue that it is sound. We refine our approach with improved calculations of effects in Sec. 5, and with more powerful static class invariants in Sec. 6. In Sec. 7 we conclude. Proof sketches can be found in the longer version of our work, at `http://www.doc.ic.ac.uk/~ajs300m/papers/staticsFull.pdf`.

## 2 Background

Visible state verification techniques are defined around the notion of *visible states*, which correspond to the beginning and the end of any method call. At these visible states, the invariants of certain objects (exactly *which* objects depends on the contents of the call stack, and on the particular technique) are guaranteed to hold.

Several visible states techniques have been suggested, *e.g.*, [12, 3, 10, 8], and they share many commonalities. As suggested in [2], these commonalities, as well as the differences, can be neatly distilled in terms of the following seven components:

$\mathbb{X}$ invariants expected to hold in visible states.
$\mathbb{V}$ invariants *vulnerable* to a method, *i.e.*, which may be broken while it executes.
$\mathbb{D}$ invariants that may depend on a given heap location[1].
$\mathbb{B}$ invariants that must be proven to hold before a method call.
$\mathbb{E}$ invariants that must be proven to hold at the end of a method body.
$\mathbb{U}$ permitted receivers for field updates.
$\mathbb{C}$ permitted receivers for method calls.

The use of these components should be clear from their description above, but is also shown in Fig. 1 through annotating a method `meth1`: $\mathbb{X}$ may be assumed to hold in the pre- and post-states of the method. Between these visible states, some object invariants may be broken, but $\mathbb{X} \setminus \mathbb{V}$ is guaranteed to hold. Field updates and method calls are allowed if the receiver object is in $\mathbb{U}$ and $\mathbb{C}$, respectively. Before a method call, $\mathbb{B}$ must be proven. At the end of the method body, $\mathbb{E}$ must be proven. Finally, assignments to **this**.f and x.g affect at most $\mathbb{D}$.

[1] This also characterises indirectly the locations an invariant may depend on.

In [2], five *soundness conditions* are presented, and it is proven that if these conditions are satisfied, then the technique is sound (the expected invariants hold at visible states). In this paper, we use the framework of [2] informally, since the technique presented here does not quite fit the present formalism. However, the soundness conditions still guided us in the design of our technique. Informally, the five sufficient soundness conditions can be described as follows:

**Definition 1 (Soundness Conditions).**

1. $\mathbb{X}_{m'} \setminus (\mathbb{X}_m \setminus \mathbb{V}_m) \subseteq \mathbb{B}$
   *When a legal (according to the technique,* i.e.*,* $\mathbb{C}$*) call is made to a method* $m'$ *from a method* $m$*, all of the invariants which are both expected to hold by the new method* ($\mathbb{X}_{m'}$)*, and are not currently known to hold in the calling method (*i.e.*, not within* $\mathbb{X}_m \setminus \mathbb{V}_m$*), must be within the proof obligations made before the method call* ($\mathbb{B}$)*.*
2. $\mathbb{V} \cap \mathbb{X} \subseteq \mathbb{E}$
   *The invariants both expected* ($\mathbb{X}$) *by and vulnerable to* ($\mathbb{V}$) *a method, must be within the proof obligations at the end of the method* ($\mathbb{E}$)*.*
3. $\mathbb{V}_{m'} \setminus \mathbb{E}_{m'} \subseteq \mathbb{V}_m$
   *If a (legal) method call is made to a method* $m'$ *from a method* $m$*, any invariants which are vulnerable to* $m'$ *and not reestablished by* $m'$*, must be vulnerable to* $m$*.*
4. $\mathbb{D} \subseteq \mathbb{V}$
   *Invariants depending on fields which may be legally modified (according to the technique,* i.e.*,* $\mathbb{U}$*) by a method, are vulnerable to the method.*
5. $\mathbb{X}_{c'} \subseteq \mathbb{X}_c$ *and* $(\mathbb{V}_{c'} \setminus \mathbb{E}_{c'}) \subseteq (\mathbb{V}_c \setminus \mathbb{E}_c)$
   *If a method is overridden, then in the subclass version, no more invariants may be expected or left broken than in the superclass version.*

One such visible states technique, the Visibility Technique (VT), was developed on top of universe types [10] with the aim to guide the verification process, and to guarantee modularity. Universe types [9] organise the heap into a tree topology, in which each object is *owned* by another object, and where an object $o$ considers another object $o'$, as its peer if they have the same direct owner; it considers it its rep if it is its direct owner[2]. The *owner-as-modifier discipline* (hereafter OAM) restricts field updates and method calls, implying in particular that the receivers of methods are only allowed to be reps or peers. Thus, at any time in execution any receiver on the call stack[3] is directly followed either by a rep or a peer. In Fig. 2, note that calls may only go "down" or "sideways".

The seven components from before have the following meaning for VT (we simplify slightly with respect to visibility, and to the exact class whose invariant we are considering):

[2] We do not discuss any or readonly references, nor pure methods.

[3] consisting of a sequence of activation records, each of which contains the then-current receiver

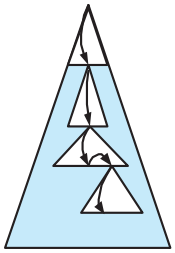

**Fig. 2.** Ownership Tree and Control Flow; the arrows show consecutive method calls and their receivers; note that calls go only "down", *i.e.*, to reps, or "sideways", *i.e.*, to peers. The shaded area indicates the area where objects satisfy their invariants.

$\mathbb{X}$ invariants of objects (reflexively, transitively) owned by peers.
$\mathbb{V}$ invariants of all transitive owners of the current receiver, plus invariants of peers of the current receiver.
$\mathbb{D}$ Invariants of peers and transitive owners may depend on the fields of an object.
$\mathbb{B}$ If the callee is a peer of the current receiver, then the invariants of all peers must be established. Otherwise, no proof obligations.
$\mathbb{E}$ the invariants of all visible peers.
$\mathbb{U}$ A field of an object may only be assigned to by the object's owner, or by any of its peers.
$\mathbb{C}$ A call is allowed if the callee is a peer or rep of the current receiver.

It can be shown that these parameters satisfy the soundness conditions of Def. 1 [2]. In particular, $\mathbb{X}$ and $\mathbb{V}$ and the owner-as-modifier discipline, guarantee that at any given time in execution, all objects are valid, except for those directly owned by one of the receivers on the call stack, *cf.* Fig. 2.

## 3 Heap Topology for Static Fields

The fundamental premise of this work is that classes should be able to own objects in the same way that other objects can. For example, if the behaviour of a class depends on a static field (to manage object creation, etc.) then this static field naturally 'belongs' to the inner workings of the class: its representation. This gives a natural interpretation of static rep fields: they should be treated analogously to instance rep fields, but with a class as their owner [7].

Thus, we extend our heap topology to include classes. Classes are the 'roots' of trees in our topology. As there are generally several classes in a program, our topology should allow for several such trees; we work with a *forest*. Furthermore, with classes acting as roots, there is no longer a need for an abstract **root** entity; these class-rooted trees make up the entire picture. Note that there are no objects at the 'same level' as the class entities, and classes do not have owners. In this paper, we do not consider a notion of static peer fields.

We interpret static fields and methods as instance fields and methods of the corresponding class object. That is, the class object (or class for short) is the receiver for an execution of a static method. We expect that modifications to static fields will be achieved by calling a static method of the class that declares

the field. In other words, static methods may update the fields of their receiver class, just like instance methods in VT may update fields of their receiver object.

To summarise the ideas so far:

1. Each point in our heap topology corresponds to either an object or a class.
2. Objects (but not classes) each have exactly one owner (a class or an object).
3. The current receiver (on the stack) can be either an object or a class.

## 4 Basic Technique

Having defined a suitable heap topology, in this section we generalise VT to our setting.

A key aspect of our technique is that we preserve the OAM property of VT. In the following technique, control is only allowed to enter a tree in the heap topology via the 'root'; *i.e.*, by calling a static method on the class at the root of the tree. Instance method calls are restricted in the same way as in VT. This implies the following property, which will be useful for our reasoning:

**Proposition 1.** *A call stack (including the current method-call) always starts with a class receiver. If an object $o$ is a receiver on the call stack, then the most recently-preceding class receiver on the call stack is the owner of the tree in which $o$ resides.*

For the moment, we treat static invariants analogously to VT instance invariants. Therefore, they can only mention expressions which start with the static fields of the same class (since they have no peers).

How then, to handle static method calls? According to VT, a method call is only allowed if the current receiver is either the owner or a peer of the callee receiver. Since classes do not have either owners or peers, this would make static methods impossible to call. We initially considered allowing arbitrary static method calls. This immediately creates problems with callbacks; in particular, how do we know the invariants of the new receiver hold when we make the call? If our current call stack has already visited this class, we may have left invariants broken.

We solve this problem by the following rule: a static method may only be called on a class $c$, if $c$ has not been a previous receiver on the call stack. However, this rule is slightly too restrictive, since it unnecessarily prohibits a static method of class $c$ from calling another static method of class $c$. Our rule of thumb is:

> *A static method of $c$ can be called if either $c$ is the current receiver, or $c$ is not already a receiver on the call stack.*

We are now in a position to define our technique in terms of the seven components. Compared with the description of VT, we need to extend $\mathbb{X}$ to reflect which invariants in other trees are expected, depending on the current call stack, and $\mathbb{C}$ to reflect the special rules for static method calls. The other five parameters

are straightforward generalisations of those for VT. We highlight the differences between our work and VT in italics, and point out the interpretation of these components with regard to a static method call in footnotes.

𝕏 invariants of objects (reflexively, transitively) owned by peers, *plus all invariants in trees not currently visited on the call stack*[4].

𝕍 invariants of all transitive owners of the current receiver, plus invariants of peers of the current receiver[5].

𝔻 Invariants of peers and transitive owners may depend on the field of an object *or class*[6].

𝔹 If the callee is a peer of the current receiver, then the invariants of all peers must be established. Otherwise, no proof obligations[7].

𝔼 the invariants of all visible peers[8].

𝕌 A field of an object *or class* may only be assigned to by its owner, or by any of its peers[9].

ℂ A call to an instance method is allowed if the callee is a `peer` or `rep` of the current receiver. *A call to a static method m on class c is allowed if either the current receiver is c itself, or else c is not on the current call stack.*

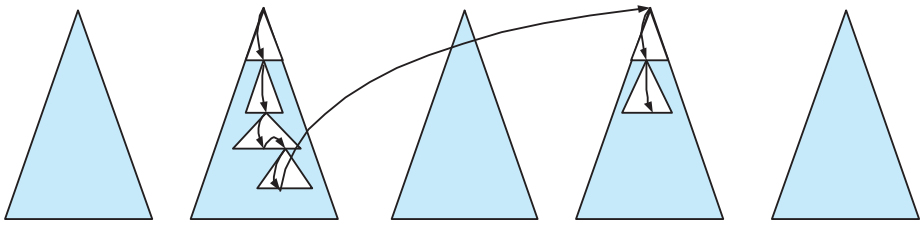

**Fig. 3.** Calls stacks across several trees, invariants hold in shaded areas.

When considering only the tree of the current receiver, the rules are essentially those of VT. However, the other trees either have none of their invariants expected, or all of them, depending on whether or not they have been visited on the current call stack. Furthermore, static methods are treated differently from

[4] For a static method, this amounts to all the invariants of the current tree, plus each unvisited tree.

[5] The only invariants vulnerable to a call of a static method in class $c$ are the static invariants of $c$ itself.

[6] The only invariants which are allowed to depend on a static field declared in class $c$ are the static invariants of $c$.

[7] If a static method is called on a class $c$ which is both caller and callee (a 'self' call), then the static invariants of $c$ must be reestablished first.

[8] For a static method, the invariants of the class.

[9] A static field can only be assigned to by the class itself.

instance method calls, in that any call is permitted so long as the callee has not been a receiver prior to the current one on the call stack.

Since $\mathbb{C}$ depends on the current call stack, it is not possible to statically verify whether a method call will be legal. We therefore identify next a way of conservatively approximating when method calls are legal.

**Effect Annotations.** For each class $c$ and method $m$, we require a set of *effects*, $\mathcal{E}ffs(c, m)$, predicting which classes may have static methods called on them as a result of calling $m$ of $c$. $\mathcal{E}ffs(c, m)$ is a (possibly empty) set of class names. This is described by requirements 1-3 in Def. 2 below.

If, from within the body of a static method $m$ of class $c$, we make a call to a (static or instance) method $m'$ defined in class $c'$ (with a different receiver), and if this method call may eventually result in a callback to $c$, then as a consequence of Def. 2, we must have $c \in \mathcal{E}ffs(c', m')$. Therefore, we can rule out dangerous callbacks on $c$ by insisting that any method which is called from a static method of $c$ does not contain $c$ in its effects. This is described through the method restriction in item 4 of Def. 2.

**Definition 2 (Valid Effects and Method Restrictions).**

1. *Within the body of a method $m$ of class $c$, if there is a call $e.m'(\ldots)$ and $e$ has static type $c'$, then $\mathcal{E}ffs(c', m') \subseteq \mathcal{E}ffs(c, m)$.*
2. *Within the body of a method $m$ of class $c$, if there is a call $c'.m'(\ldots)$ to a static method $m'$ of class $c'$, then*
   (a) *$\mathcal{E}ffs(c', m') \subseteq \mathcal{E}ffs(c, m)$ and*
   (b) *if $m$ is an instance method or $c \neq c'$ [10], then $c' \in \mathcal{E}ffs(c, m)$.*
3. *If $c'$ is a subclass of $c$ which overrides a method $m$, then $\mathcal{E}ffs(c', m) \subseteq \mathcal{E}ffs(c, m)$.*
4. *A static method $m$ of $c$ is legal, only if $c \notin \mathcal{E}ffs(c, m)$.*

**Soundness.** We focus on the first item from Def. 1: the guarantee that when a method call is made, the invariants expected in the new method will hold (because they have been preserved, or proven before the call is made). We claim that the other points can be easily established.

In the technique presented, all invariants may only depend on the fields of peers (if any) and any objects transitively owned. Furthermore, fields may only be modified by peers. Therefore, we have the following property:

**Proposition 2 (Broken Invariants).** *If, at runtime, the invariants of an object (or class) do not hold, then one of the receivers on the call stack (possibly the current one) must be the object (or class) itself or one of its peers.*

[10] *i.e.*, a static method always may call another static method from the same class.

To demonstrate that our restrictions using effects (Def. 2) are sufficient to guarantee that our desired notion of valid method call ($\mathbb{C}$) is always adhered to, we need a deeper discussion of possible sequences of calls. We require some notation to capture these sequences; we wish to track the receiver-method pairs from (consecutive) fragments of the call stack. We write $(c, m)$ for a call of static method $m$ on class $c$, and $(o, m)$ for a call of instance method $m$ on object $o$. For any receivers $r$,$r'$ (which may each be either classes or objects), we write $(r, m)\ \overline{call}\ (r', m')$ to denote a sequence of legal calls[11] beginning with $m$ and ending with $m'$, *i.e.*, method $m$ on receiver $r$ calls some method $m_1$ on some receiver $r_1$, etc., which eventually leads to calling method $m'$ on receiver $r'$. These sequences of calls correspond to consecutive regions of a call stack, in which only the receiver and method information is retained. Note that such sequences need not begin from the initial (class) receiver of a call stack. We consider only call-sequences which are legal according to our technique.

We can now show how calls are restricted by the effect annotations:

**Proposition 3 (Effects are Conservative).**

1. *For any call-sequence* $(o, m)\ \overline{call}\ (c', m')$*, if* $c$ *is the dynamic class of* $o$*, then* $c' \in \mathcal{E}ffs(c, m)$*.*
2. *For any call-sequence* $(c, m)\ \overline{call}\ (c', m')$*, if either* $c \neq c'$ *or any of the intermediate receivers in* $\overline{call}$ *are not* $c$*, then* $c' \in \mathcal{E}ffs(c, m)$*.*
3. *Any call-sequence* $(c, m)\ \overline{call}\ (c, m')$ *consists only of calls where* $c$ *is the receiver.*
4. *If* $o$ *and* $o'$ *are peers, then any call-chain* $(o, m)\ \overline{call}\ (o', m')$ *features only peers of* $o$ *(and* $o'$*) as receivers.*

Finally, we can prove that the invariants of a new receiver are always guaranteed by the proof obligations in the technique:

**Theorem 1.**

1. *If a static method* $m$ *is to be called on* $c$*, then the proof obligations imposed by the technique guarantee that* $c$*'s invariants hold.*
2. *If an instance method* $m$ *is to be called on* $o$*, then the proof obligations imposed by the technique guarantee that* $o$*'s invariants hold.*

## 5 Refined Effects

The effects as described so far require annotations for *all* classes used in a program. This requirement leads to a high annotation burden, compomises information hiding, and limits the usability of the technique presented so far, as the following example illustrates.

[11] *i.e.*, calls which are permissible according to Def. 2.

*Example 1 (Method Overriding and Effects).* Consider the String class of the Java API. An implementation of this class can exploit that fact that strings are immutable in Java, and so share instances of objects, by using static fields from class String to maintain a 'pool' of used String instances. This would imply that the constructor String calls String static methods, and would have String in its effects. Consider now that we want to write a class which overrides the equals() method inherited from Object:

```
class MyClass extends Object{
  boolean equals(Object o)
  {
    System.out. println (new String("equals()  called"));
    return this == o;
  }
}
```

Obviously, we need to have String$\in \mathcal{E}ffs$(MyClass,equals), and because of Def. 2 (item 3), we also need that String$\in \mathcal{E}ffs$(Object,equals). But, it is unlikely that this effect was predicted when the class Object was given effect annotations. Therefore, this method definition would be illegal. This illustrates an annotation problem (annotations may need recomputing), an information-hiding problem (our code should not need to know how String is implemented), and a usability problem (our technique forbids this method declaration).

To alleviate this burden, we introduce a refinement, whereby we group classes in a linear hierarchy of 'levels', such that the code of lower-level classes does not mention the higher-level classes[12]. The intuition is that library classes should have been previously verified and belong on a 'lower level' than the classes which the programmer is now writing. We express the levels through a function $\mathcal{L}vl(\_)$ which maps classes to integers.

**Definition 3 (Valid Levels).** *c mentions* $c' \Rightarrow \mathcal{L}vl(c) \geq \mathcal{L}vl(c')$.

Because classes in the lower levels do not 'know about' classes in the upper levels, it is impossible for them to make static calls on the classes in the upper levels (*cf.* Fig. 4). Therefore, if we consider verification of the topmost level, then when a call is made down to a lower level, the effect annotations are no longer necessary. [13] Thus, we refine our effect annotation sets to only mention classes on the same level as the method being verified. The new conditions on effects (in which differences in comparison with Def. 2 are shown in roman font) are:

**Definition 4 (Refined Effects).**

[12] For example, we could consider the Java API classes (*e.g.*, Object and String) to be on a lower level than our classes, and it would be naturally guaranteed that the API classes do not mention ours.

[13] To handle dynamic binding, we require the effects of methods that override methods in lower levels to be empty and, thus, independent of the effects of the overridden method.

1. If $c'$ is in $\mathcal{E}ffs(c, m)$ then $\mathcal{L}vl(c') = \mathcal{L}vl(c)$.
2. *Within the body of a method $m$ of class $c$, if there is a call $e.m'(\ldots)$ and $e$ has static type $c'$,* and $\mathcal{L}\mathbf{vl}(\mathbf{c}) = \mathcal{L}\mathbf{vl}(\mathbf{c}')$, *then* $\mathcal{E}ffs(c', m') \subseteq \mathcal{E}ffs(c, m)$.
3. *Within the body of a method $m$ of class $c$, if there is a call $c'.m'(\ldots)$ to a static method $m'$ of class $c'$* and $\mathcal{L}\mathbf{vl}(\mathbf{c}) = \mathcal{L}\mathbf{vl}(\mathbf{c}')$, *then:*
   (a) $\mathcal{E}ffs(c', m') \subseteq \mathcal{E}ffs(c, m)$
   (b) *if $m$ is either an instance method or $c \neq c'$, then $c' \in \mathcal{E}ffs(c, m)$.*
4. *If $c'$ is a subclass of $c$ which overrides a method $m$, then*
   (a) If $\mathcal{L}\mathbf{vl}(\mathbf{c}) = \mathcal{L}\mathbf{vl}(\mathbf{c}')$, then $\mathcal{E}ffs(c', m) \subseteq \mathcal{E}ffs(c, m)$
   (b) If $\mathcal{L}\mathbf{vl}(\mathbf{c}) < \mathcal{L}\mathbf{vl}(\mathbf{c}')$, then $\mathcal{E}\mathbf{ffs}(\mathbf{c}', \mathbf{m}) = \emptyset$
5. *A static method $m$ of $c$ is legal, only if $c \notin \mathcal{E}ffs(c, m)$.*

The refined conditions given permit smaller effects sets for methods than those of Def. 2. Considering the example at the start of the section, it is no longer necessary (or indeed, allowed) for String to be in $\mathcal{E}ffs$(MyClass,equals).

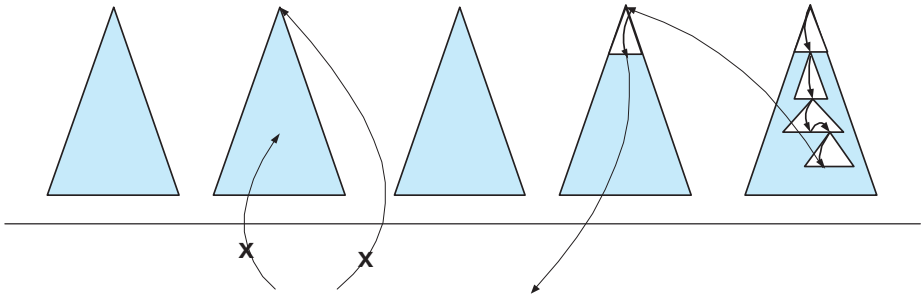

**Fig. 4.** Trees in one level. The current level may call into the lower level, but no calls from the lower level may come into the current level. The level of an object is determined by the class that transitively owns the object, not by the object's type.

**Soundness.** As in the previous section, we focus on ensuring that the proof obligations made before method calls are always sufficient to guarantee the expected invariants. Furthermore, we make the assumption here that we are only interested in verifying the 'top-level'; we assume that the classes on lower levels have already been verified. This can be used to construct an inductive verification of the entire class-structure, if needed, but also allows us a more-modular approach; once the classes on a lower level have been verified, we need not repeat the process if we are only adding classes to higher levels.

We write $\mathcal{L}vl(o)$ for the level of an object, defined to be the level of the class which transitively owns the object (*i.e.*, the class which is the 'root' of the appropriate tree). We can then show the following property:

**Proposition 4 (Levels do not Increase through Calls).**

1. *If object $o$ is transitively owned by class $c$, and if $c'$ is the dynamic class of $o$, then $\mathcal{L}vl(c) \geq \mathcal{L}vl(c')$.*
2. *For any call-sequence $(c, m)$ $\overline{call}$ $(o, m')$, where $\overline{call}$ consists exclusively of instance method calls, if $c'$ is the dynamic class of $o$, then $\mathcal{L}vl(c) \geq \mathcal{L}vl(c')$.*
3. *For any sequence of calls $(r_1, m_1)$ $\overline{call}$ $(r_2, m_2)$, in which $r_1, r_2$ can be any receivers,* i.e.*, classes or objects, we have $\mathcal{L}vl(r_1) \geq \mathcal{L}vl(r_2)$.*
4. *For any call-sequence $(c, m)$ $\overline{call}$ $(c, m')$, for all the intermediate receivers $r$, we have $\mathcal{L}vl(r) = \mathcal{L}vl(c)$.*

This allows us to construct similar arguments to those in the previous section, regarding soundness of method calls. Proposition 2 still holds for this refinement. Proposition 3 holds in the restricted case that all receivers involved are from the top-level. Theorem 1 then holds for all such receivers.

**Remarks.** We have allowed the organisation of levels to be very flexible, and thus the effects and levels can be used to complement each other in various different ways. Considering the extreme case of only one level, we return to our original effects proposal from the previous section, in which all the work must be done by the effects. On the other hand, if every class has a level to itself, we essentially impose a total ordering on classes (which may not be possible within our restrictions, for all programs), and no effect annotations are required at all. In practice, we envisage that the levels will be used to separate away previously written library classes from those being currently developed and verified.

## 6 Extended Technique

So far, static invariants cannot mention the fields of instance objects, and instance invariants cannot mention static fields. It seems reasonable to question whether this is enough. For example, if we wished to write a class MyThread in which each instance object was assigned a unique identifier id, we might like an invariant to express that distinct MyThread objects have different ids[14]. These kinds of invariants involve both static fields and instance fields. It is desirable to extend our technique to handle these more-expressive invariants. We could allow instance invariants to mention static fields (of the same class, and perhaps superclasses) in their invariants. The alternative approach is, instead of enriching instance invariants, to enrich *static* invariants with the ability to quantify over *all* instances of a class. In fact, any instance invariant mentioning static fields can always be expressed as a static invariant by adding a quantified object to replace all the mentions of **this**. However, enriching static invariants in this way can be more general if we allow multiple quantifiers. If we wanted to express the described invariant of MyThread, we could do so by the static invariant forall MyThread $o_1, o_2$: $o_1 \neq o_2 \Rightarrow o_1.\mathsf{id} \neq o_2.\mathsf{id}$. However, it is not clear how to express this at the level of an instance invariant (without quantifiers).

[14] This is an actual invariant of the Thread class in the Java API.

We choose to add the ability to quantify over fields of instances in static invariants. In static invariants of class $c$, if $o$ is a quantified object variable, the only fields of $o$ which may be mentioned in the invariants are those declared in class $c$. This restriction corresponds to the notion of *subclass separation* described for VT (see [10] for details).

*Remark.* Although it is true that any instance invariant mentioning static fields can be encoded as a static invariant quantifying over instances, this does not quite mean the two are interchangeable with respect to our technique. The reason is that although these invariants express the same properties, because one is an invariant per object, and one is an invariant of the class, they will be expected to hold at different times.

To work out exactly what changes were needed to our technique in order to retain soundness, we were guided by the soundness conditions of [2] (*cf.* Def. 1). Essentially, having made a change to our $\mathbb{D}$ parameter (by changing which invariants can depend on instance fields), the conditions presented there implied the minimal necessary changes to the other parameters of our technique in order to restore soundness. We highlight the differences between the new and the previous technique through the use of italics.

$\mathbb{X}$ invariants of objects (reflexively, transitively) owned by peers, plus all invariants in trees not currently visited on the call stack.

$\mathbb{V}$ invariants of all transitive owners of the current receiver, plus invariants of peers of the current receiver, *and their classes.*

$\mathbb{D}$ Invariants of peers and transitive owners may depend on the field of an object or class. *Additionally, static invariants of the class in which the field is declared.*

$\mathbb{B}$ Before making a method call, *the invariants of the classes of all of the peers of the current receiver must be established.* Furthermore, if the callee is a peer of the current receiver, then the invariants of all peers must be established.

$\mathbb{E}$ the invariants of all visible peers, *and their classes.*

$\mathbb{U}$ A field of an object (or class) may only be assigned to by its owner, or by any of its peers.

$\mathbb{C}$ A call to an instance method is allowed if the callee is a `peer` or `rep` of the current receiver. A call to a static method $m$ on class $c$ is allowed if either the current receiver is $c$ itself, or else $c$ is not on the current call stack.

**Soundness.** Informally, the soundness of this extended technique follows from the soundness of the previously-presented versions, as follows:

Proposition 2 no longer holds. Namely, because of the extended language of invariants in this new version of our technique, it is possible for many more methods to cause such invariants to break. However, our technique does *not* allow these invariants to remain broken in any more visible states than was previously allowed. Essentially, any invariants which are broken due to the quantification over instances now possible, will always be reestablished at the next visible state (either the end of the method call, or before the next method call; whichever is

the sooner). This is reflected in our $\mathbb{B}$ and $\mathbb{E}$ defined above. Therefore, although Proposition 2 does not hold, Theorem 1 can still be proved, essentially because enough extra proof obligations are imposed before a method call takes place.

## 7 Conclusions, Related Work, and Future Work

We have outlined a verification technique based on VT, catering for static fields, methods, and invariants. In the process, we extended the usual heap topology of ownership types, and tackled potential callbacks through a combination of effects, levels, and the OAM discipline.

Universe types as implemented in JML [5] require static fields to be readonly. JML's static invariants may only refer to static fields, while instance invariants may refer to both static and instance fields [6, Sec. 8.2]. In JML and in our work, both instance and static invariants are supposed to hold in visible states [10]. In JML's universe types, static methods are executed relative to the context of the object who called the static method. This allows one to implement static factory methods, which create new objects in the context of their caller. We can extend our approach to support factory methods by incorporating *ownership transfer* [11], allowing a method to create a new object, but to postpone the decision of assigning it an owner.

In [7], Leino and Müller extend the Boogie methodology [1] to static invariants: static fields may be reps; class invariants may mention static rep fields and also quantify over objects of their class. The callback problem is solved by making explicit the state in which static invariants may be assumed to hold, and by enclosing expressions that potentially break the static invariant of a class in expose blocks. In order to support abstraction in method specifications, a *validity ordering* is used to allow a class to implicitly expect the static invariants of 'smaller' classes. This issue is similar to one of the motivations for introducing our levels. The validity ordering, however, has the side-effect for static initialisation that subclasses be initialised before superclasses.

In Jacobs et al.'s work [4], Spec# annotations are suggested to cater for local reasoning in the presence of multithreading. Again, static fields may be reps, and static invariants may depend on the (transitively) owned objects. Both our system and theirs need to address potential circularities: ours in order to avoid visiting classes in an inconsistent state, and [4] in order to prevent deadlocks. They require a partial ordering of locks, which, in a way, corresponds to our levels. Two locks on the 'same level' are not allowed to be consecutively acquired. In contrast, we permit method calls between classes on the same level, if the effects allow it. Our work may be seen as the visible-states-based counterpart of [4, 7].

We have not discussed static initialisation in this paper. In brief, we expect to be able to incorporate the Java semantics for static initialisation. In terms of our topology, initialisation is best modelled by considering that the tree owned by a class comes into existence at the moment static initialisation of the class begins (and is initially empty, apart from the owning class). Static initialisers may

assume all of the invariants of lower levels, and no others (since the restrictions on method calls are not respected by the execution of static initialisers). Exploring these issues in more detail will be the subject of future work. We also plan to complete the formal presentation of our work, and to study class visibility, modularity, readonly fields, pure methods, and factory methods.

**Acknowledgements.** This paper has been greatly improved by comments and generous feedback from Adrian Francalanza and the (anonymous) FTfJP reviewers. We are also grateful to Nick Cameron, Werner Dietl, and Jayshan Raghunandan for discussions on static methods and verification. This work was funded in part by the IST-2005-015905 MOBIUS project.

## References

1. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, LNCS. Springer, 2005.
2. S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. A unified framework for verification techniques for object invariants. In *ECOOP*, 2008.
3. K. Huizing and R. Kuiper. Verification of object-oriented programs using class invariants. In *FASE*, volume 1783 of *LNCS*, pages 208–221. Springer, 2000.
4. B. Jacobs, J. Smans, F. Piessens, and W. Schulte. A simple sequential reasoning approach for sound modular verification of mainstream multithreaded programs. *Electronic Notes on Theoretical Computer Science special issue on Thread Verification (TV06),*, 174:23–47, 2007.
5. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML Reference Manual—section on Universe annotations, February 2007. `www.eecs.ucf.edu/~leavens/JML/jmlrefman/jmlrefman_18.html#SEC205`.
6. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML Reference Manual. Available from `http://www.jmlspecs.org`, May 2008.
7. K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In *Formal Methods*, 2005.
8. Y. Lu, J. Potter, and J. Xue. Object Invariants and Effects. In *ECOOP*, volume 4609 of *LNCS*, pages 202–226. Springer, 2007.
9. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, 2002.
10. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.
11. P. Müller and A. Rudich. Ownership transfer in Universe Types. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 461–478. ACM, 2007.
12. A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, 1997.