JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2008

Vol. 8, No. 4, Special Issue: Workshop on FTfJP and IWACO 2009

Universe-Type-Based Verification Techniques for Mutable Static Fields and Methods

A. J. Summers, Imperial College London
S. Drossopoulou, Imperial College London
P. Müller, ETH, Zürich

We present three novel techniques for the verification of invariants in the setting of Java-like languages including static fields and methods. Our techniques structure the heap through universe types, and extend the Visibility Technique of Müller et al.
In order to cater for mutable static fields, we extend the classical universe types heap topology with multiple trees, where each tree is rooted in a class. Thus classes may naturally own objects as static fields.
We present a basic version of our approach, which allows trees to be visited at the top and then navigated "downwards", and which avoids dangerous call-backs through effects which track static method calls. As well as the usual kinds of proof obligations defining that certain invariants must hold at a given state, we employ a second kind of obligation to show that certain other invariants are *preserved* between two states (i.e., *if* they hold in the former state then they will still hold in the latter). This allows us to deal with invariants whose expected truth-value cannot always be determined statically in a modular way.
We then present two extensions of our basic technique, aimed at improving usability. Firstly, we introduce a new universe annotation to allow safe callbacks between trees, whereby trees may be visited not at the top, but at the point where a previous visit "had left off". Secondly, we refine our heap topology with a notion of 'levels', which stratify the heap and provide modularity with regard to library classes and the required effects annotations.

1 INTRODUCTION

We propose three novel techniques for the verification of invariants in the setting of Java-like languages including static fields and methods. The inclusion of static fields leads naturally to a notion of *static invariants*, which belong to a class rather than its instances. Our techniques extend the Visibility Technique (hereafter, VT) of Müller et al. [11].

Visible states verification techniques permit certain invariants to be temporarily broken during the execution of a method. These invariants are expected to hold at the initial and final states of the method: the *visible states*[1]. VT is one such

[1]In the previous work of Müller et al. [11], the terminology "pre-state" and "post-state" is used. We use these instead to refer to the states immediately before and after a method is called (i.e., states of the caller), while we use "initial" and "final" to refer to the states immediately after

technique, and uses *universe types* [1, 10] to hierarchically structure the heap. With universe types, each object is either unowned or owned by another object, its *owner*. In this paper, we assume that the unowned objects are owned by a designated `root` entity, such that the overall topology is a tree. In VT, an object's invariants may only depend on the fields of the objects it owns, and those of its *peers* (those with the same owner[2]). An object may only modify the fields of its peers, and may only call methods of the objects it owns, and those of its peers (cf. Fig. 2 on page 92). Thus *call–backs*, i.e., the possibility that while an object is executing a method it (indirectly) calls a further method on itself, are only possible between peers. VT requires method calls on peers to be preceded by a proof obligation establishing the peers' (and thus also the receiver's own) invariants; the invariants of an object are guaranteed to hold upon all entries to its method calls, including call-backs.

Incorporating statics, which are not part of VT, raises 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?

We generalised the VT heap topology, and handle *class objects* in a natural way, so that they can own objects referenced by their static fields. Since classes do not naturally have owners themselves, we argue that the topology is naturally divided into many trees, each one 'rooted' by a class. In order to make static methods usable, we allow static method calls to be made from (inside) one tree to the top of another. Thus, method calls may enter trees from the top, then navigate downwards, and may then visit other trees, again entering from the top and navigating downwards (cf. Fig. 3 on page 100). This feature introduces a new risk of dangerous callbacks not present in VT—this time across classes, since a tree may be 'reentered' while a method call on a receiver in the tree is already taking place.

In our basic technique, "VT with Statics" (VTS hereafter), we avoid such call-backs by restricting method calls through an effects system which conservatively approximates which classes may be (indirectly) visited during a method execution. Furthermore, we extend the notion of proving invariants with that of *preserving*

the method is called, and before the method returns (cf. Definition 1).

[2]Note that an object is itself one of its peers.

invariants. This is because we allow static invariants to depend on the fields of (all) instances of a class: since instance methods may alter fields of such instances, it is possible for instance methods to break static invariants. Rather than require such methods to establish that the appropriate static invariants actually hold (which is generally unfeasible, since an instance method will not generally have knowledge of all other instances of the same class), we require these methods to *preserve* static invariants, i.e., to prove that if the static invariant held at the initial state of a method execution, it will still hold at the final state. We developed a detailed proof of soundness for VTS.

For our design we used an idea of 'filtering': our aim was that filtering a stack of VTS method calls by considering only the calls within a particular tree, should result into a verification effort which would 'look like' that of VT (which is already known to be sound for a heap topology based on a single tree). This idea guided both our design of VTS, and the structure of our soundness proof.

Filtering also led us to "Strong VTS" (SVTS hereafter), an extension of VTS which is a more permissive discipline with respect to calls between trees. To this end, we introduced the new type-annotation *strong*, which expresses the permission to call back into the middle of a previously-visited tree, at a location that could have been visited from the last method call in that tree, (cf. Fig. 5 on page 111). The resulting technique is motivated by practical considerations, and admits many more examples. For example, when one passes an `Object` to a method call `System.out.println(o)`, it may be that the method implementation wishes to callback a `toString()` method on o. Examples similar to this can be handled in SVTS because of the ability to allow callbacks into previously-visited trees.

We also developed "layered VTS" (LVTS herafter), an alternative extension of VTS, which stratifies the heap topology using 'levels', intended to reflect a layered software architecture, and corresponding verification effort. In particular, we aim to abstract away from the details of previously written classes (such as library classes). The key observation is that library classes will never (directly) call static methods on newer classes being verified, and so the possibility of dangerous callbacks is naturally reduced when calling classes on a 'lower level', (cf. Fig. 6 on page 119). This refinement allows for fewer (and more modular) effect annotations.

Unfortunately, the combination SVTS and LVTS seems to be possible only in a limited form. Because the extra call-backs allowed by SVTS make it possible to call from trees on lower levels to trees on higher levels, the refined effect annotations employed in LVTS are no longer sufficient to predict all static method calls, and this leads to potential unsoundness. We identify a restriction which restores soundness, but at the cost of applicability to some examples.

This paper extends our preliminary work [15], presented at FTfJP 2008. The VTS technique is similar to the "Basic Technique" in [15], but with quantified static invariants, improvements to the effects annotations which allow more static method calls between classes, and the concept of preservation. The LVTS technique is similar to the "Extended Technique" in [15], but again with the improvements mentioned for VTS. The SVTS technique introduces the concept of strong references, which is totally new.

Furthermore, the soundness proofs required the development of some exciting formal techniques. In joint work with Francalanza [2] we have presented a generic framework for describing existing visible state verification techniques, and for easily obtaining soundness results for such techniques. However, dealing with the preservation (rather than establishing) of invariants makes it impossible to directly adopt the machinery of [2], and thus we developed our own proofs which reflected the filtering idea, and which builds on the existing soundness result for VT. In order to structure our arguments, we also developed the concept of *semi visible state* for a method execution (which identify important mid-states of the method execution), and the concept of *losing* (i.e., invalidating) invariants between such semi-visible states. Several proofs run by induction on the number of indirect method calls (thus reflecting the complete depth of execution), and on the number of semi-visible states (thus reflecting the breadth of execution).

In Sec. 2 we give the background to visible states verification techniques, universe types, and VT. In Sec. 3 we present the extended heap topology and static invariants. In Sec. 4 we present VTS, and prove that it is sound. In Sec. 5 we present the extensions to SVTS and LVTS. In Sec. 6 we discuss related work and conclude.

2 BACKGROUND

In this section, we give a brief introduction to the concepts on which our work is based, particularly the Visibility Technique.

Visible States

Visible states verification techniques are defined around the notion of *visible states*, which correspond to the initial and final states of each method execution. As mentioned in the introduction, we make use of slightly different terminology for visible states than in the literature, as our soundness arguments rely on being able to distinguish the point in execution just before calling a method, from the point at the beginning of method execution, and similarly to distinguish the point of returning from a method body from the point immediately after the call. This is made formal by the following definition:

Definition 1 (Visible states terminology) *During execution of a method body m, and considering a call to a further method m', we use the following terminology with respect to the call to m':*

pre-state *refers to the point in execution of the body of m immediately before the call to m' is executed.*

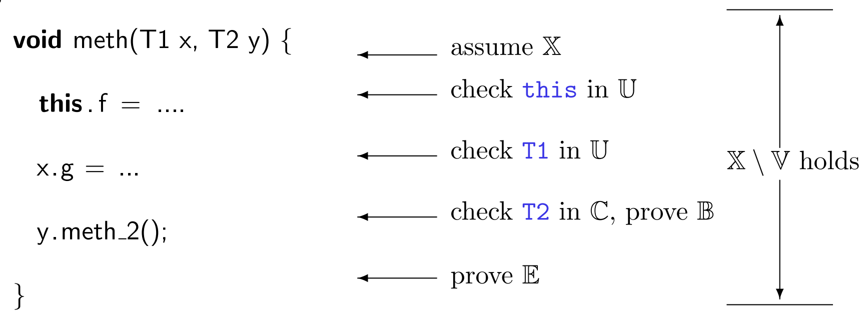
initial state *refers to the point in execution of the body of m' just as it is starting to be executed.*

final state *refers to the point in execution of the body of m' just before control returns from the call.*

post-state *refers to the point in execution of the body of m immediately after the call to m' is executed.*

The visible states *of a call to a method m' are the initial and final states of the 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.



```
void meth(T1 x, T2 y) {

  this.f = ....

  x.g = ...

  y.meth_2();

}
```

assume $\mathbb{X}$
check this in $\mathbb{U}$
check T1 in $\mathbb{U}$
check T2 in $\mathbb{C}$, prove $\mathbb{B}$
prove $\mathbb{E}$
$\mathbb{X} \setminus \mathbb{V}$ holds

Figure 1: Illustration of the use of the seven components.

Several visible states techniques have been suggested, e.g., [13, 3, 11, 8], and they share many commonalities. As suggested in [2], these commonalities, as well as the differences, can be 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[3].

$\mathbb{B}$ 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.

[3]By defining the invariants which may depend on a location, this component also characterises indirectly the locations an invariant may depend on.

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 initial and final 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 is made (i.e., in the pre-state of the method call), $\mathbb{B}$ must be proven. At the end of a method execution (i.e., in the final state of the method execution), $\mathbb{E}$ must be proven. Finally, assignments to `this.f` and `x.g` affect at most the corresponding $\mathbb{D}$.

Universe Types and the Visibility Technique

One visible states technique, the Visibility Technique (VT), was developed on top of universe types [11] with the aim to guide the verification process and to guarantee modularity. Universe types [10] organise the heap into a tree topology, in which each object is *owned* by another object. *Universe modifiers* describe the relative position in the heap topology of one object with respect to another. For example, an object o considers another object o', as its peer if they have the same direct owner. An object o considers o' its rep if o is the direct owner of o'. The modifier rep stands for 'representation', and the intention is that the objects owned by an object make up its "inner working" or representation. In particular, object invariants are restricted to only depend on objects transitively owned, or peer objects. In order to make verification of invariants depending on peer objects feasible, a notion of *visibility* between classes is employed [11]. We ignore the consideration of visibility in this paper, since they are orthogonal to the issues we address. In order to incorporate visibility, one needs only to read all occurrences of 'peer' in our definitions as 'visible peer'.

In VT, reference types carry *universe annotations* (such as rep and peer), specifying the intended topology. A third universe annotation, any, can be used to denote that no topological constraint is made; such a reference may point to an object at an arbitrary position in the heap topology.

Definition 2 (Contexts) *For any receiver*[4] r*, we use* the context of r *to mean* r *itself plus all peers of* r *and all objects which are transitively owned by peers of* r*. We write* invariants within the context of r *to mean all the invariants of receivers in the context of* r*.*

The *owner-as-modifier discipline* restricts field updates and method calls, such that the fields of an object may only be modified when its owner is on the call stack[5] (intuitively, its owner "knows" about the modification). This is enforced by

[4]In the setting of VT, receivers are just objects. However, we will employ the same definition later, when we will allow both classes and objects to be receivers.

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

requiring that the receiver of a new method call is always a rep or peer of the current receiver. Such a call stack is illustrated in Fig. 2 in page 92; note that calls may only go "down" or "sideways". To control modifications, the fields of an object are only allowed to be updated by one of its peers (note that an object regards itself as a peer). Given these restrictions, VT imposes sufficient proof obligations to ensure that whenever a method is called, the invariants of the new receiver and all objects which the new receiver may (transitively) make calls on, are guaranteed to hold.

A more liberal policy is employed in the case of *pure methods* (i.e., methods without side-effects - see [14] for a rigorous treatment). Since such methods cannot make changes to the heap, it is clear that they cannot violate any invariants, and so the rigorous discipline described above is not needed. For this reason, pure methods may, in principle, be called on any references. However, even pure methods may *depend* on certain invariants holding for their proper behaviour, and so it is only safe to call even these if it can somehow be known that the new receiver is within the context of the current receiver (and therefore some knowledge about the invariants it may require to hold is available). This kind of information can usually only be demonstrated by some means external to the type system.

The seven components from before have the following meaning for VT:

Definition 3 (The Visibility Technique) *With the exception of $\mathbb{D}$ and $\mathbb{U}$, these parameters are considered with respect to a current method execution, say a method m on receiver r.*

$\mathbb{X}$ *The invariants of receivers within the context of r are expected.*

$\mathbb{V}$ *The invariants of all transitive owners of r are vulnerable, plus invariants of peers of r.*

$\mathbb{D}$ *The invariant of a receiver r may depend on the fields of a receiver r' only if r is a peer or transitive owner of r'.*

$\mathbb{B}$ *In the pre-state of a method call, if the callee is a peer of r, the invariants of all peers must be established to hold.*

$\mathbb{E}$ *The invariants of all peers of r must be established to hold in the final state of a method execution.*

$\mathbb{U}$ *A field of an object may only be assigned to by its peers (i.e., when one of its peers is the current receiver).*

$\mathbb{C}$ *A call to another instance method is allowed if the callee is a* peer *or* rep *of r, or if the method is pure and the callee is known to be within the context of r.*

It can be shown that these parameters satisfy the soundness conditions presented in [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.

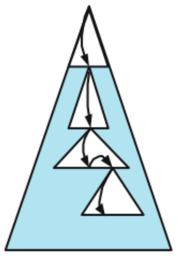


Figure 2: Ownership Tree and Control Flow; the arrows show consecutive method calls and their receivers; note that calls go only "down", i.e., to reps, or "sideways", i.e., to peers. The shaded area indicates the area where objects satisfy their invariants.

Lemma 4 (Soundness of VT) *For any method execution (r, m) in a program verified by VT, if the expected invariants $\mathbb{X}$ hold at the initial state of the method execution, then:*

1. *At the pre-state of every direct call (r', m') made during execution of the method, the corresponding expected invariants $\mathbb{X}_{(r',m')}$ hold.*
2. *At the final state of the method execution (r, m), the expected invariants $\mathbb{X}_{(r,m)}$ hold.*

3 HEAP TOPOLOGY FOR STATIC FIELDS AND STATIC INVARIANTS

In this section, we introduce the basic principles of our work, including a heap topology which includes representations for classes, and a discussion of static invariants.

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. Such fields would complicate the techniques we present here, since we depend on the "roots" of trees in our topology being fixed. If we were to allow (mutable) static peer fields, this assumption would not hold, and the techniques we provide would need to be extended. However, we have not seen any practical need for code with static peer fields, and so we do not regard this to be an important issue in this paper.

In most respects we interpret static fields and methods as if they were 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. Executions of static methods may update the fields of their receiver class, just like instance methods in VT may update fields of their receiver object. We will use the terminology "receiver" to refer to either a class (in the case of a static method) or an object (in the case of an instance method).

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. Receivers (on the stack) can be either an object or a class.

Static Invariants

Since we propose that classes may own objects, it is natural to permit classes to have invariants, describing the consistent states of those objects, just as objects may describe the consistency of their owned objects using invariants. We call invariants which belong to the class, rather than the instances of the class, "static invariants"[6]. As an example of the use of such invariants, consider the following code:

```
class Person {
  static rep List<any Person> population = new rep Vector<any Person>();
  static double temperature = 15.0;

  any String name;

  //@ static invariant temperature == 45 − 30/(1+population.size())

  public Person(any String name)
  {
    this.name = name;
    population.add(this); // temporarily violates static invariant
    temperature = 45 − 30/(1+population.size()); // reestablishes invariant
  }

}
```

[6]Note that the term "class invariant" is used in the literature inconsistently with our intention—it is often used synonymously with "object invariant".

The static invariant describes a temperature which ranges from 15 to 45, and increases as the population increases in size[7]. This invariant is temporarily broken during the execution of the constructor, since a new person is added to the list before the temperature can be recalculated. However, the static invariant can be reestablished by the end of the constructor, exactly as the visible states semantics requires.

As well as the natural extension of ownership-based invariants to the case of static fields, it is interesting to consider invariants which describe properties over all instances of a class. 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`[8]. These kinds of invariants can involve both static fields and instance fields. We could also consider adding to the example above, an invariant to express that all instances of `Person` are contained within the list `population`.

It is desirable for 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`:` $o_1 \neq o_2 \Rightarrow o_1$`.id`$\neq o_2$`.id`. However, it is not clear how to express this at the level of an instance invariant (without quantifiers).

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 [11] 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 possibilities 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. For example, a static invariant of the form $\forall o{:}c,\ p(o)$ will be required to be preserved by all method calls, in our technique. However, the corresponding instance invariant $p(o)$ in class c is allowed to be temporarily broken for a

[7]The static fields can be thought of as belonging to the 'world' in which the people live, and as such, describe state common to all of the people.

[8]This is an actual property of the `Thread` class in the Java API, except in the case where identifiers are manually specified.

particular object o while methods are executing on, for example, objects owned by o. On the other hand, the static invariant $\forall o$:`Singleton`, $o ==$ `Singleton.instance` could be broken while a static method of `Singleton` is executing (perhaps some extra instances are temporarily created, for debugging purposes), while the corresponding instance invariant $o ==$ `Singleton.instance`, enforced per object, would require (for a verified program) that no other instances *ever* exist (since these objects' invariants could never be established).

4 THE BASIC TECHNIQUE (VTS)

In this section, we present our basic technique, "VT with Statics" (VTS hereafter), in which we allow for static fields, methods and invariants, and avoid dangerous call-backs via static methods by restricting method calls through an effects system. Furthermore, we extend the notion of proving invariants with that of *preserving* invariants. This is because we allow static invariants to depend on the fields of (all) instances of a class: since instance methods may alter fields of such instances, it is possible for instance methods to break static invariants. Rather than require such methods to establish that the appropriate static invariants actually hold (which is generally unfeasible, since an instance method will not generally have knowledge of all other instances of the same class), we require these methods to *preserve* static invariants, i.e., to prove that if the class invariant held in the initial state of a method execution, it will still hold at the final state. We will present a detailed proof of soundness for VTS.

Having defined a suitable heap topology and notion of invariants, in this section we generalise VT to our setting, defining a new technique, VTS. In order to describe our ideas, we require a notation to discuss sequences of legal calls in our technique. For reasons which will become apparent later in this section, we find it useful to employ a notion of call stack which records the current receiver (which may be either an object or a class), and the current method name. We omit an explicit treatment of addressing and the tracking of method parameters, since they are orthogonal to the issues we address here.

Definition 5 (Receivers and call stacks) *A* receiver r *is either an object o (in the case of an instance method call) or a class c (in the case of a static method call).*

A call stack σ *is a sequence of* stack frames, *each of which is a pair of receiver and method-name (r, m) (representing an active call to receiver r of method m). We write ϵ for an empty stack. Stacks are formally defined by:*

$$\sigma \;=\; \epsilon \;\mid\; \sigma\circ(r, m)$$

We write $\sigma_1+\sigma_2$ to denote concatenating two stacks, in the obvious way.

The most important aspect of the design of our technique can be understood as follows. We understand how VT guarantees soundness using a heap topology with only one tree. We wish to make use of our more general topology with many trees, but in such a way that we maintain the essential discipline of VT. Therefore, we aim to ensure that the behaviour of our technique as regards each individual tree is analogous with what VT prescribes. In other words, when one examines only a single tree, our technique should 'look like' VT. We refer to this design principle as 'filtering', since we consider legal sequences of method calls after they are filtered to only include those receivers in a particular tree. In particular, we designed VTS to be a proper generalisation of VT, in the sense that if one only ever visits one tree in a particular program, the resulting verification effort is exactly that specified by VT. We will bear this principle in mind throughout the rest of this paper. It seems reasonable to specify that instance method calls should be restricted in exactly the same way as in VT (e.g., a call to a method which is not pure may only be made on a peer or rep receiver).

How then, to handle static method calls? According to VT, a method call is only allowed if the caller 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. Instead, we consider the implications of our 'filtering' idea. According to this idea, we should allow calls to static methods exactly when the resulting permitted stacks, when filtered by each individual tree, would yield permitted VT stacks. This means that there is no problem with calling a static method on a class whose tree has not yet been visited on the call-stack, since, when filtered, this will 'look like' a call-stack which begins at the root of the tree (as all stacks do in VT). Furthermore, it is acceptable for a tree to be revisited (re-entered) by a static method call, so long as no instance methods have been called (otherwise, since the tree will be entered back at the root, the filtered stack will 'jump' back upwards), and so long as the invariants of the class have been preserved (this then 'looks like' a call from the class at the root of the tree to itself, when filtered).

However, consider the following situation. Suppose we start off with a class c as receiver of a (static) method execution. c then calls an instance method on an owned (rep) object o. By some mid-point of this method's execution, it may be that the invariants of both c and o are broken (as is permitted by visible states). Suppose a further subcall is made to a static method of a class c'. Now, if it is either the case that $c = c'$, or that a further sequence of subcalls results in a callback to c, we have a problem; the invariants of c, and objects owned by c have been left broken, and so it is not safe to revisit c's tree. This callback problem can also be understood from the point of view of filtering: in the problematic cases described, the filtered stack concerning only the tree of c will involve a sequence of receivers c, o, c, and the implicit call from o back up to c is not permitted by VT, precisely because the expected invariants are not guaranteed to hold in this case.

We are led to the following two conclusions, regarding the executions of static methods (on a class c, say). Firstly, if a subcall is made to an *instance* method of an object, and this subcall may eventually result in a callback to a static method on the same class c, we must forbid it (due to the invariants remaining broken in c's tree). Secondly, if an execution of a static method of c makes a subcall to a *static* method of a class c', and this subcall may eventually (or immediately, in the case $c' = c$) result in a callback to a static method on the same class c, we must ensure that the invariants of the class c are reestablished before the call is made (so that these invariants hold if and when the callback is made).

In both scenarios, we require the ability to predict whether a method call may eventually result in a call to a class c. Since this information cannot be inferred modularly at compile-time, we instead approximate the required information using effect annotations.

Effect Annotations.

For each class c and method m, we require a set of *effects*, $\mathcal{E}\mathit{ffs}(c, m)$, predicting which classes may have static methods called on them as a result of calling m of c. $\mathcal{E}\mathit{ffs}(c, m)$ is a (possibly empty) set of class names. This is described by requirements 1-3 in Def. 6 below.

Definition 6 (Effect Annotations)

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 $\mathcal{E}\mathit{ffs}(c', m') \subseteq \mathcal{E}\mathit{ffs}(c, m)$.*
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) *$\mathcal{E}\mathit{ffs}(c', m') \subseteq \mathcal{E}\mathit{ffs}(c, m)$ and*
   (b) *$c' \in \mathcal{E}\mathit{ffs}(c, m)$.*
3. *If c' is a subclass of c which overrides a method m, then $\mathcal{E}\mathit{ffs}(c', m) \subseteq \mathcal{E}\mathit{ffs}(c, m)$.*

For convenience, we will allow ourselves to write $\mathcal{E}\mathit{ffs}(o, m)$ to mean $\mathcal{E}\mathit{ffs}(c, m)$ where c is the dynamic class of o. This means in particular that $\mathcal{E}\mathit{ffs}(r, m)$ is defined for any receiver r which had a method m.

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. 6, we must have $c \in \mathcal{E}\mathit{ffs}(c', m')$. Therefore, we can rule out dangerous callbacks on c by insisting that any instance method which is called from a static method of c does not contain c in its effects. Furthermore, if, from a static method of c, a further static method is called which has c in its effects, then the invariants of c must be reestablished before the call is made.

Note that the third criterion above presents issues for subclassing: in order to retain modularity, one cannot override a superclass method with a definition which has extra effects. This seems potentially quite a strong restriction, since (at least) one might consider using static methods of different library classes in a new definition. This problem is addressed in Section 5.

We depend on the following property for the soundness of VTS:

Definition 7 (System invariant for VTS) *For any method (r, m), if there is a possible execution of the method which results (possibly indirectly) in a call to a static method (c, m'), then $c \in \mathcal{E}ffs(r, m)$.*

This essential property states that the effect annotations conservatively predict static method calls. Such predictions must be 'propagated backwards' through the sequence of calls. The following result formalises these facts.

Proposition 8 (Effects are propagated and conservative)

1. *For any permitted sequence of method calls $\sigma \circ (r, m)$, and for all $(r', m') \in \sigma$, we have $\mathcal{E}ffs(r, m) \subseteq \mathcal{E}ffs(r', m')$.*
2. *For any permitted sequence of method calls $\sigma \circ (c, m)$, and for all $(r', m') \in \sigma$, we have $c \in \mathcal{E}ffs(r', m')$.*

Proof 9

1. *By straightforward induction on the definition of σ, using Definition 6 above.*
2. *By straightforward induction on the definition of σ, using part 1 and Definition 6.*

Preserving quantified invariants

Our decision to allow static invariants to quantify over all instances of the same class means that additional proof obligations are required in order to preserve these invariants. Any method which modifies the field of an object may potentially break the invariants of the class of the object, or any of its superclasses. Since objects can be modified by their peers, this means that any instance method execution on an object o can potentially break the invariants of the classes and superclasses of the peers of o. It is however difficult from the point of view of an instance method execution on o to determine statically which of these static invariants are *actually* expected during the method execution. For example, if o happens to be owned by class c (and therefore the instance call on o is the (possibly indirect) result of a static method call on c), the invariants of c may reasonably be broken for the entire duration of the instance method execution. However, o is not naturally aware of its owner, and so is unable to tell whether the invariant of c is expected.

Since it cannot be statically determined whether a particular static invariant is expected for an instance method execution, a conservative approach is to insist that the static invariants which are vulnerable to an instance method execution are *preserved* by the execution; that is, it is required to prove that *if* such an invariant holds in the initial state of the instance method, it is guaranteed to hold in the final state. This is a different kind of proof obligation from those considered in [2], and indeed, in most verification techniques based on invariants. The formal notions of the seven parameters employed in [2] are not sufficient to express proof obligations regarding *preserving* rather than *asserting* invariants, nor to cover the possibility that the precise expected invariants for a method execution may not be known statically[9]. In Sec. 4 we will give a proof for our technique which reflects this idea of preserving invariants, where appropriate. We also discuss a concept of *filtering*, which is an attempt to map the soundness issues for our complex technique back to the soundness issues in VT. However, this attempt is only partial: the issue of preserving invariants remains essential to our work, and without a significant extension to [2] this makes the generic soundness proof of that paper impossible to make use of. Nonetheless, we believe that the seven parameters described in that paper neatly characterise the essential aspects of a verification technique, and we will employ the same seven concepts (in a purely illustrative role) here, in order to structure our definitions.

Definition 10 (VTS) *We highlight the main differences with VT (cf. Definition 3) in* **bold** *below. As previously, these parameters are considered with respect to a current method execution* (r, m)*.*

$\mathbb{X}$ *The invariants of receivers within the context of* r *are expected,* **plus invariants within the context of all classes** c **such that** $c \in \mathcal{E}ffs(r, m)$.

$\mathbb{V}$ *The invariants of all transitive owners of* r *are vulnerable, plus invariants of peers of* r*,* **and the static invariants of their classes and superclasses**.

$\mathbb{D}$ *The invariant of a receiver* r *may depend on the fields of a receiver* r' *only if* **either** r *is a peer or transitive owner of* r' **or** r' **is an object and** r **is the class of** r'**.**

$\mathbb{B}$ *In the pre-state of a method call, if the callee is a peer of* r*, the invariants of all peers must be established to hold.* **Similarly, if the caller is a class** c **and the new call is to a static method with** c **in its effects, the invariants of** c **must be established. Furthermore (in all cases), the invariants of the (super)classes of all of the peers of the caller receiver must be shown to be preserved (since the initial state of the caller's method body).**

[9]Although this kind of proof obligation is unusual compared with the existing literature, it has been suggested in the recent work of Middelkoop et. al. [9].

$\mathbb{E}$ *The invariants of all peers of r must be established to hold in the final state of a method execution.* **Furthermore, the invariants of the (super)classes of all of the peers of the current receiver must be shown to be preserved (since the initial state of the method execution).**

$\mathbb{U}$ *A field of an object may only be assigned to by its peers (i.e., when one of its peers is the current receiver).* **A (static) field of a class may be assigned to when the current receiver has a peer of that class (or a subclass of that class).**

$\mathbb{C}$ *From an instance method body (i.e., r is an object), a call to another instance method is allowed if the callee is a* peer *or* rep *of r, or if the method is pure and the callee is known to be within the context of r.* **From an instance method body, a call to a static method is always allowed. From a static method body (r is a class c), a call to a static method (of any class) is always allowed, while a call to an instance method is allowed if c is not in the effects of the method, and either the callee is a rep of c or the method is pure and the callee is known to be within the context of r.**

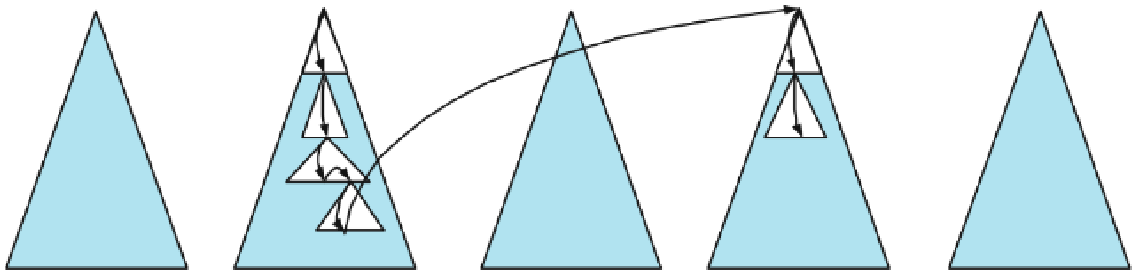


Figure 3: Calls stacks across several trees, invariants hold in shaded areas.

Soundness.

Throughout this section (and particularly in the statement of our results), we assume that all methods have been successfully verified by VTS (i.e., all of the static proof obligations have been shown). In order to aid our discussions later, we consider the concept of a *filtered call stack*: given a call-stack and a specific tree in the topology, the corresponding filtered call-stack is the sequence of frames obtained by deleting all those with receivers which are not in the tree.

Definition 11 (Roots and filtered stacks) *Define the* root *of a receiver as follows: For any class c, define $root(c) = c$. For any object o with $owner(o) = r$, define $root(o) = root(r)$.*

Now, define the filtering of a stack by a tree ('rooted' at class c) recursively, by:

$$\begin{array}{rcl} \mathit{filter}(\epsilon, c) & = & \epsilon \\ \mathit{filter}(\sigma \circ (r, m), c) & = & \left\{ \begin{array}{ll} \sigma' \circ (r, m) & \mathit{root}(r) = c \\ \sigma' & \mathit{otherwise} \end{array} \right\} \mathit{where}\ \sigma' = \mathit{filter}(\sigma, c) \end{array}$$

We can show the following result, which states that if a class c' is in the effects of the currently-executing method, then the current stack either contains no receivers from the tree of c', or at most c' itself. In particular, once an object from the tree of c' is on the stack, it is impossible that c' is in the effects of the currently-executing method.

Lemma 12 (Effects and stacks) *For any call-stack $\sigma \circ (c, m)$ resulting from a permitted sequence of method calls, if $c' \in \mathcal{E}ffs(c, m)$ then the only receiver (if any) occurring in frames in filter$(\sigma \circ (c, m), c')$ is c' itself.*

Proof 13

By straightforward induction on the definition of σ, using Definition 6. The proof depends on the fact that no objects from the tree of c' can be receivers on the stack unless c' precedes them as a receiver on the stack. In this case, the restrictions of Definition 6 prevent any instance method from being called which still has c' in its effects. Therefore, no such instance methods can be called, by Proposition 8.

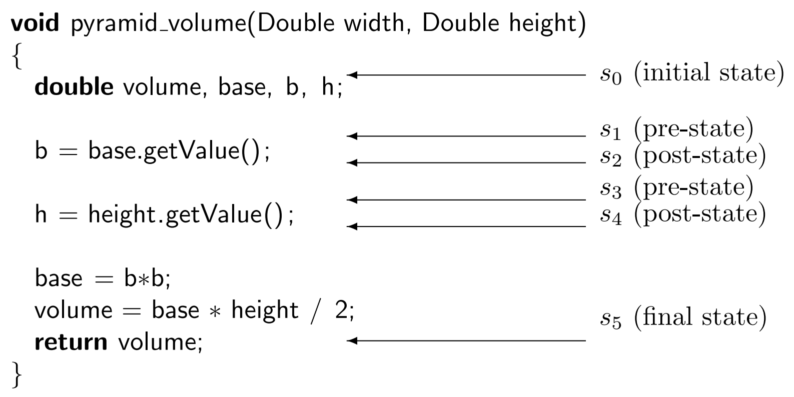
We also find it convenient to describe significant 'mid-points' of a method execution. In order to reason about soundness for subcalls, it is useful to consider the pre- and post-states of these, along with the initial and final states of the method body.

Definition 14 (Semi-visible states) *For any method execution (r, m), the* semi-visible states[10] *of the method execution are each of the following states which are reached during the execution:*

1. *the initial state of the method execution*
2. *the pre- and post-states of each direct subcall made during the method execution*
3. *the final state of the method execution*

Note that, if the semi-visible states are numbered from zero, then between any odd semi-visible state which is not the final state and the subsequent semi-visible state, a direct subcall (and no other execution) is made. Between any even semi-visible state and the subsequent one, no subcalls are made (control remains solely with receiver r). This is illustrated in Figure 4.

[10]Note that the initial and final states are themselves visible states in our terminology, whereas the pre- and post-states of subcalls immediately precede or follow visible states (initial and final) of the called method, respectively.



```
void pyramid_volume(Double width, Double height)
{
  double volume, base, b, h;

  b = base.getValue();

  h = height.getValue();

  base = b*b;
  volume = base * height / 2;
  return volume;
}
```

Figure 4: Semi-visible states.

Note also that semi-visible states are a runtime notion, defined with respect to a particular method execution; in general it cannot be known statically what the semi-visible states of a particular method execution will be at runtime, since the particular branches taken and termination of the method cannot be known. However, they are used in our reasoning about soundness with respect to actual executions of programs.

Definition 15 (Lost invariants) *For any method execution and any two semi-visible states of the execution s_i and s_j (where $i \leq j$), the* invariants lost between s_i and s_j *are those invariants which were true at s_i and no longer hold at s_j.*

Before we tackle the main soundness result, it is useful to state a lemma describing limits on the invariants which could be lost while execution remains with a particular receiver (i.e., no subcalls are made). In fact, we consider execution between two consecutive semi-visible states of a method execution during which no subcall is made. In this case, the invariants of objects which are reps and peers of the current receiver may be violated (lost), just as in VT. Because static invariants may also depend on the fields of such objects, in principle one might expect these to be potentially lost. However, VTS imposes strong proof obligations about these invariants, guaranteeing essentially that none are lost since the initial state of the method execution. When considering invariants lost between two consecutive semi-visible states s_i and s_{i+1}, there is still one obscure case to consider: it could be that a static invariant which did not hold at the initial state happens to have been made true by state s_i. In this case, the proof obligations say nothing about this static invariant (which did not hold at the initial state), and so it may in fact be lost between s_i and s_{i+1}.

Lemma 16 *For any method execution (r, m), let s_i and s_{i+1} be any consecutive semi-visible states of the method execution such that i is even (in which case, there are no calls made between these states; control remains with receiver r). Then the only invariants lost between these states are either invariants of peers and transitive owners of r, or static invariants of classes of peers of r which did not hold at the initial state of the method execution.*

Proof 17 *Between these states, the only fields which can be updated ($\mathbb{U}$) are fields of peers of r and static fields of classes (and superclasses) of peers of r. Therefore, the only invariants ($\mathbb{D}$) which may potentially be lost between s_i and s_{i+1} are those of peers and transitive owners of r, and those of classes of peers of r. However, our technique imposes a proof obligation at semi-visible state s_{i+1} ($\mathbb{B}$ or $\mathbb{E}$ depending on whether s_{i+1} is the pre-state of a subcall or the final state of the method execution) that no invariants of classes of peers of r are lost between s_0 and s_{i+1}.*

We can now show a more general result, which states that, for a sequence of method calls to break an invariant from a certain tree, at least one method call must have a receiver in the same tree. This is so essentially because the only invariants vulnerable 'across trees' are (quantified) static invariants, and the proof obligations of VTS ensure that these are preserved during execution.

Proposition 18 (Trees must be called to be broken) *For any method execution (r, m), let $s_0, s_1, \ldots$ be the semi-visible states of the execution. Then, for any class c and for any semi-visible state s_i of the execution, if during execution between s_0 and s_i, control never reaches a receiver r' with root$(r') = c$, then none of the invariants within the context of c are lost between s_0 and s_i.*

Proof 19 *By induction on the number, j, of (direct or indirect/transitive) method calls made between s_0 and s_i (note that this number must be finite, since we have assumed s_i to be a semi-visible state of the method execution). We show the result for an arbitrary class c. By assumption, root$(r) \neq c$.*

$(j = 0)$**:** *Then $i = 1$ and either s_i is the final state of the method execution (r, m), or is the pre-state of the first subcall. By Lemma 16, the only invariants which may be lost between s_0 and s_1 are those of peers and transitive owners of r. In particular, no invariants from other trees can be lost.*

$(j = k + 1)$**:** *Let s_l be the semi-visible state preceding the last direct subcall (r', m') between s_0 and s_i (i.e., $l = i - 1$ if i is even, and $l = i - 2$ if i is odd). Between s_0 and s_l there are strictly fewer than j method calls, none of which reach a receiver from c's tree, by assumption. By induction, no invariants from c's tree are lost between s_0 and s_l. Now consider the execution of the subcall (r', m'). Between the initial and final state of this method execution, there must be strictly fewer than j method calls, and so by induction, no invariants from the tree of c are lost during execution of this method, either. Therefore, no invariants from the tree of c are lost between the pre- and post-states (s_l and s_{l+1}) also. In the case $l = i - 1$ we are done. On the other hand, if $l = i - 2$ then i is odd and so $l + 1$ is even. By Lemma 16, the only invariants which may be lost between s_{l+1} and s_{l+2} are those of peers and transitive owners of r. In particular, no invariants from other trees can be lost.*

In the following, we will write $\mathbb{X}^{VT}_{(r,m)}$ to denote the expected invariants for a method (r, m) in VT (considering only the heap topology restricted to the tree of $root(r)$)[11]. We will write $\mathbb{X}^{VTS}_{(r,m)}$ to denote the expected invariants for a method (r, m) in our technique (Definition 10).

Definition 20 (Notations for expected invariants) *We re-express the expected invariants for VTS as follows:*

$$\mathbb{X}^{VTS}_{(r,m)} = \mathbb{X}^{VT}_{(r,m)} \cup \bigcup_{c \in \mathcal{E}ffs(r,m)} context(c)$$

Theorem 21 (Soundness of VTS follows from soundness of VT) *For any method execution (r, m) in a program verified by the VTS technique, suppose the current stack is of the form $\sigma\circ(r, m)$. If $\mathbb{X}^{VTS}_{(r,m)}$ holds at the initial state of the method execution, then:*

1. *At every semi-visible state of the method execution:*
   (a) *For every class c, if any invariants within $context(c)$ have been lost since the initial state of the method execution, then $filter(\sigma\circ(r, m), c)$ is a non-empty stack, with lastmost frame (r', m'), say. Furthermore, any invariants within $context(c)$ which are lost are all invariants of peers or transitive owners of (r', m').*
   (b) *All invariants within $\bigcup_{c\in\mathcal{E}ffs(r,m)} context(c)$ hold.*
   (c) *If the state is a pre-state of a direct subcall (r', m'), say, then the invariants $\mathbb{X}^{VTS}_{(r',m')}$ hold.*
2. *At the final state of the method execution, $\mathbb{X}^{VTS}_{(r,m)}$ holds.*

Proof 22 *We show all parts simultaneously, by strong induction on the number k of (transitive) subcalls made during execution of (r, m)*

$(k = 0)$**:** *Then there are only two semi-visible states: the initial and final states of the method execution, and control is always with receiver r in between. We show each part separately:*

1. (a) *The result is immediate for the initial state, since no invariants can yet have been lost. In the final state, Lemma 16 implies that the only invariants lost must be those of peers and transitive owners of r.*

[11]Note that for any receiver r, $\mathbb{X}^{VT}_{(r,m)} = context(r)$. However, we use the former notation when we wish to explicitly depend on the definitions in VT, for our arguments.

(b) *The result is immediate for the initial state, since no invariants can yet have been lost. In the final state, Lemma 16 implies that the only invariants lost must be in the tree of* r. *The only way that* $\bigcup_{c\in\mathcal{E}ffs(r,m)} context(c)$ *can include any invariants from the tree of* r *is if* $root(r) \in \mathcal{E}ffs(r,m)$. *By Lemma 12, this would imply that* $r = root(r) = c'$, *say. By part 1a, at most the invariants of* c' *itself may have been lost from this tree, in this case. But the proof obligations for the end of the method execution* ($\mathbb{E}$) *ensure that these invariants are preserved.*

(c) *The case is vacuous.*

2. *By assumption,* $\mathbb{X}^{VTS}_{(r,m)}$ *holds at the initial state of the method execution. By the previous part, the only invariants from* $\mathbb{X}^{VTS}_{(r,m)}$ *which may potentially be lost between the initial state and the final state are those of peers of* r. *However, the proof obligations for the final state of the method execution* ($\mathbb{E}$) *ensure that these invariants are preserved.*

$(k = j+1)$**:** 1. *By secondary induction on the index* i *of the semi-visible state.*

$(i = 0)$**:** (a) *Immediate, since no invariants can yet have been lost.*

(b) *By assumption,* $\mathbb{X}^{VTS}_{(r,m)}$ *holds at the initial state of the method execution.*

(c) *The 0-th semi-visible state is the initial state of the method execution, not the pre-state of any subcalls.*

$(i = n+1)$**:** *By inner induction, we can assume the result holds at the n-th visible state. We consider two cases:*

(n is even): (a) *By Lemma 16, the only invariants lost between* s_n *and* s_{n+1} *are either peers and transitive owners of* r, *or else are invariants of classes which did not hold at* s_0 *(and therefore are not lost between* s_0 *and* s_{n+1}*).*

(b) *By assumption, the invariants within* $\bigcup_{c\in\mathcal{E}ffs(r,m)} context(c)$ *hold at the initial state of the method. By induction, they also hold at the nth state. Lemma 16 implies that the only invariants lost between* s_n *and* s_{n+1} *must be in the tree of* r. *The only way that* $\bigcup_{c\in\mathcal{E}ffs(r,m)} context(c)$ *can include any invariants from the tree of* r *is if* $root(r) \in \mathcal{E}ffs(r,m)$. *By Lemma 12, this would imply that* $r = root(r) = c'$, *say. By part 1a, at most the invariants of* c' *itself may have been lost from this tree, in this case. But the proof obligations imposed at state* s_{n+1}, *which is with the pre-state of a method call* ($\mathbb{B}$) *or the final state of the current method execution* ($\mathbb{E}$) *ensure that these invariants are preserved.*

(c) *When the n+1-th state is a pre-state of a call* (r', m'), *we must show that* $\mathbb{X}^{VTS}_{(r',m')}$ *holds. By part 1a, we know that the only invariants lost in the tree of the current receiver are those which* *could have been lost in an instance method call in VT on the same receiver. Since the* $\mathbb{B}$ *proof obligation made before the call to* (r', m') *in the VTS technique implies the* $\mathbb{B}$ *proof obligation for the VT technique, then by the soundness of VT (Lemma 4), it guarantees the invariants* $\mathbb{X}^{VT}_{(r',m')}$ *hold in the n+1th state. We conclude by part 1b (and Definition 20).*

(n is odd): *Then between the nth and n+1th state, a single method call is made, say to* (r', m')*. We consider two cases:*

root$(r') =$ ***root***(r)**:** *Then, for the call to be legal,* r' *must be either the* rep *or* peer *of* r*, i.e., this is a call which would have been allowed in VT. We wish to show that the invariants* $\mathbb{X}^{VTS}_{(r',m')}$ *hold in the nth state. By inner induction and part 1a, we know that considering the tree of* r *alone, no more invariants have been lost since the initial state (when* $\mathbb{X}^{VT}_{(r,m)}$ *held) than could have been lost for a method body according to VT (i.e., only invariants of peers and transitive owners of* r*). Furthermore, our* $\mathbb{B}$ *proof obligation in VTS implies the analogous proof obligation for VT. Therefore, by the soundness of VT (Lemma 4(1)), the invariants* $\mathbb{X}^{VT}_{(r',m')}$ *hold at state* s_n*.*
By induction, the invariants in $\bigcup_{c \in \mathcal{E}ffs(r,m)} context(c)$ *hold at state* s_n*.*
By Proposition 8, we know that $\mathcal{E}ffs(r', m') \subseteq \mathcal{E}ffs(r, m)$*, and it follows that the invariants in* $\bigcup_{c \in \mathcal{E}ffs(r',m')} context(c)$ *hold at* s_n*. This means (Definition 20) that all of the invariants* $\mathbb{X}^{VTS}_{(r',m')}$ *hold at* s_n*, the pre-state of the call to* (r', m')*. By (outer) induction, all of these invariants hold at the post-state of the call* (r', m') *and 1a holds. To show 1b, we need to be sure that* $\bigcup_{c \in \mathcal{E}ffs(r,m)} context(c)$ *hold at state* s_{n+1}*. We know that these invariants held at* s_n*, and furthermore that for all* $c \in \mathcal{E}ffs(r', m')$ *the invariants in* $context(c)$ *still hold at* s_{n+1}*. However, for any* $c \notin \mathcal{E}ffs(r', m')$*, none of the invariants in* $context(c)$ *may be lost during the call* (r', m')*, by Propositions 8 and 18, therefore we can conclude 1b. Note that 1c is vacuous in this case.*

root$(r') \neq$ ***root***(r)**:** *Then* $r' = c'$ *for some* $c' \in \mathcal{E}ffs(r, m)$*. Again, we aim to show that the invariants* $\mathbb{X}^{VTS}_{(r',m')}$ *hold in the nth state. Note that* $\mathbb{X}^{VT}_{(c',m')} = context(c')$ *which is contained within the set* $\bigcup_{c \in \mathcal{E}ffs(r,m)} context(c)$ *known to hold at* s_n*. Similarly to the previous case, we know that* $\mathcal{E}ffs(c', m') \subseteq \mathcal{E}ffs(r, m)$*, and so the rest of the invariants in* $\mathbb{X}^{VTS}_{(r',m')}$ *also hold at* s_n*. By (outer) induction, all of these invariants hold in the final state of the call* (r', m')*, and so they hold at* s_{n+1}*, and any invariants lost are those permitted by 1a. We obtain 1b by similar argument to the previous case, and 1c is again vacuous.*

2. *In the final state of (r, m), by part 1a, we know in particular that in the tree of r, no more invariants may have been lost (since the initial state of the method) than might be in VT. Furthermore, the $\mathbb{E}$ obligation imposed by VTS implies that imposed by VT in the final state. By the soundness of VT (Lemma 42), we know that these proof obligations are sufficient to guarantee that $\mathbb{X}^{VT}_{(r,m)}$ hold at the final state of (r, m). Furthermore, by part 1b, we know the invariants in $\bigcup_{c \in \mathcal{E}ffs(r,m)} context(c)$ also hold. Therefore, all invariants in $\mathbb{X}^{VTS}_{(r,m)}$ hold, as required.*

Static Initialisation

We have not discussed static initialisation so far in this paper. In brief, we are able to incorporate the Java semantics for static initialisation with our technique. 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). We require that static initialisers establish the invariants of the corresponding class. When verifying the code of a static initialiser, it is safe to assume the invariants of any objects which are created during the initialisation (and owned by the class), but no static invariants of other classes can be known to hold, and so it would be dangerous to permit calls to static methods of other classes (we forbid this). This is because static initialisers are not called in a controlled manner, and there is no opportunity to employ the effects sets in the way we do for method calls in order to ensure that certain classes are not already on the call-stack when the code is executed. It may also safely be assumed that no instances of the class exist when the static initialisation is executed (since, by the Java semantics, the class must be loaded and initialised before any instances can be created). Therefore, any static invariants which quantify over instances of the class will automatically be vacuously true. In practice, we believe we can handle many practical examples of static initialisers, since we can naturally permit new objects to be created, and have the fields modified and methods called in the usual way for object instances. The only serious restriction is the inability to make static method calls during an initialiser, but this restriction can be partly lifted by the extended technique presented in Section 5.

Assuming Static Invariants in Proofs

An aspect of verification we have not examined in detail is how proof obligations are actually discharged. In particular, when one wishes to do verification in practice, it is important to know what assumptions can be made for the proofs which our techniques require. According to the discussions so far, static invariants are only known to hold at the initial and final states of static method calls on the same class, as well as in the initial and final states of methods with the corresponding class in their effects (since such a class is guaranteed by our technique to be in a consistent state—cf. Definition 20). However, it seems reasonable in practice that an instance method might wish to depend on static invariants actually holding for its verification. For example, if the `Thread` class maintains a static invariant that all instances have unique identifiers, this is likely to be useful for the verification of instance methods of `Thread`. Our technique as presented thus far verifies instance methods without general explicit knowledge of which static invariants hold (although such invariants must be preserved). It is, however, easy to extend method specifications to allow explicit declarations of extra static invariants the method depends on. This can be most easily encoded into VTS by manually adding some classes to the effects set of the method (in which case, their invariants will be required to hold by VTS). Even better, such explicitly declared dependencies on static invariants can be treated as a separate kind of effects set (secondary to those existing in VTS), which can be propagated independently. This allows us to handle extra examples in which a static invariant is required and can be guaranteed to hold, but the class is already on the call-stack, and some of its owned objects may *not* be in a consistent state. In this case, it would be unsafe to make a call on the class (which might result in calls to its owned objects), but can still be safe to depend on the class' static invariants holding, which could be expressed through the two separate effects sets.

5 REFINEMENTS

In this section we propose two refinements to VTS which make it more widely-applicable. The first refinement, "strong VTS" (hereafter SVTS) adds an extra facility for callbacks to be made between trees: In VTS, re-entrance of a tree is only possible when a static method of class c calls out to another tree and then that call re-enters c. SVTS allows control to re-enter a previously visited tree generally at positions which could have been called from the last frame in that tree. This extra flexibility is needed, for instance, when a static method calls instance methods of its arguments. The second refinement, "layered VTS" (hereafter LVTS), allows for a more-refined notion of effect annotations, by stratifying the heap topology into "levels" in such a way that calls down to lower levels can never callback, and so need not be considered in the effect annotations. This refinement reduces the annotation burden and also allows overriding methods to call static methods of lower levels, even if their classes are not explicitly mentioned in the effects of the overridden method. Finally, we will discuss the issues involved in combining these two extensions.

SVTS—Safe callbacks

In the definition of the VTS technique, we observed that much of the design of the technique, and the argument for soundness, depended on the idea of 'filtering'; that, from the point of view of just one tree in the topology, the 'filtered' flow of control within that tree corresponds to a sequence of method calls permitted by VT, and that the proof obligations imposed also correspond, and guarantee invariants in a similar way to VT. However, our technique currently only allows control to pass to a new tree at the 'root', by calling static methods. This means that is impossible to "resume" control in a tree except in the special case that only the 'root' has been called so far. From the point of view of the 'filtering' idea, it seems that it should be possible to allow a more permissive discipline regarding control passing between trees, so long as the original ideas of VT are respected 'per tree'.

Introducing callbacks.

We consider a refinement of our technique with the ability to make calls directly to objects in other trees. This kind of extension seems desirable for practical, as well as philosophical reasons. To illustrate the practical reasons we consider a slight variation of the standard way to print objects to the terminal in Java[12] [13]:

```
class System {
  static void print(any String s)
  {
    // send the string to the terminal window
  }

  static void print(any Object o)
  {
    System.print(o.toString()); //subcall not permitted on any reference
  }
}
```

The overloaded method `print` has a direct implementation for `String` arguments, and requires all other objects to first generate their `String` representations, which can then be passed to the direct implementation. This code is very natural, but requires a callback to be made on `Objects` `o` which are passed to a static method of `System`. In the technique presented thus far, this callback is not permitted, since even pure methods (which `toString` could feasibly be) may be called only on receivers in the context of the current receiver. The *reason* for this restriction is that otherwise our technique cannot guarantee statically that the invariants of the callee will hold, and therefore to guarantee the safe execution of the method.

However, this restriction seems too strong in our generalised heap topology. From the point of view of the receiver which calls `System.print(o)`, if it passes an object which it regards as its rep or peer, it is easy for it to guarantee that the invariants of `o` hold. Furthermore, unless a callback is made into the tree in which `o` belongs, its invariants will continue to hold during execution of `System.print(o)` (regardless of the actual code in the `print` method), by Proposition 18. We therefore propose a new universe annotation, *strong_any*, whose intended semantics is that the object

[12]Thanks to Rustan Leino for suggesting this example.

[13]We have made the `print` methods directly available as static methods of the `System` class, rather than as instance methods of the stream in the static `out` field of the class. This is because, in the technique presented in this section, we cannot generally handle direct calls to instance methods of static fields (rather than calling via a static method of the class). We will explain how to eliminate this restriction at the end of this subsection.

referred to is guaranteed to be safe to make a callback on; we will permit all calls on *strong_any* references. We consider *strong_any* to be a specialisation of any, and a *strong_any* reference can be upcast to any if desired (losing the special semantics associated with a *strong_any* reference type), but downcasts are not permitted. Our *strong_any* annotation expresses that an object's invariant may be assumed to hold without constraining the object's position in the ownership topology. In that sense, it is similar to a Spec# specification stating that an object is peer-consistent. However, since we use a type system, we have to over-approximate the consistency of objects and enforce a stronger system invariant, as we discuss next.

Details.

In order to ensure soundness, we design an extension of our technique to guarantee the following 'system invariant':

Definition 23 (System invariants for SVTS) *At any semi-visible state of any method execution (r, m), for which o is a strong_any reference in scope:*

1. *o is guaranteed to be from a different tree to the current receiver.*
2. *The invariants within o's context are guaranteed to hold.*
3. *Any two strong_any references in scope refer either to different trees from each other, or to objects which have identical contexts (i.e., which are peers of one another).*
4. *o is guaranteed to be from a different tree than any class c with $c \in \mathcal{E}ffs(r, m)$.*

The second criterion is necessary in order to ensure that the currently-executing method cannot violate invariants within the context of o. The need for the third and fourth criteria will be made clear later in this section.

Since this system invariant is expressed and guaranteed in terms of the current receiver and the history of calls on the stack, it would not be consistent to allow *strong_any* to be used for the type of a field (since, a different sequence of calls might reach the same object while the referred object's invariants did not hold). Because the properties guaranteed by *strong_any* are specific to a particular method execution, we only allow this universe annotation to be used for the formal parameters to methods. We do not allow it for the return types of methods (since again, the system invariants might not be maintained after the method returns, particularly the fourth criterion above).

In fact, a general difficulty with the creation of *strong_any* references is that whenever a receiver has concrete knowledge about its relation in the heap topology with an object (i.e., when it regards the object as a rep or peer), then it comes from the same tree as the object, and to consider such an object as a *strong_any* would

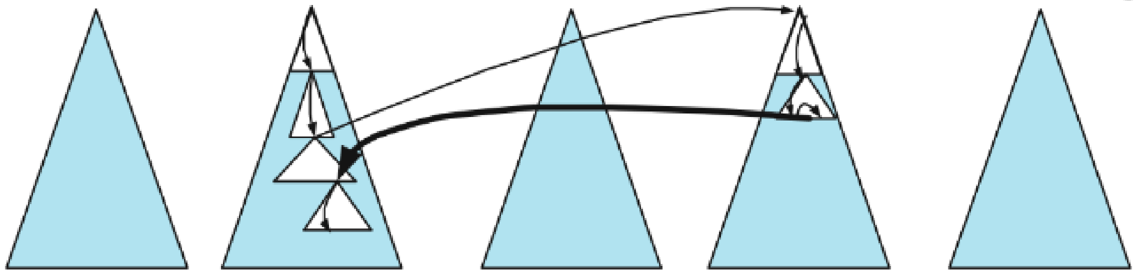


Figure 5: Calls across trees, invariants hold in shaded areas. The bold line indicates a call on a *strong_any* argument.

violate our system invariant. We evade this difficulty by allowing *strong_any* to be used as a modifier for formal method parameters, and allowing actual arguments to be implicitly promoted from rep or peer to *strong_any* at the time of calling the method, so long as the method call is to a different tree. This means that the invariants within the context of the passed reference can be naturally guaranteed, automatically in the case of a rep, and by imposing a proof obligation that all peer invariants are established first, in the case of a peer. Note that we do *not* allow explicit casts of rep or peer references to *strong_any*. Thus, the only way in which new *strong_any* references can come into existence in the program is by implicit promotion, during calls to other trees.

As a further refinement to our idea, there is actually no need to avoid *impure* calls being made on *strong_any* references. The reason for this is that we have chosen that only a rep or peer of the last receiver in the same tree could be passed as a *strong_any* reference, and we impose the same proof obligations beforehand as we would if we were calling a (potentially impure) method on the referred object directly. Therefore, we can view our *strong_any* references, as 'permitted next receivers' in other trees, which relates neatly to the idea of filtering.

Example.

We can now rewrite the example presented above in a way which can be accommodated:

```
class System {
  static void print(any String s)
  {
    // send the string to the terminal window
  }

  static void print(strong_any Object o)
  {
    System.print(o.toString()); //subcall permitted on strong_any reference
  }
}
```

```
class c{
  rep Object o;

  void m() {
    System.print(o); // promote rep as strong_any
  }
}
```

Note that during an execution of `m()` above, it is guaranteed to be safe to promote `o` to *strong_any* when making the call on `System`; it must be that `System` is from a different tree, since otherwise such an execution of `m()` could not be reached (by the owner-as-modifier property, it would have to be reached via another active static method execution on `System`, and the effects would have ruled out reaching the method `m()` which might call back to `System`.

Passing *strong_any* references.

We observe that it is safe (i.e., the system invariant is preserved) to pass an existing *strong_any* reference from a method to a subcall, so long as the subcalls do not reenter the tree in which the referenced object resides. There are two ways in which this potential problem could occur.

Firstly, the tree could be reentered at the root, via a static method call to the class c at the root of the tree. Due to the effect annotations, this can only happen if the *strong_any* reference was created during execution of another method on class c, and no objects in the same tree are currently receivers on the call stack. We can therefore rule out this problematic case specifically by defining that, during execution of a static method of class c, if a (static) subcall is to be made for which c is in the effects, no *strong_any* references may be passed.

Secondly, the tree could be reentered by a callback to a *strong_any* reference, as we have now permitted. We can avoid violating the system invariant by requiring that any references passed as *strong_any* must be known to have contexts disjoint from the *strong_any* receiver. In practice, unless some extra machinery is employed to guarantee such disjointnesses, this will mean that when a call is made to a *strong_any* receiver, no *strong_any* arguments can be passed.

We do not allow *strong_any* annotations to be used in the *return* types of method calls: this is because there are no guarantees that the point to which the method returns will not be within the same tree as the referenced object, and so violate the system invariant.

Overlapping *strong_any* references.

The ability to allow (impure) callbacks into trees adds great flexibility. However, a subtle problem arises when multiple *strong_any* parameters have different (but overlapping) contexts, as is illustrated by the following example:

```
class CompositeInt {
  rep Integer child = new Integer(1);
  int total = 2;

  // invariant total > child.getValue()

  int getTotal() {
    return total;
  }

  double calculateRatio() {
    return total / (total − child.getValue());
  }

  void violate() {
    CompositeInt.bad(child, this); // pass rep/peer as strong_any
  }

  static void bad(strong_any Integer i, strong_any CompositeInt o) {
    i.setValue(o.getTotal());
    o.calculateRatio();
  }

  void attempt() {
    child.setValue(this.getTotal()); // breaks invariant of this
    this.calculateRatio(); // rejected by our technique
  }
}
```

In this case, the method `violate` passes a `CompositeInt` instance o along with its `child`, both as *strong_any* references (to match the signature of `bad`). The resulting call `i.setValue(o.getTotal())` breaks and does not reestablish the invariant of o, which then receives a dangerous callback. The reason for this problem can be seen by examining the method `attempt`. Here, equivalent method calls are made to those resulting from `violate`, but without the indirection of the *strong_any* references. The difference is that control returns to the instance in between these two calls, at which point an extra proof obligation (which fails) is required before calling `calculateRatio`. This obligation is applied too early when the analogous calls are made via `bad`; when the `this` reference is passed as a *strong_any* the invariants are established ready for any callback to be made, but these invariants are then broken in between this point and the call to `calculateRatio`.

To guarantee that the contexts of multiple *strong_any* references only overlap when they are identical (and therefore avoid the unsoundness described above), we employ the following restriction: it is not permitted for both a rep and a peer to be implicitly promoted to *strong_any* for the same method call. This avoids the mismatch between proof obligations: even if many callbacks are made to *strong_any* references, the invariants expected and reestablished for each of these callbacks will be the same each time.

For similar reasons, it would be unsound for a method with a class c in its effects to have a *strong_any* reference passed to it which refers to an object in the same tree as c. This would mean that the method would have the ability to callback to different points in the same tree, just like in the example above. However, the only way in which this could happen is when a class c calls another class c', passing a rep object as a *strong_any* parameter. We rule this case out explicitly in our definitions below.

Definitions.

In the following definition, we highlight the main differences with VTS (cf. Definition 10) in **bold**:

Definition 24 (The SVTS technique) $\mathbb{X}$ *The invariants of receivers within the context of r are expected, plus invariants within the context of all classes c such that $c \in \mathcal{E}ffs(r, m)$,* **plus invariants within the contexts of each *strong_any* reference in scope**.

$\mathbb{V}$ *The invariants of all transitive owners of r are vulnerable,* **plus all transitive owners of each *strong_any* reference in scope**[14]*, plus invariants of peers of r, and the static invariants of their classes and superclasses.*

$\mathbb{B}$ *In the pre-state of a method call, if the callee is a peer of r,* **or if a peer of r is to be promoted as a *strong_any* argument to the call**, *the invariants of all peers must be established to hold. Similarly, if the caller is a class c and the new call is to a static method with c in its effects, the invariants of c must be established. Furthermore (in all cases), the invariants of the (super)classes of all of the peers of the caller receiver must be shown to be preserved (since the initial state of the caller's method body).*

$\mathbb{C}$ *From an instance method body (i.e., r is an object), a call to another instance method is allowed if the callee is a peer or rep of r,* **or if the callee is a *strong_any* reference**, *or if the method is pure and the callee is known to be within the context of r. From an instance method body, a call to a static method is always allowed. From a static method body (r is a class c), a call to a static method (of any class)* ***is allowed unless both a strong_any parameter is passed to the method and c is in the effects of the method****, while a call to an instance method is allowed if c is not in the effects of the method, and either the callee is a rep of c or the method is pure and the callee is known to be within the context of r.*

Any *strong_any* references in scope when a method call is made may be passed as *strong_any* arguments to the method call so long

[14]These invariants are vulnerable because they may depend on fields modified during calls to the *strong_any* reference. However, note that such invariants will also not be expected to hold for these calls, and so do not play a significant role in our soundness arguments.

as it is not a call to a different tree (i.e., not a call on a *strong_any* reference itself, or a static method call, excepting the special case of a class "calling itself"). Such references may also be 'upcast' as any references, if that is all that the method requires.

If a method requires *strong_any* arguments and rep or peer refrences are passed in their place, these references are implicitly promoted to *strong_any*, provided the call is made to a different tree (i.e., the call is either itself a call on a *strong_any* reference, or a call to a static method, excepting the special case of a class "calling itself").

The components $\mathbb{D}$, $\mathbb{E}$, and $\mathbb{U}$ are identical to Definition 10.

Soundness.

We discuss briefly how soundness of the SVTS technique can be deduced from soundness of the VTS technique. Essentially, the argument follows exactly the same lines. However, our expected set now has the following format (cf. Definition 20):

$$\mathbb{X}^{SVTS}_{(r,m)} = \mathbb{X}^{VT}_{(r,m)} \cup \bigcup_{c\in\mathcal{E}ffs(r,m)} context(c) \cup \bigcup_{o\in strong\_any(r,m)} context(o)$$

where by $strong\_any(r,m)$ we mean the set of all *strong_any* references in scope for the method body (r,m)[15].

We made our soundness argument for VTS rather more general than it needs to be (particularly regarding filtering of stacks, which in VTS may only reenter a tree when only the root has been visited anyway), in order that the argument adapts straightforwardly for SVTS. Indeed, we can prove an analogous result to Theorem 21 in a very similar way. The only extra difficulty is in showing that, when control switches from one tree to another (either calling or returning from a method), the expected invariants contributed by the *strong_any* references in scope always hold. This is achieved by adding extra cases to the soundness theorem to require that the system invariant is preserved at all times. In fact, we can show as a lemma that for each $o \in strong\_any(r,m)$, $root(o) \neq root(r)$, and for $o_1, o_2 \in strong\_any(r,m)$, either $context(o_1) = context(o_2)$ or $root(o_1) \neq root(o_2)$. We then prove a main soundness theorem with a similar form to that of VTS, but add the requirement that at every semi-visible state of the method (r,m), the invariants within the contexts of all *strong_any* references hold.

[15]Technically, the required information about *strong_any* references in scope requires us to store arguments in our stack frames, as well as receivers and method names. Such an extension is straightforward, and does not affect the rest of this work.

Remarks.

We can add further flexibility to the SVTS technique by allowing static final fields to act as further "roots" in our topology. In our example above, we made `print` a static method of `System`. This simplified the example, but in reality, a `String` is printed in Java via an instance method of the stream in the static field `System.out` (as in `System.out.println()`). Firstly, this is problematic for our technique, since it involves a direct call to an object owned by `System`, without calling an intermediate static method on the class. More subtly, our technique would impose a restriction that the implementation of `println()` could not call other static methods of `System`, because our effect annotations would rule this out while control was with an object owned by `System`. We observe firstly, that for static *final* fields[16], it would usually be guaranteed that there is a unique, fixed instance object stored, and it would be possible to extend our approach to treat this as a 'fixed position' in our topology, just as the classes are. If we were to make `System.out` a 'root' of a tree in its own right, and consider it analogously to a separate class as far as effect annotations were concerned, this would make calling methods such as `System.out.println()` directly a natural feature, while providing additional flexibility for such methods to make use of other parts of the `System` class.

Unfortunately, although this approach works in general for static final fields, it is not actually applicable to the particular case of `System.out`. This is because, despite the `final` declaration, for backwards-compatibility reasons in Java it is possible to modify the object referenced by this field, using the static method `setOut`. This opens up a more-difficult problem: if `out` is a mutable object, then it cannot be easily considered a root of a tree in our topology (and be included soundly in effect annotations etc.), but at the same time, calls are made directly to `out` and not via static methods of `System`: our technique would naturally forbid these. We argue that the *reason* such calls are made directly is that the `PrintReader` referenced by `out` has only a weak relationship with the class `System` itself. While `System` is responsible for initialising and re-assigning the object, and provides a means of accessing the object, via the class name, it does not obviously have concerns which are deeply entangled in the workings of the object. In particular, it seems plausible that any invariants declared in the class `System` would not need to depend on the fields of the object `out`. This suggests a weaker association than that which the universe modifier `rep` specifies: there is a kind of ownership in terms of managing the identity of the `out` object, but it does not need to be as deep as `rep` allows. In particular, it is not necessary for `System` to have invariants which depend on the fields of `out`. If this weaker ownership relationship could be formalised in a variant of our system (perhaps with another universe modifier), we could then consider allowing calls such as `System.out` directly. This could be sound, even though it involves entering a tree lower than the root, because there is no danger of leaving invariants further up in the tree broken. In terms of effect annotations, we would

[16]The field `out` of `System` *is* declared final in Java, but (as a special case) can actually be modified - see next paragraph

consider a call to a method of `System.out`, say, in the same way as a call to `System`, in order to rule out dangerously reentering the tree, as usual.

The possible implications of this notion of ownership without deep dependencies should be explored in future work: it might well be that it is useful for describing certain object structures in a flexible way, as well as for dealing with static fields in a flexible manner.

LVTS—reducing effects annotations through levels

In this section we revert to the original definition of VTS (without *strong_any* references), and consider a different extension of the technique (after which, we will consider the implications of combining the two refinements).

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

Example 25 (Method Overriding and Effects) *Consider the* `String` *class of the Java API. An implementation of this class can exploit the 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. A method* `intern` *is provided in class* `String` *to access these pooled references, implemented using static fields and methods. 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)
  {
    String s = new String("Equals() Called");
    System.print(s.toLowerCase());
    return this == o;
  }
}
```

It happens that the implementation of `toLowerCase` *involves calling the static method* `intern` *of the* `String` *class, which in turn results in accessing the static data of the class. In this case, our previous definitions imply that we need to have* `String`$\in \mathcal{E}\mathit{ffs}($`MyClass`,`equals`$)$*, and because of Def. 6 (item 3), we also need that* `String`$\in \mathcal{E}\mathit{ffs}($`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 the methods in* `String` *are implemented), and a usability problem (our technique forbids our intended 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[17]. 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 26 (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. 6). We therefore design our technique around the following crucial 'system invariant':

Definition 27 (System Invariant for LVTS) *If a method call is made to a receiver in a particular level, no resulting subcall is made to a receiver in a higher level. In particular, when a method call is made to a receiver with a lower level than the current receiver, it may not result in calls being made to receivers on the same level as the current receiver.*

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[18]. 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. 6 are shown in **bold**) are:

Definition 28 (Refined Effects) *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) $\mathcal{E}ffs(c', m') \subseteq \mathcal{E}ffs(c, m)$ and

   (b) $c' \in \mathcal{E}ffs(c, m)$.

4. *If c' is a subclass of c which overrides a method m, then $\mathcal{E}ffs(c', m) \subseteq \mathcal{E}ffs(c, m)$*

[17]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.

[18]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.

The refined conditions given permit smaller effects sets for methods than those of Def. 6. Considering the example at the start of the section, it is no longer necessary (or indeed, allowed) for `String` to be in $\mathcal{E}\!f\!f\!s$(`MyClass`,`equals`).

In the following definition, the only explicit change compared with Definition 10 is to the $\mathbb{X}$ parameter, but because of the restriction of effects sets to only classes on the same level, the points at which the effects sets are checked are less restrictive.

Definition 29 (The LVTS technique) $\mathbb{X}$ *The invariants of receivers within the context of* r *are expected, plus invariants within the context of all classes* c *such that* $c \in \mathcal{E}\!f\!f\!s(r, m)$, **plus invariants within the contexts of all classes** c' **with** $\mathcal{L}vl(c') < \mathcal{L}vl(r)$.

All other components are identical to Definition 10.

Remark. Considering the earlier discussion of static initialisation (Section 4), with the LVTS refinements we no longer need the full restriction that no static method calls can be called from a static initialiser. Rather, it is safe for classes of lower levels (than the class being initialised) to have static methods called on them, but not classes of the same level. This is because, a static initialiser c will only be executed when c, or a subclass of c, is (first) mentioned in the body of an executing method. By our restrictions on levels, the method body must come from a class of equal or higher level than c, and thus, the call-stack must not have currently visited classes of lower levels than c.

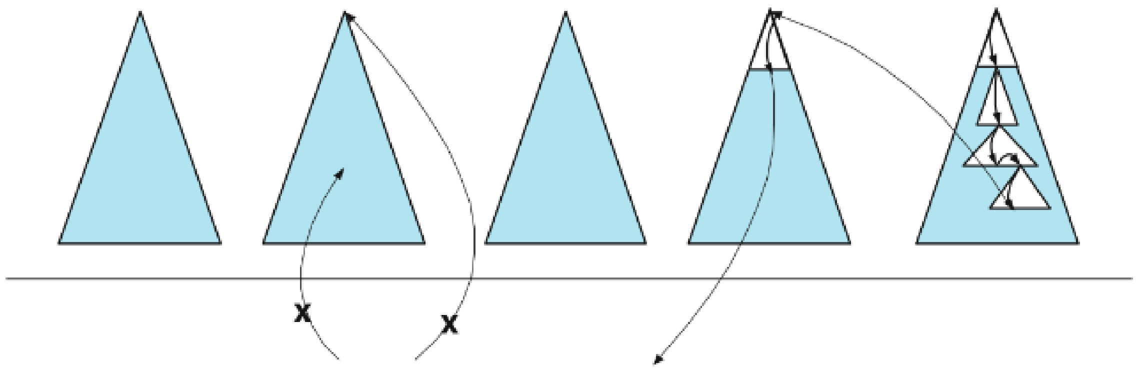


Figure 6: Trees in one level. The current level may call into the lower level, but no calls from the lower level may come into the current level. The level of an object is determined by the class that transitively owns the object, not by the object's type.

Soundness.

We discuss briefly how soundness of the LVTS technique can be deduced from soundness of the VTS technique.

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, which corresponds to our claimed system invariant:

Proposition 30 (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 stack $\sigma{\circ}(c,m){+}\sigma'{\circ}(o,m')$, in which σ' 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 stack $\sigma{\circ}(r_1,m_1){+}\sigma'{\circ}(r_2,m_2)$, in which r_1, r_2 can be any receivers, we have $\mathcal{L}vl(r_1) \geq \mathcal{L}vl(r_2)$.*
4. *For any stack $\sigma{\circ}(c,m){+}\sigma'{\circ}(c,m')$, for all the receivers r from frames in σ', we have $\mathcal{L}vl(r) = \mathcal{L}vl(c)$.*

Proof 31

1. *By induction on the number of objects created (in the tree). For the inductive step, we use the fact that the class of the creator of an object (i.e., the receiver of the method execution which causes it to be created) must explicitly mention the dynamic class of the new object in its code, and so we obtain the required inequality by Definition 26.*
2. *Since instance method calls can only be made on peers and reps, it must be the case that c transitively owns o. If we iteratively follow the creator of o, its creator, etc., we must eventually reach c (since this tree originally contained only c). Therefore, the result follows from the previous two parts.*
3. *By induction on the number of static method calls in the sequence. If none, then the result follows from the previous part. For the inductive step, suppose we make an extra static call on class c'. If the currently-executing method is a static method (c,m), say, then c' must be mentioned in the method m of c, and by Definition 26 we obtain $\mathcal{L}vl(c) \geq \mathcal{L}vl(c')$. On the other hand, if the previous method is an instance method (o,m), let c_1 be the dynamic type of o, and let c_2 be the class in which the definition of m is found, from class c_1. We have $c_1 \leq c_2$, and so $\mathcal{L}vl(c_1) \geq \mathcal{L}vl(c_2)$. Since the method m of class c_2 mentions c', we must have $\mathcal{L}vl(c_2) \geq \mathcal{L}vl(c)$.*
4. *Consider any intermediate receiver r. By applying Proposition 30(2), twice, we obtain $\mathcal{L}vl(c) \geq \mathcal{L}vl(r)$ and $\mathcal{L}vl(r) \geq \mathcal{L}vl(c)$.*

This allows us to construct similar soundness arguments to those for VTS. In fact, we explain here a neat way to obtain such a soundness result. We can, in fact, consider an encoding of LVTS into a variant of VTS, in which we regard all classes on lower levels than the receiver of a method to be implicitly included in the effects of the method. In this way, we make explicit in the effects sets the implicit permission to call down to any classes in lower levels. If we consider these extended effects sets then it turns out that the soundness argument works out analogously to that of VTS. The proposition above guarantees that even with these implicitly extended effects sets, effects sets never increase as new calls are made, and that the effects conservatively predict the trees which may be entered by new calls (cf. Proposition 8).

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 ways. Considering the extreme case of only one level, we return to VTS, where 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.

Combining levels with callbacks

In this section we consider the combination of both refinements: the inclusion of both *strong_any* references and the heap topology stratified with levels. Unfortunately, the two extensions do not work well together. The reason for this is that the smaller effects sets made possible with the levels are only adequate so long as callbacks are known not to be made from lower to higher levels. With the ability to pass *strong_any* references in a call 'down' levels in the topology, callbacks from lower to higher levels become possible.

As an outlined example, consider the following code:

```
// LEVEL 2
class B extends A{
    void m1(){
        A.m2(this);     // promote this as strong_any
    }
    void m3( ){
        A.m4();
   }
}

// LEVEL 1
class A{
     .... // some static invariant

    static void m2(strong_any A a){
         // do something to break the static invariant of A
        a.m3();
    }

    static void m4(){ .... }
```

The call to `A.m4` is made while the static invariant of `A` is temporarily violated. However, since `A` is on a lower level to `B`, no effects need to be given for the call to `A.m2`. This means that the resulting call to `m4` is not accurately predicted by the effects of `m2`, and so can be reached when the appropriate invariant does not hold. Note that it would not be possible to exploit this problem to make dangerous static method calls on the higher level, since these calls would be caught by the effects; it is the fact that we re-enter the lower levels without computing effects on that level that creates the difficulty.

Unfortunately, this problem seems hard to avoid. The very nature of the effects computed for LVTS means that there is no statically-available information about the results of calls down to lower levels (since such calls are not regarded as significant in the computation of effects). Furthermore, it is not statically known whether a *strong_any* reference in scope might refer to an object in a higher level (its static type may be a class of a lower level than its dynamic type).

We can recover soundness by enforcing the (very strong) restriction that whenever a *strong_any* reference is passed to a lower level, no invariants are broken in the lower level (while the *strong_any* reference is in scope). In this way, if the lower level is unexpectedly re-entered, no invariants will be broken at the time. Unfortunately, a method with a *strong_any* parameter has no information about which level the *strong_any* reference came from, and so we must conservatively impose that *all* methods with *strong_any* parameters avoid breaking any invariants. This creates particular problems for instance methods, since the receiver will not typically know its owner, and will not be able therefore to ensure that it preserves all invariants which may depend on the fields it modifies. For this reason, we must forbid *strong_any* references from being passed to impure instance methods. In the case of pure methods, no invariants can be lost during their execution and so a restriction is unnecessary[19].

6 CONCLUSIONS, RELATED WORK, AND FUTURE WORK

We have outlined three verification techniques based on the visibility technique [11], 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 owner-as-modifier discipline. We have shown two ways in which our basic technique can be improved, making it applicable to

[19]Note that pure methods may create new objects, and this might in principle have implications for any static invariants quantifying over all instances of a class. However (although we have not explicitly mentioned constructors in this paper), any constructors should preserve such static invariants appropriately, just like instance methods.

more examples.

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 [5, Sec. 8.2]. In JML and in our work, both instance and static invariants are supposed to hold in visible states [11]. 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* [12], allowing a method to create a new object, but to postpone the decision of assigning it an owner. Alternatively, we can incorporate a second 'flavour' of static methods, which are executed relative to the callee. This has the advantage that it can also naturally handle *utility classes*, in which additional functionality is added to an existing class via static methods in a secondary class. Since this pattern is fairly common, this extension seems worthwhile. However, it is straightforward to add these 'relative' methods to our work, and orthogonal to the main issues tackled in this paper.

Leino and Müller [7] extend the Boogie methodology [6] to static invariants: static fields may be reps; static 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].

Acknowledgements.

We are very grateful to Rustan Leino for extensive discussions on our previous version of this work, and for providing useful examples which have fuelled many of the new developments in this work. The previous version of this work was also 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] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *JOT*, 4(8):5–32, October 2005.

[2] S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. A unified framework for verification techniques for object invariants. In *ECOOP*, LNCS. Springer, 2008.

[3] M. Huisman and E. Poll. Formal Techniques for Java-like Programs (FTfJP). Technical Report ICIS–R08013, Radboud University Nijmegen, June 2008.

[4] K. Huizing and R. Kuiper. Verification of object-oriented programs using class invariants. In *FASE*, volume 1783 of *LNCS*, pages 208–221. Springer, 2000.

[5] 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.

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

[8] K. R. M. Leino and P. Müller. Modular verification of static class invariants. In *Formal Methods*, volume 3582 of *LNCS*, pages 26–42. Springer, 2005.

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

[10] 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.

[11] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer, 2002.

[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] P. Müller and A. Rudich. Ownership transfer in Universe Types. In *OOPSLA*, pages 461–478. ACM, 2007.

[14] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, 1997.

[15] A. Salcianu and M. C. Rinard. Purity and side effect analysis for Java programs. In *VMCAI*, volume 3385 of *LNCS*, pages 199–215. Springer, 2005.

[16] A. J. Summers, S. Drossopoulou, and P. Müller. A universe-type-based verification technique for mutable static fields and methods—work in progress. *Electronic Proceedings of FTfJP 2008 [?]*, pages 111–124, 2008.

ABOUT THE AUTHORS

Alex Summers is a Research Associate in the Department of Computing at Imperial College London, and is working on the Möbius project. He can be reached by email at alexander.j.summers@imperial.ac.uk
See also his webpage at http://www.doc.ic.ac.uk/~ajs300m

Sophia Drossopoulou is Professor in Programming Languages in the Department of Computing at Imperial College London. She can be reached by email at s.drossopoulou@imperial.ac.uk
See also her webpage at http://www.doc.ic.ac.uk/~scd

Peter Müller is Professor in Programming Methodology at ETH Zürich. He can be reached by email at peter.mueller@inf.ethz.ch
See also his webpage at http://pm.inf.ethz.ch/people/mueller