Modular verification of static class invariants

K. Rustan M. Leino [1] and Peter Müller [2]

[1] Microsoft Research, Redmond, WA, USA, leino@microsoft.com
[2] ETH Zürich, Switzerland, peter.mueller@inf.ethz.ch

Abstract. Object invariants describe the consistency of object-oriented data structures and are central to reasoning about the correctness of object-oriented software. But object invariants are not the only consistency conditions on which a program may depend. The data in object-oriented programs consists not just of object fields, but also of static fields, which hold data that is shared among objects. The consistency of static fields is described by *static class invariants*, which are enforced at the class level. Static class invariants can also mention instance fields, describing the consistency of dynamic data structures rooted in static fields. Sometimes there are even consistency conditions that relate the instance fields of many or all objects of a class; static class invariants describe these relations, too, since they cannot be enforced by any one object in isolation.
This paper presents a systematic way (a *methodology*) for specifying and verifying static class invariants in object-oriented programs. The methodology supports the three major uses of static fields and invariants in the Java library. The methodology is amenable to static, modular verification and is sound.

Keywords: Static class invariant, verification, object-oriented programming, static field

1 Introduction

A central problem in reasoning about a computer program's correctness comes down to reasoning about which program states the program can ever reach. Programmers rely on that only some states are reached, most prominently by assuming that the program's data structures satisfy certain properties. These properties are called *invariants*. By declaring invariants explicitly, the programmer can get support from tools (like the tools for JML [4] or Spec# [2]) that make sure the program maintains the invariants. In this paper, we present a systematic way (a *methodology*) for specifying and reasoning about invariants in object-oriented programs. In particular, we consider invariants that are described and enforced at the level of each class, called *static class invariants*.

The main data structures in modern object-oriented programs are stored as the state of individual objects, in variables known as *instance fields*, and as the state of classes, in variables known as *static fields*. We have identified three major uses of static fields and invariants in the standard Java libraries (Java 2 Standard Edition version 5.0). First and foremost, static fields are used to store shared values. For example, the well-known static field $System.out$ in Java provides an output stream whose characters flow to the console. Second, static fields are used to hold the roots of object data structures. For example, the implementation of the $String$ class in Java has a shared pool of interned strings, storing canonical string references for certain character sequences. Third, static fields are occasionally used to reflect something about all instances of a class. For example, Java's $Thread$ class assigns unique identifiers to its instances and uses a static

field to keep track of which identifiers are in use. In all of these three cases, implicit or informally documented static class invariants describe the intended consistency conditions. The methodology we present in this paper enables the explicit specification and formal verification of these invariants.

Previous work on specifying and verifying invariants in object-oriented programs have developed methodologies for *object invariants*, which describe the consistent state of individual objects and aggregate objects, formations of individual objects into one logic unit [16, 1, 10, 3]. However, these methodologies do not apply to static class invariants, because the classes of a program build on each other in a way that is different from the principal way in which objects in an aggregate build on each other: whereas an object has a unique point of use in an aggregate, a class is used by many other classes.

Our basic methodology draws from the Boogie methodology for object invariants [1, 10, 3], but innovates in significant ways to handle static class invariants. First, to address the abstraction problem that arises when a class is used by several other classes, our methodology performs different bookkeeping for invariants, tailored to work with any partial order among classes. Second, our methodology introduces such a partial order on classes. This order makes it possible for a method override to rely on the static class invariant of the subclass even when the method specification in a superclass is not able to name the subclass. The order also prescribes how to initialize classes, which provides a way to avoid unexpected class initialization errors. Third, our methodology adds the ability for an invariant to quantify over objects (for example, specifying that no two linked-list nodes have the same successor), which involves a syntactic restriction on programs. We present our methodology for a programming language similar to the sequential subset of Java or C#. The only major semantic difference is how class initialization is performed, as explained later.

To support programming in the large, a crucial aspect of any specification and verification methodology is that it be *modular*. That is, it should be possible to reason about smaller portions of a program at a time, say a class and its imported classes, without having access to all pieces of code in the program that use the class or extend the class. Our methodology is modular.

To save space, we combine the three kinds of static class invariants into one running example, the $Service$ and $Client$ classes in Figs. 1 and 2. Objects of class $Service$ represent instances of a system service. These instances share a common job cache, which is referenced from the static field $jobs$. The first class invariant in $Service$ says that $jobs$ is non-null, and the second says that the non-null elements of the cache are distinct. Objects of class $Client$ represent users of the service. Each client has an ID and the static field ids keeps track of the number of IDs ever given out. The first class invariant says that ids is a natural number, the second says that ids exceeds all client IDs, and the third says that clients have unique IDs. Here and throughout, quantifications over objects range over allocated, non-null objects. The quantifications in the class invariants in $Client$ are also restricted to *valid* $Client$ objects, indicated by $c.inv_{Client} = valid$ and explained later. Note that the third invariant in $Client$ does not mention any static fields, but we nevertheless consider it a class invariant since individual objects cannot maintain this invariant. No previous methodology can handle these kinds of invariants

```
class Service imports Client {
  static rep Client[ ] jobs ;
  static invariant Service.jobs ≠ null ;  // simple
  static invariant ( ∀ int i, j | 0 ≤ i < j < Service.jobs.length •
        Service.jobs[i] ≠ null ⇒ Service.jobs[i] ≠ Service.jobs[j] ) ;  // ownership
  static initializer {Service.jobs := new ⟨Service⟩Client[10] ;}
  static void cache(Client c)
    requires Service.sinv = tvalid ∧ c ≠ null ;
  {
    if (c ∉ Service.jobs) {
      int free := arbitrary value in { 0, ..., Service.jobs.length − 1 } ;
      expose Service {
        expose Service.jobs for Client[ ] ;  Service.jobs[free] := c ;
        unexpose Service.jobs for Client[ ] ;
}  }  }  }
```

Fig. 1. *Service* has a static field *jobs*, which references an array of *Client* objects. The class invariants guarantee that *jobs* is not **null** and that each *Client* object is stored at most once in the *jobs* array. Both invariants are established by the static initializer. The **rep** keyword in the declaration of *jobs* indicates an ownership relation between the *Service* class and the array referenced by *jobs*.

in an object-oriented setting where the dynamic call order does not follow any statically determined order on the classes, but our methodology handles all of them.

In Sec. 2, we present our methodology for *simple class invariants*, which talk about the static fields of a class, handling the shared-values use of static fields. In Sec. 3, we extend this methodology to *ownership-based class invariants*, handling the roots-of-object-structures use of static fields. In Sec. 4, we further extend the methodology to *global class invariants*, which quantify over all valid objects of the class, handling the all-instances use of static fields and invariants. We formalize the methodology and state a soundness theorem in Sec. 5. We end the paper with related work and a conclusion.

2 Methodology

In this section, we introduce our methodology for class invariants, explain how we overcome the central problem of abstraction and information hiding, and prescribe class initialization. We focus on the general ideas, tightening up the details in Section 5.

As illustrated by the examples in Figs. 1 and 2, class invariants are declared by clauses of the form **static invariant** P; where P is a predicate that can mention fields. A class C can contain several invariant clauses. *The class invariant of* C, denoted by $ClassInv_C$, refers to the conjunction of all invariant clauses declared in C. In this section, we focus on simple class invariants, that is, invariants where the only fields mentioned in P are static fields of C.

2.1 Basic methodology

Two fundamental issues drive the design of a methodology for class invariants. First, in general, class invariants relate the values of several fields. Therefore, it is not possible

```
class Client imports String {
  int id ;        static int ids ;
  static invariant 0 ≤ Client.ids ; // simple
  static invariant ( ∀ Client c | c.inv_Client = valid • c.id < Client.ids ) ; // global
  static invariant ( ∀ Client c | c.inv_Client = valid •
            ( ∀ Client d | d.inv_Client = valid • c ≠ d ⇒ c.id ≠ d.id )) ; // global
  static initializer { Client.ids := 0 ; }
  Client()
    requires Client.sinv = tvalid ;
    ensures this.inv_Client = valid ;
  {
    expose Client {
      id := Client.ids ;    Client.ids := Client.ids + 1 ;
      unexpose this for Client ;
    }
  }
  static String debugMsg()
    requires Client.sinv = tvalid ;
  { result := "Client objects created : ".appendNat(Client.ids) ; }
}
```

Fig. 2. Every object of class $Client$ has an ID. The next available ID is stored in the static field ids. Class invariants guarantee that ids has not been assigned to a $Client$ object and that all $Client$ objects have different IDs.

to expect class invariants to hold at every program point; we must allow class invariants to be temporarily violated.

Second, it is not possible to completely free clients of the responsibility of making sure the class invariant holds when a method of the class is called. This is because of the possibility of reentrancy: a method m declared in class C can call methods that cause control to reenter C. A problem would occur if m makes such a call at a time C's invariant is temporarily violated and the method through which C is reentered expects the invariant to hold. It would be overly restrictive to forbid method calls while an invariant is temporarily violated. For example, one may want to invoke a method on the data structure rooted in a static field.

To deal with these two fundamental issues, the methodology must permit times when the class invariant becomes violated. For this reason, we introduce a special program statement, **expose** C { s }, which allows the invariant of class C to be violated for the duration of the sub-statement s, throughout which time we say that C is *mutable*. Any update of any static field $C.g$ must take place while C is mutable (but there are no restrictions on when variables can be read). The class invariant is checked to hold at the end of the **expose** block. We define **expose** blocks to be non-reentrant; that is, it is illegal to expose an already mutable class. (Non-reentrancy and condition J2, below, are what guarantee that the class invariant holds on entry to sub-statement s.)

When reasoning modularly about a program, it is important to know whether or not a class is mutable. For example, $Client$'s constructor in Fig. 2 needs to declare a precondition that says class $Client$ is *valid*, that is, not mutable; otherwise, it would not be possible to prove that the program meets the non-reentrancy requirement of the

expose block in the constructor's implementation. To facilitate mentioning the validity status of a class, we introduce for each class C a special static field $C.sinv$ (whose possible values we will describe later), which can be mentioned in method specifications. $C.sinv$ is an abstraction of the static class invariant in C: a specification can mention $C.sinv$ to require C to be valid, which in effect says that C's invariant holds but does not reveal the details of the invariant itself.

A program cannot update the static field $C.sinv$ directly. Instead, the value of $C.sinv$ is changed automatically on entry and exit of each **expose** statement. We postpone until Section 2.4 the issue of setting the initial value of $C.sinv$.

It is important to understand that our methodology does not use a *visible state semantics*, where methods can automatically assume all class invariants to hold in the pre-state. Instead, a method is allowed to rely only on those invariants whose validity follows from the explicit precondition. Conversely, one does not have to prove that all class invariants hold when the method terminates. Instead, we prove that (1) the only static fields that are assigned to are those of mutable classes, and (2) the class invariant of a class C holds at the end of each **expose** C statement.

2.2 Abstraction and information hiding

The special static field $sinv$ makes it possible for a program to record, usually in preconditions of methods, when a class invariant is expected to hold. However, whenever one class uses another, it would be clumsy, at best, to have to mention explicitly in a precondition all classes whose validity is needed. For example, suppose the $String$ class contains a global cache of integers and their $String$ representations. Then, many methods of $String$, including $appendNat$ which is called by $debugMsg$ (Fig. 2), would have a precondition that requires the $String$ class to be valid. Method $debugMsg$, in turn, would then need to declare the precondition that both $Client$ and $String$ are valid. And so on, for the methods of other classes that may transitively call $debugMsg$. As this example suggests, preconditions would become unwieldy. Moreover, if one class deep in a program one day is changed to call a method of $String$, then all transitive callers would have to be changed to add $String$ validity as a precondition. Such a programming methodology would not respect good principles of information hiding.

To address this problem, we derive from the class declarations of a program a partial order on classes, the so-called *validity ordering*, and provide the ability, using the special static field $sinv$, to express the transitive validity of a class. A class C is *transitively valid* (or *t-valid* for short) if the invariant of C holds and all classes that precede C in the validity order are t-valid.

The most common edge in the validity ordering arises when one class is a client of another class. We require that if a class C refers to a class D or to an entity declared in D, then either D is a superclass of C or C is declared explicitly to *import* D. (Note that in the latter case, the import declaration is mandatory, in contrast to Java's "import" construct, which is just a convenient alternative to writing fully qualified names.) If C imports D, then this import also gives rise to the edge $D \leftarrow C$ ("D *precedes* C") in the validity ordering. For instance, class $Client$ imports $String$, which, in particular, allows $debugMsg$ to call a method of $String$. The case where D is a superclass of C is handled conversely as explained below.

It is now time we introduce actual values for the $sinv$ field:

- $C.sinv = tvalid$ says that C is transitively valid, that is, that the invariant of C holds and that all classes that precede C in the validity order are t-valid.
- $C.sinv = valid$ says that the invariant of C holds, but says nothing about the validity of C's predecessors.
- $C.sinv = mutable$ says that C's invariant may be violated and that the program is allowed to execute statements that assign to the static fields of C.

As suggested by these bullets, and as we later shall justify, our methodology guarantees that the following properties are *program invariants*, that is, that they hold at every control point in a program (here and throughout, quantifications over class names range over all classes of a program):

J1: $(\forall\, C, D \bullet\ D \leftarrow C \ \wedge\ C.sinv = tvalid \ \Rightarrow\ D.sinv = tvalid\ )$
J2: $(\forall\, C \bullet\ C.sinv = tvalid \ \vee\ C.sinv = valid \ \Rightarrow\ ClassInv_C\ )$

We can now spell out the preconditions of the methods involved in the Fig. 2 example. Since $Client$ imports $String$, $String$ precedes $Client$ in the validity ordering. Assume the following declaration in class $String$:

$String\ appendNat(\mathbf{int}\ n)$
  $\mathbf{requires}\ 0 \leq n \ \wedge\ String.sinv = tvalid\ ;$

Method $debugMsg$ needs the precondition $Client.sinv = tvalid$, since it not only needs $Client$'s invariant in order to establish that the parameter passed to $appendNat$ is non-negative, but also needs the t-validity of $String$ in order to meet the precondition of $appendNat$. $Client$'s constructor can require $Client.sinv = tvalid$, $Client.sinv = valid$, or $Client.sinv = tvalid \vee Client.sinv = valid$, since the implementation of the constructor does not depend on the validity of other classes. However, $Client.sinv = tvalid$ is generally to be preferred, because that specification is general enough to allow the implementation to be changed to rely on the validity of other classes.

2.3 Subclasses

Since validity-ordering edges are introduced along with the imports relation, a declared class becomes a successor of all classes it imports. In this subsection, we show that subclassing has to be treated differently from other uses-relations between classes.

To illustrate with an example, consider a hierarchy of classes representing decision procedures for various theories, as may be used in the implementation of an automatic theorem prover. Each theory implements a method $assertLiteral$ that adds a constraint to the decision procedure. Fig. 3 declares class $Theory$, the root of the hierarchy.

Now, consider a particular theory, say the theory of linear arithmetic, represented by a subclass $LATheory$, see Fig. 3. Being a method override, $LATheory.assertLiteral$ has the same specification as the overridden $Theory.assertLiteral$, and in particular, the override cannot strengthen the precondition of the overridden method.

Suppose the $LATheory$ implementation of $assertLiteral$ makes use of some static fields of $LATheory$ and relies on the class invariant to hold of these static fields. This means that $LATheory.assertLiteral$ relies on the t-validity of $LATheory$. Since

```
class Theory { void assertLiteral(Literal l) { ... } ... }
class LATheory extends Theory imports String {
  static String version ;
  static invariant LATheory.version ≠ null ;
  static initializer {LATheory.version := "Version ".appendNat(3) ;}
  override void assertLiteral(Literal l) { ... }
  ... }
```

Fig. 3. An example to illustrate the specification problem of a method override that relies on a static class invariant.

this method override cannot strengthen the precondition for $assertLiteral$ defined in $Theory$, the precondition of $Theory.assertLiteral$ must imply that $LATheory$ is t-valid. But how can such a precondition be declared in class $Theory$ without explicitly mentioning $LATheory$ (since $Theory$ may not know about the existence of $LATheory$, which may be authored long after the authoring of $Theory$)?

If $LATheory$ precedes $Theory$ in the validity ordering, then we can solve the specification problem on account of program invariant J1. The method in class $Theory$ then declares the precondition

requires $Theory.sinv = tvalid$;

which by J1 implies $LATheory.sinv = tvalid$, as needed in the method override. In other words, a caller of method $assertLiteral$, which may not even know about the existence of $LATheory$ but may nevertheless hold a reference to an object of allocated type $LATheory$, must establish the t-validity of $Theory$ at the time of call, which allows the implementation of $LATheory$ to determine that $LATheory$ is t-valid, too.

To allow class $LATheory$ to define the edge $LATheory \leftarrow Theory$ in the validity ordering, we use the *extends* relation that is already used to declare subclasses. That is, as part of our methodology, a subclass precedes its superclasses in the validity ordering. A class can extend one superclass (single inheritance) and import any number of other classes. However, we require that the resulting validity ordering is acyclic.

An acyclic validity ordering prevents mutually dependent classes (except when one class is a subclass of the other). Cyclic references between classes can be allowed by grouping classes into *modules* and declaring the validity ordering on modules instead of classes. Then, the classes in one module can mutually depend on each other. We explain and formalize this approach in our technical report [9].

2.4 Class initialization

A static class invariant is first established by the static initializer of the class, a designated block of code that is invoked exactly once. Static initializers are invoked by the runtime system, so as to orchestrate the initialization of multiple classes. For brevity, we do not consider dynamic class loading here, but our methodology can handle it [9].

Since the static initializer of a class C may access fields and methods of imported classes, it requires C's predecessors in the validity ordering to be valid. This is achieved by initializing classes in the order of the validity ordering.

A program is executed by invoking the static method $main$ on a specified class, say M. Before $main$ is actually called, the runtime system loads M and all classes

that M transitively imports or extends. The static fields of all classes are initialized to zero-equivalent values, in particular, $sinv$ is initialized to $mutable$. Next, the runtime system executes the static initializer of each class, according to the validity ordering. After executing the static initializer of a class C, $C.sinv$ is set to $tvalid$.

C's static initializer can, therefore, assume on entry that (1) C is mutable, which allows the initializer to assign to C's static fields, and (2) all predecessors of C are t-valid. That is, C's static initializer may assume that the following precondition holds:

$$C.sinv = mutable \wedge (\forall\, D \bullet\ D \leftarrow C \ \Rightarrow\ D.sinv = tvalid\ )$$

The initializer is responsible for making sure the implicit assertion $\mathbf{assert}\ ClassInv_C$ holds on exit. For example, consider class $LATheory$ in Fig. 3. Because $String$ precedes $LATheory$, the second conjunct of the precondition implies $String.sinv = tvalid$; therefore, the initializer can meet the precondition of $appendNat$. Because of the first conjunct of the precondition, the assignment to $LATheory.version$ is permitted. Provided $appendNat$ returns a non-null value, the implicit assertion at the end of the initializer body will hold.

Note, by the way, that the $LATheory$ initializer cannot assume $Theory$ to be t-valid, since $Theory$ does not precede $LATheory$. This is different from the initialization order in Java, where superclasses are initialized before their subclasses. Most correct programs that require that superclass invariants are established before subclasses are initialized can be modeled or rewritten to follow Java's initialization ordering. The key idea is to separate out static fields and invariants of the superclass into a helper class, which is imported by both the superclass and the subclasses.

2.5 Summary

We summarize the steps that lead us to our methodology: Class invariants can state relations between multiple static fields, and thus the methodology must permit class invariants to be temporarily violated. To allow calls while a class invariant is violated and since such calls may reenter the class, we explicitly represent (by $sinv$) whether or not a class invariant might be violated, which allows preconditions to be explicit about which invariants are assumed to hold. The explicit representation reveals the central problem of abstraction, which we address by allowing classes to be ordered (by the validity ordering) and by representing transitive validity of classes along that ordering. Finally, the validity ordering has an impact on class initialization.

The methodology allows programmers to specify invariants on the state of each class. The programmer is assured that the invariant of a class C holds whenever $C.sinv$ is $valid$ or $tvalid$. Thus, by requiring that, for example, $C.sinv = tvalid$ holds on entry to a method, the implementation of the method can safely rely on C's class invariant to hold on entry. Dependencies between classes are indicated by edges in the validity ordering, which coupled with our initialization order avoids class initialization errors.

3 Ownership-based class invariants

Simple class invariants refer only to static fields of the enclosing class, but not to instance fields. Preventing class invariants from depending on instance fields is too restrictive for many interesting programs. For instance, class $Service$ (Fig. 1) uses a global

cache, which is implemented by an array and rooted in the static field $jobs$. $Service$ imposes restrictions on the elements stored in this array object.

Assume that the class invariant of a class C refers to the instance field f of an object X. The reason the methodology introduced so far cannot handle such class invariants is that a method m outside C that gets hold of a reference to X can update $X.f$, thereby potentially violating C's invariant. Since the invariant might not be known to m, it is not possible to determine modularly that this update has to be guarded by an assertion that C is mutable. In our example, any method that has a reference to the $Service.jobs$ array can break the $Service$ invariant by assigning to elements of the array.

In this section, we extend our methodology by the notion of *ownership*. This extension allows the invariant of a class C to depend on fields of objects *owned* by C without restricting where these fields are declared. The extended methodology ensures that a field of an object can be updated only if the object's owning class is mutable.

3.1 Ownership

Ownership organizes objects into a hierarchy of *contexts*, where the objects in each context have a common owner (see, *e.g.*, [5, 16]). In this paper, we use a restricted form of ownership where objects can be owned by a class, but not by other objects. This restriction allows us to focus on the methodology for class invariants without getting into details of the corresponding methodology for object invariants. An extension to ownership among objects is presented in our report [9]. We do not restrict references; classes and objects may have non-owning references to objects.

Following the encoding in our work on object invariants [10], we represent ownership by a special field $owner$ for every object. The value of $owner$ is a class name. It is set when an object is created. The allocation statement $x := \mathbf{new}\ \langle C \rangle\, T$ creates a new object of class T owned by class C. In this paper, we assume $owner$ to be immutable after object creation. We described how to handle a mutable $owner$ field for objects (ownership transfer) in a previous paper [10].

We allow the $owner$ field to be mentioned in class invariants. To specify ownership relations, we introduce a modifier $\mathbf{rep}$ that can be used in the declaration of any static field. A field declaration $\mathbf{static\ rep}\ S\ g;$ in a class C gives rise to the following implicit class invariant about ownership in C:

$$C.g \neq \mathbf{null} \Rightarrow C.g.owner = C\ ;$$

The invariant of a class C is allowed to refer to fields of objects owned by C. In our example, the $\mathbf{rep}$ modifier of the static field $Service.jobs$ indicates that the object referenced by $Service.jobs$ is owned by $Service$, which allows the class invariant of $Service$ to refer to the fields (that is, array elements) of $jobs$. Accessing array elements is handled analogously to field access, as if each element were a field.

3.2 Mutability of owned objects

Analogously to the static field $sinv$ and following our methodology for objects, each object type (class or array type) C declares a two-valued field inv_C that indicates whether the object invariant declared in C may be assumed to hold. If $X.inv_C =$

$valid$, we say that object X *is valid for* C, or just X *is valid* if the object type is clear from the context. Conversely, we say X *is mutable for* C if $X.inv_C = mutable$. An instance field f declared in C can be assigned to only if the instance is mutable for C. That is, an update $X.f := E$ is guarded by the precondition $X.inv_C = mutable$.

Consider a class C that owns an object X. C's invariant is allowed to depend on $X.f$ even if f is declared in another class. Consequently, an update of $X.f$ may potentially violate C's invariant. Our methodology handles this situation by the following rule: If a class C is valid, then all objects owned by C are valid for their object type and all supertypes. That is, if X is mutable for D so that $X.f$ can be assigned to (where f is declared in class D), then X's owner, C, is also mutable, so violations of C's static class invariant are allowed.

This rule is enforced by manipulating the inv_D field according to a strict protocol: inv_D can be manipulated only by statements analogous in functionality to **expose** for classes. However, although exposing and unexposing objects typically is done in a block-structured way, it does not have to be. In particular, some constructors, like $Client$'s constructor in Fig. 2, unexpose the newly created object without previously exposing it. Therefore, instead of an **expose** block statement for objects, we use two separate statements, **expose** X **for** D and **unexpose** X **for** D, which expose and unexpose an object X for a class D, respectively. When applied to a valid object X and for a class D, **expose** X **for** D checks that X's owner is mutable and sets $X.inv_D$ to $mutable$. **unexpose** X **for** D checks that X is mutable and sets $X.inv_D$ to $valid$. When an object of class C is created, its inv_D fields start off as $mutable$ for all superclasses D of C.

The $cache$ method of class $Service$ in Fig. 1 illustrates how ownership-based invariants are handled. It requires $Service$ to be t-valid. To satisfy the precondition $Service.jobs.inv_{Client[\,]} = mutable$ of the update $Service.jobs[free] := c$, $Service$ has to be mutable. The **expose** statement sets $Service.sinv$ to $mutable$, which makes the $jobs$ array exposable and allows updates to temporarily violate $Service$'s class invariant. In our example, the invariant is not actually violated by the update, because we insert c only if it is not already contained in the array.

Because of lack of space and to focus on static class invariants, we do not present in this paper the complete object-centered methodology that allows a program to ensure its field updates apply only to mutable objects, but see [1, 10].

4 Global class invariants

In this section, we explain how our methodology allows class invariants to quantify over all valid objects of the enclosing class.

4.1 Quantification over valid objects

A class invariant of a class C is allowed to universally quantify over all C objects that are valid for C and to refer to those instance fields of these objects that are declared in C. For example, $Client$'s third invariant (Fig. 2) quantifies over valid $Client$ objects and refers to the id field declared in $Client$. We only let a class invariant quantify over

valid objects, because during the time when an object is being updated, which occurs when the object is mutable, the object cannot be expected to satisfy all invariants.

A global class invariant of a class C is potentially violated by unexposing C objects for C, since making a C object valid for C enlarges the range of the quantification in the invariant. Therefore, additional requirements for **unexpose** are needed to guarantee that a class C is mutable whenever one of its objects is unexposed for C. Note that updates of instance fields do not require additional proof obligations for global class invariants, because only fields of mutable objects can be updated.

For soundness, it is sufficient to guard the statement $\mathbf{unexpose}\ X\ \mathbf{for}\ C$ by a precondition $C.inv = mutable$. However, stronger requirements are necessary to achieve a practical solution, as we discuss next.

4.2 Practicality

Assume that class $Client$ has a method $Client\ Foo()$ and that we want to verify the statement $\mathbf{expose}\ Client\ \{\ v := X.Foo();\ \}$, where X is a valid $Client$ object. To prove that $Client$'s class invariant holds at the end of the **expose** block, we have to show in particular that the call to Foo does not create new valid $Client$ objects that violate the global class invariants. For instance, the following implementation of Foo does violate $Client$'s third invariant:

$Client\ Foo()$
  $\mathbf{requires}\ Client.inv = mutable \wedge \mathbf{this}.inv_{Client} = valid\ ;$
{
  $\mathbf{result} := \mathbf{new}\ \langle Object \rangle Client()\ ;\ \mathbf{result}.id := \mathbf{this}.id\ ;$
  $\mathbf{unexpose\ result\ for}\ Client\ ;$
}

Since allocation and initialization of objects is typically considered an implementation detail [8, 11], Foo's specification will in general be too weak to determine whether Foo creates valid objects, which makes it impossible to verify the **expose** block above.

To be able to reason about the effects of a method call on a global class invariant, we impose a syntactic requirement that prevents methods and constructors from unexposing objects for C, if called in a state in which class C is mutable: each $\mathbf{unexpose}\ X\ \mathbf{for}\ C$ operation has to be textually enclosed by C's static initializer or by an $\mathbf{expose}\ C$ block. This syntactic requirement guarantees that $C.inv = mutable$ holds before making an object valid for C . That is, we do not have to impose this condition as a precondition for unexposing C objects explicitly.

This syntactic requirement prevents a method called from within an $\mathbf{expose}\ C$ block from unexposing objects for C, in particular, newly created objects. This property gives rise to an implicit postcondition that allows one to verify the $\mathbf{expose}\ C$ block. In our example above, Foo does not meet the requirement because it unexposes **result** outside an $\mathbf{expose}\ Client$ block.

5 Technical treatment

In this section, we define precisely which invariants are admissible, explain the proof obligations that are necessary to maintain the program invariants J1 and J2 (Sec. 2.2), and present a soundness theorem.

5.1 Admissible invariants

A class invariant of class C can refer to static fields of C, instance fields of objects owned by C, and, by quantification, instance fields of valid C objects:

Definition 1 (Admissible class invariant). *A class invariant declaration in class C is admissible if its subexpressions typecheck according to the rules of the programming language and if each of its field-access expressions has one of the following forms:*

1. $C.g$.
2. $C.g.f$ *where* $C.g$ *is declared* $\mathbf{rep}$.
3. $o.f$ *where* f *is declared in* C *and* o *is bound by a quantification of the form* $(\forall\, C\ o \mid o.inv_C = valid \bullet \ \ldots\ o.f\ \ldots)$

The static field g must not be the predefined field $sinv$, and the instance field f must be different from all inv_T fields.

Simple class invariants contain only access expressions of Case 1. Access expressions of Case 2 allow ownership-based class invariants to depend on fields of objects owned by C. Invariants that contain access expressions of Case 3 are global.

5.2 Proof rules

The methodology presented in this paper does not assume a particular programming logic to reason about programs and specifications. Special rules are required only for class initialization and those statements that deal with the $sinv$ and inv_T fields (static and instance field update, class $\mathbf{expose}$, and object $\mathbf{expose}$ and $\mathbf{unexpose}$) as well as $owner$ (object creation). In this subsection, we present these rules and explain why they are necessary to maintain the program invariants J1 and J2 presented in Sec. 2.2.

The proof rules are formulated in terms of assertions, which cause the program execution to abort if evaluated to $\mathbf{false}$. Proving the correctness of a program therefore amounts to statically verifying that the program does not abort due to a violated assertion. To do that, each assertion is turned into a proof obligation. One can then use an appropriate program logic to show that the assertions hold (*cf.* [18, 7]). All of the proof obligations can be generated and shown modularly. For the proof, one may assume that the program invariants J1 and J2 hold.

Class loading and initialization. The program invariants J1 and J2 are first established during class loading and initialization. Program execution starts with a class loading phase, followed by a class initialization phase. In the loading phase, each class of the program is loaded and its static fields are set to zero-equivalent values. The zero-equivalent value for $sinv$ is $mutable$. This guarantees that all classes are mutable after the loading phase, which implies that both J1 and J2 hold.

In the following initialization phase, classes are initialized according to the validity ordering, that is, a class C is initialized after its predecessors in the validity ordering. For each class C, C's static initializer is called before setting $C.sinv$ to $tvalid$. Since $C.sinv$ is set to $mutable$ by the loading phase and since C's predecessors in the validity ordering are initialized before C, the precondition of C's static initializer is established (see Sec. 2.4). In particular, all predecessors of C are t-valid. The postcondition of this initializer, $ClassInv_C$, guarantees that J2 is preserved when $C.sinv$ is

set to $tvalid$. Since C's predecessors are tvalid, J1 is preserved as well. Consequently, both J1 and J2 hold after the initialization phase.

The static initializer of a class D can create valid objects only for D's predecessors in the validity ordering. Consider a class C that is *not* a predecessor of D. D's static initializer cannot expose C since C is mutable, that is, the precondition of $\mathbf{expose}\ C$ is not satisfied. Therefore, it cannot unexpose an object for C since the $\mathbf{unexpose}\ X\ \mathbf{for}\ C$ statements can occur only within $\mathbf{expose}\ C$ blocks and C's static initializer. Consequently C's static initializer may assume the precondition $(\forall\, C\ X \bullet\ X.inv_C = mutable\ )$, which is important to prove that it establishes C's global class invariants.

Static field update. Updating a static field cannot affect program invariant J1. For J2, we have to ensure that a static field update does not break the invariant of a valid or t-valid class C. The only static fields C's class invariant can refer to are static fields of C (Def. 1). Consequently, we can maintain J2 by requiring C to be mutable, which is enforced by guarding each static field update of the form $C.f := E$ by the check $\mathbf{assert}\ C.sinv = mutable$.

Instance field update. Program invariant J1 is trivially preserved. An update $X.f := E$ potentially breaks the class invariant of a class C if (1) X is owned by C (ownership-based invariants) or (2) f is declared in C and X is valid for C (global invariants). The check $\mathbf{assert}\ X.inv_C = mutable$ guarantees that (1) X's owner class is mutable (see proof rule for $\mathbf{expose}$) and (2) X is not valid for C.

Class expose. As explained in Sec. 2.1, $\mathbf{expose}\ C\ \{\ s\ \}$ essentially sets $C.sinv$ to $mutable$, executes s, and restores the original value of $C.sinv$. To prevent reentrant $\mathbf{expose}$ blocks, an assertion checks that C is not already mutable before the statement. Program invariant J2 is maintained by asserting that C's class invariant holds before $C.sinv$ is restored.

Maintaining J1 is a bit more involved. Changing $C.sinv$ from $tvalid$ to $mutable$ implies that C's t-valid successors in the validity ordering are no longer t-valid, but just valid. Therefore, for each class D that transitively succeeds C (that is, $C \leftarrow D$), if $D.sinv = tvalid$, then the $\mathbf{expose}$ statement temporarily changes $D.sinv$ to $valid$. At the end of the $\mathbf{expose}$ block, the initial values of the $D.sinv$'s are restored. This results in the following pseudo code for $\mathbf{expose}$:

$$
\begin{array}{l}
\mathbf{expose}\ C\ \{\ s\ \}\ \equiv \\
\quad \mathbf{assert}\ C.sinv \neq mutable\ ; \\
\quad \mathbf{let}\ Q = \{\ D\ \mid\ C \leftarrow D\ \wedge\ D.sinv = tvalid\ \}\ ; \\
\quad \mathbf{foreach}\ D \in Q\ \{\ D.sinv := valid\ ;\ \} \\
\quad C.sinv := mutable\ ; \\
\quad s\ ; \\
\quad \mathbf{assert}\ ClassInv_C\ ; \\
\quad \mathbf{foreach}\ D \in \{C\} \cup Q\ \{\ D.sinv := \mathbf{old}(D.sinv)\ ;\ \}
\end{array}
$$

Object expose. $\mathbf{expose}$ and $\mathbf{unexpose}$ for objects do not modify $sinv$ of any class, so J1 is preserved. Exposing an object cannot break a class invariant. $\mathbf{expose}\ X\ \mathbf{for}\ C$ requires X's owner to be mutable before setting $X.inv_C$ to $mutable$ to maintain the property that an object can be mutable only if its owner class is mutable. Since

$\mathbf{unexpose}\ X\ \mathbf{for}\ C$ modifies only the field $X.inv_C$, the only class invariant that can be potentially broken by this operation is a global class invariant in class C. As discussed in Sec. 4.2, a syntactic requirement guarantees that C is mutable at the time when X is unexposed for C, so no extra precondition is required. The Boogie methodology for object invariants requires X's object invariant to hold before X is unexposed. We omit this assertion since we do not consider object invariants in this paper. In summary, we have the following pseudo code for $\mathbf{expose}$ and $\mathbf{unexpose}$:

$\mathbf{expose}\ X\ \mathbf{for}\ C\ \equiv$
  $\mathbf{assert}\ X \neq \mathbf{null} \wedge X.inv_C = valid\ ;$
  $\mathbf{assert}\ X.owner.sinv = mutable\ ;$
  $X.inv_C := mutable$

$\mathbf{unexpose}\ X\ \mathbf{for}\ C\ \equiv$
  $\mathbf{assert}\ X \neq \mathbf{null} \wedge X.inv_C = mutable\ ;$
  $X.inv_C := valid$

Object creation. Again, program invariant J1 is trivially preserved. As explained in Sec. 3.2, the created object has its inv_T fields set to $mutable$ and its $owner$ field initialized with the class C given in the creation expression. These assignments have no impact on class invariants with field-access expressions of Forms 1 (no static field involved), 2 (the new object is not referenced from a static field), or 3 (the new object is not valid for its class) of Def. 1. Since the new object is mutable, its owner class, C, has to be mutable as well, which is checked by the precondition $C.sinv = mutable$.

5.3 Soundness

A program **P** is well-formed if **P** is syntactically correct and type correct, **P**'s invariants are admissible (Def. 1), and the syntactic requirement for $\mathbf{unexpose}$ (Sec. 4.2) is met.

Theorem 1. *In each reachable state of a well-formed program, J1 and J2 hold.*

For a lack of space, we do not present the soundness proof in this paper. We have explained the arguments of the soundness proof along with the presentation of the proof rules. The complete proof is found in our technical report [9].

6 Related Work

Classical proof systems for objects and invariants such as Meyer's work [15] or the approach of Liskov, Wing, and Guttag [13, 12] do not consider static fields or quantification over objects.

JML [8, 4] provides both object and class invariants (called instance and static invariants, respectively). Object invariants may refer to static fields, but class invariants cannot refer to the states of objects. In contrast to our work, JML applies a visible state semantics, where invariants have to hold in the pre- and post-states of all non-helper methods. It does not provide a sound modular proof system.

The use of static fields is sometimes considered bad programming style that can be avoided by using instance fields of a singleton object, u. Simple and ownership-based class invariants can then be expressed as an object invariant of u. However, such a programming model requires that all objects that need access to the shared state of u have references to u and can expose and modify u. Therefore, reasoning about the

shared object, in particular, about the validity of u, is tedious. It is generally based on the fact that u is a singleton, which is difficult to express by standard object invariants.

Eiffel's *once methods* [14] provide a better abstraction mechanism for shared objects. A once method computes its result when it is called the first time. This result is cached and then returned upon succeeding calls. Therefore, each object can access shared data (in particular, a reference to the singleton u) through a once method instead of storing the reference in a field. Validity of u can then be guaranteed by the postcondition of the once method returning the reference. However, since u may not be valid in all execution states in which the method might be called, an additional flag is needed for each once method, indicating whether the object returned by the method is valid. The methodology required to maintain such a flag is identical to our methodology for class invariants and the $sinv$ field.

The Boogie methodology for object invariants [1, 3, 10] does not admit class invariants. However, visibility-based object invariants [3, 10] can be generalized to allow object invariants to mention static fields. For instance, $Client$'s second class invariant can also be expressed by the object invariant $\mathbf{this}.id < Client.ids$. With such object invariants, a static field update potentially violates the object invariant of many objects, all of which would have to be exposed before the update. Barnett and Naumann [3] show that *update guards* can be used to exploit monotonicity properties to avoid exposing all objects possibly affected by a field update. An update guard specifies a condition under which a field update is guaranteed not to break an invariant. For instance, increasing the value of $Client.ids$ cannot violate the above invariant for any $Client$ object.

Like our work, Pierik *et al.* [17] extend the Boogie methodology to class invariants. They handle simple class invariants in the same way as we do. Ownership-based class invariants are not supported. Therefore, class invariants can refer to instance fields only in a limited way. Invariants are allowed to quantify over all objects of a class, for example, to specify that a singleton object is the only instance of a class. Invariants that quantify over all objects of a class rather than over all *valid* objects can be broken by object creation. Therefore, one has to expose a class before creating an instance of it, an obligation that unfortunately falls on the client of a class. The client is then responsible for reestablishing the class invariant. Alternatively, a client can prove that it establishes a *creation guard*, which specifies a condition under which an object creation is guaranteed not to break an invariant. However, a creation guard cannot refer to the newly allocated object, so it is typically $\mathbf{false}$. Pierik *et al.* do not address either the abstraction problem for class invariants or the initialization-order problem for classes.

Müller's thesis [16] also uses a visible state semantics for object invariants. It supports invariants over so-called abstract fields in a sound way, which we consider future work for the methodology presented here.

Leino and Nelson [11] developed a modular treatment of object invariants over abstract fields, which was used in the Extended Static Checker for Modula-3 [6]. Leino and Nelson treat some aspects of class invariants, but neither Müller's nor Leino and Nelson's work fully supports class invariants.

7 Conclusions

We have presented a verification methodology for class invariants, which allows class invariants to specify properties of static fields, of object structures rooted in static fields, and of all valid objects of a class. The methodology is sound and covers all typical applications of static fields we have found in programs. This work is part of a larger effort to advance programming theory to catch up with the current programming practice.

As future work, we plan to build on our previous work on visibility-based invariants [10] to support less common class invariants that refer to static fields and that quantify over objects of other classes. Moreover, we plan to implement our methodology as part of the .NET program checker Boogie, which is part of the Spec# programming system [2].

References

1. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6), 2004. `www.jot.fm`.
2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS 2004*, volume 3362 of *LNCS*, pages 49–69. Springer-Verlag, 2004.
3. M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *MPC 2004*, volume 3125 of *LNCS*, pages 54–84. Springer-Verlag, July 2004.
4. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer (STTT)*, 2004.
5. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA '98*, pages 48–64. ACM, October 1998.
6. D. L. Detlefs, K. R. M. Leino, G. Nelson, and J. B. Saxe. Extended static checking. Research Report 159, Compaq SRC, December 1998.
7. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI 2002*, pages 234–245. ACM, 2002.
8. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev27, Iowa State University, 2003.
9. K. R. M. Leino and P. Müller. Modular verification of global module invariants in object-oriented programs. Technical Report 459, ETH Zürich, 2004.
10. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP 2004*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.
11. K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *TOPLAS*, 24(5):491–553, September 2002.
12. B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science Series. MIT Press, 1986.
13. B. Liskov and J. M. Wing. A behavioral notion of subtyping. *TOPLAS*, 16(6):1811–1841, 1994.
14. B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
15. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1997.
16. P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002. PhD thesis.
17. C. Pierik, D. Clarke, and F. S. de Boer. Controlling object allocation using creation guards. In *Formal Methods (FM 2005)*, LNCS. Springer-Verlag, 2005. In this volume.
18. A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In *ESOP 1999*, volume 1576 of *LNCS*, pages 162–176. Springer-Verlag, 1999.