Modular verification of global module invariants in object-oriented programs

K. Rustan M. Leino
Microsoft Research
Redmond, WA, USA
leino@microsoft.com

Peter Müller
ETH Zurich
Switzerland
peter.mueller@inf.ethz.ch

Manuscript KRML 139, 26 July 2004, draft.

Abstract

Modules and objects both contain variables whose values may be constrained by invariants. For example, in the object-oriented languages Java and C#, a module is a class and its static fields, and an object is an instance of a class and its instance variables. The invariants of modules work differently both from the invariants of objects alone and from the invariants of modules in a procedural language. This paper presents a methodology for module invariants in an object-oriented setting. The methodology is sound, prescribes an initialization order of a program's modules, supports the dynamic loading of modules and classes, and is amenable to static, modular checking.

0 Introduction

In a computer program, a *module* is a collection of variables and procedures. An *object*, too, is a collection of variables and procedures. The difference is that a program contains one instance of a module, whereas it may contain any number of object instances. Moreover, the identity of a module is a name that can be mentioned anywhere in a program, whereas the identity of an object is a reference that is created through dynamic allocation and is accessible only if this reference flows as a value to the site of use. Objects are therefore used in dynamic data structures whereas modules are entities that are shared in a program. In programming, there are uses of both modules and objects.

Both modules and objects can have data, represented, respectively, as *module variables* (also called *global variables*) and *instance variables* (also called *fields*). The data values in each can be constrained by programmer-declared *invariants*, as they are in, for example, the Java Modeling Language [14]. The correctness of a program relies on these invariants. Consequently, any tool or technique for checking the correctness of the program—manual or automatic, static or dynamic—depends on these invariants and a methodology stating their exact meaning and guiding their use.

In this paper, we consider the specification and verification of module invariants in object-oriented programs, which, perhaps surprisingly, require a methodology different from that of module invariants in procedural languages and different also from that of object invariants. We prove that our methodology is sound for modular verification, meaning that the separate verification of each module implies the correctness of the whole program.

Our modules are similar to those in, for example, the object-oriented language Modula-3. A module is a named scope containing declarations of variables, procedures, and classes. In some popular object-oriented languages, including Java and C#, the notion of a module has been combined with the notion of a dynamically instantiable class. To see the connection between such languages and our paper, a Java or C# class T can be modeled as a module MT whose variables are the static fields of T, whose procedures are the static methods of T, and whose classes are a single class CT, where CT is T with all static members removed.

Module invariants have been well understood since the pioneering work of, especially, Hoare [11]. The solution is to use a methodology that insists that each module's invariant hold on the module's control boundaries, that is, at each call into and return from a procedure defined in the module. The module invariant thus holds whenever the program's control is outside the module. This solution requires that the module's variables can be modified only by the module's procedures and that modules cannot be reentered. These requirements are easily met by a procedural language with modules. In particular, the first requirement can be met by a language-defined variable-accessibility policy; the second requirement can be met by requiring a module to *import* another module if it calls a procedure in the other module and requiring the imports relation among modules to be acyclic.

This classic methodology for module invariants is insufficient in object-oriented programs where subtyping and dynamically dispatched methods give rise to useful patterns of reentrancy into an object's procedures. In these programs,

the dynamic call structure between modules does not follow the static imports relation between modules.

Because of reentrancy, object invariants, too, require a richer methodology than one analogous to the classic methodology for module invariants [0, 16, 1]. As part of the support for object invariants, it seems useful to partition a program's objects into *contexts*, which are hierarchically ordered by an ownership relation. This methodology is not suitable for module invariants, because a module generally cannot be "owned" by just one other module.

To our knowledge, the methodology we present in this paper is the first methodology for module invariants in object-oriented programs.

As part of our methodology for module invariants, we prescribe a scheme for initializing modules. Our scheme is flexible enough to handle dynamic module and class loading. The scheme is different from the initialization schemes of Java and C#. Nevertheless, the assumptions we make are weak enough to be applicable to practical programming languages.

Given the combination of modules and objects, it is interesting to consider not just invariants over global variables alone or fields alone, but also to consider invariants over a combination of global variables and fields: shared data is often a dynamically allocated data structure rooted from a global variable, and objects that share some data may need invariants about their relation to the shared data. In this paper, we include treatment of the first combination, but not the second combination.

The rest of this paper is structured as follows. We start by motivating and describing our core methodology (Sec. 1). We then formalize the ideas (Sec. 2) and prove a soundness theorem (Sec. 3). Using ownership, we extend the methodology to allow module invariants over more dynamic structures (Sec. 4), for which we also prove a soundness theorem (Sec. 5). We end the paper with some discussion, related work, and conclusions.

1 Methodology

In this section, we introduce our basic methodology for module invariants, explain how we overcome the central problem of abstraction in a multi-module setting, identify an issue that arises when subclasses are declared in different modules, and prescribe the initialization of modules. We focus on the general ideas, tightening up the details in the next section.

1.0 Basic methodology

We allow every module to declare an invariant over its module variables. For example, the module in Fig. 0 declares the invariant $B.x \leqslant B.y$.

```
module B imports STRING {
  int x ;
  int y ;
  invariant B.x ⩽ B.y ;
  procedure increase( ) {
    expose B {
      B.x := B.x + 1 ; // invariant might be violated here
      B.y := B.y + 2 ; // invariant is now restored
    }  // invariant is checked to hold here
    B.x := 15 ;  // illegal assignment; allowed only during
                 // execution of an expose block
  }
  procedure STRING.String getDifference( ) {
    result := STRING.natToString(B.y − B.x) ;
  }
}
```

Figure 0: An example program showing a module with two variables, an invariant, and two procedures. The preconditions of the shown procedures have been omitted in this figure; they are discussed in the text. Not shown in this example is the fact that modules can also declare instantiable classes.

Since the invariant may relate the values of several variables, the methodology must permit times when the module invariant becomes violated. For this reason, we introduce a special program statement

$$\textbf{expose}\ A\ \{\ S\ \} \tag{0}$$

which allows the invariant of module A to be violated for the duration of the sub-statement S, throughout which time we say that A is *exposed*. Any update of any variable $A.x$ must take place while A is exposed (but there are no restrictions on when variables can be read). The module invariant is checked to hold at the end of the **expose** block. So that the module invariant can be relied on just inside an **expose** block, **expose** blocks are non-reentrant. That is, it is illegal to expose an already exposed module.

When reasoning modularly about a program, it is important to know whether or not a module is exposed. For example, procedure $B.increase$ in Fig. 0 would want to declare a precondition that says module B is *valid*, that is, not exposed; otherwise, it would not be possible to prove that the program meets the non-reentrancy requirement of the **expose** block in the procedure's implementation. To facilitate mentioning the validity status of a module, we introduce for each module A a special variable $A.si$ (whose possible values we'll describe later), which can be mentioned in procedure and method specifications. Note that $A.si$ is an abstraction of the invariant in A: a specification can mention $A.si$ to require A to be valid, which in effect says that A's invariant holds but doesn't give the details of the invariant itself.

A program cannot update $A.si$ directly. Instead, the value of $A.si$ is changed automatically on entry and exit of

each **expose** statement (0). We postpone until Section 1.3 the issue of setting the initial value of $A.si$.

1.1 Multi-module issues

The special module variable si makes it possible for a program to record, usually in preconditions of procedures and methods, when each module's invariant is expected to hold. However, whenever one module uses another, it would be clumsy, at best, to have to mention explicitly in a precondition all modules whose validity is needed. For example, suppose the $STRING$ module contains a global cache of integers and their $String$ representations. Then, many procedures and methods in $STRING$, including $natToString$ which is called in Fig. 0, would have a precondition that requires the $STRING$ module to be valid. Procedure $B.getDifference$, in turn, would then need to declare the precondition that both B and $STRING$ are valid. And so on, for the procedures of other modules that may transitively call $B.getDifference$. Moreover, if one module deep in a program one day is changed to call a procedure in $STRING$, 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 provide the ability, using the special module variable si, to express the *transitive validity* (or *t-validity* for short) of a module. It is now time we introduce actual values for the si variables:

- $A.si = tvalid$ says that A is transitively valid, that is, that the invariant of A holds and that all modules that precede A in the validity order (defined below) are t-valid.
- $A.si = valid$ says that the invariant of A holds, but says nothing about the validity of A's predecessors.
- $A.si = mutable$ says that A's invariant may be violated and that the program is allowed to execute statements that assign to the module variables of A.

As suggested by these bullets, and as we later shall prove, our methodology guarantees that the following properties are *program invariants*, that is, that they hold at every point in a program:

J0: $(\forall A, B \bullet B \leftarrow A \wedge A.si = tvalid \Rightarrow B.si = tvalid)$

J1: $(\forall A \bullet A.si = tvalid \vee A.si = valid \Rightarrow ModuleInv(A))$

where $B \leftarrow A$ says that B precedes A in the validity order and $ModuleInv(A)$ denotes the invariant declared in module A.

Having introduced these values, we can now spell out the preconditions of the procedures involved in the Fig. 0 example. Let's assume $STRING$ precedes B in the validity ordering and assume the following declaration in module $STRING$:

$$\textbf{procedure}\ STRING.String\ natToString(\textbf{int}\ n)$$
$$\textbf{requires}\ 0 \leqslant n \wedge STRING.si = tvalid\ ;$$

Procedure $B.getDifference$ needs the precondition $B.si = tvalid$, since it not only needs B's invariant in order for the parameter passed to $natToString$ to be non-negative, but also needs the t-validity of $STRING$. Procedure $B.increase$ can use either $B.si = tvalid$ or $B.si = tvalid \vee B.si = valid$ as its precondition. However, the former is generally to be preferred, or the specification would not allow the implementation to in the future rely on the validity of other modules.

The *validity ordering* is a programmer-specified partial order on the modules of a program. We could introduce a special declaration for introducing edges in the validity ordering, but since the ordering tends to follow certain programming patterns, we instead tie the introduction of validity-ordering edges to other declarations. The most common edge in the validity ordering arises when one module is a client of another module. Therefore, we piggyback the introduction of validity-ordering edges to the imports relation: if a module A is declared to import a module B, then this import also gives rise to the edge $B \leftarrow A$ ("*B precedes A*"). In other words, the validity ordering includes the imports relation on modules.

The validity order on modules is a partial order, but it is generally not a hierarchical tree order (unlike the somewhat analogous ownership relation on objects, used to reason about object invariants [0, 16]). This requires some care in our refined definition of the **expose** statement. In the presence of transitive validity, the precondition of statement (0) is

$$A.si = tvalid \ \vee \ A.si = valid$$

The statement temporarily changes $A.si$ to $mutable$. Moreover, for each module Z that transitively succeeds A (that is, A transitively precedes Z), if $Z.si = tvalid$, then the **expose** statement (0) temporarily changes $Z.si$ to $valid$. At the end of the **expose** block, the initial values of $A.si$ and the $Z.si$'s are restored. The reason for temporarily changing these $Z.si$ from $tvalid$ to $valid$ is to maintain program invariant J0.

1.2 Modules and subclasses

In the validity-ordering edges introduced along with the imports relation, a declared module becomes a successor of all modules it imports. But there are cases when one wants to "insert" a module as a predecessor of some other module. In this subsection, we give a motivating example and set up a way to introduce such a validity-ordering edge.

```
module TH {
  class Theory { method assertLiteral(Literal l) ; ... }
  ...
}
module ARITH ... {
  class LATheory extends TH.Theory {
    override assertLiteral(Literal l) { ... } ...
  }
  ...
}
```

Figure 1: An example to illustrate the specification problem of a method override that relies on a module invariant.

Consider a hierarchy of classes representing decision procedures for various theories, as may be used in the implementation of an automatic theorem prover (*cf.* [7]). Each theory has a method $assertLiteral$ that adds a constraint to the decision procedure. Fig. 1 declares class $Theory$, the root of the hierarchy, enclosed in a module TH.

Now, consider a particular theory, say the theory of linear arithmetic, represented by a subclass $LATheory$ declared in a different module, $ARITH$, see Fig. 1. Being a method override, $LATheory.assertLiteral$ has the same specification as $Theory.assertLiteral$, and in particular, the override cannot strengthen the precondition of the overridden method.

Suppose the $LATheory$ implementation of $assertLiteral$ makes use of some $ARITH$ module variables (perhaps indirectly, by calling some procedure in TH) and relies on the module invariant to hold of these variables. The specification problem is then how to declare a precondition for $Theory.assertLiteral$ that is strong enough to imply $ARITH.si = tvalid$ (for the benefit of the method override in $LATheory$), but without explicitly mentioning $ARITH$ in TH (since TH may not know about the existence of $ARITH$, which may be authored long after the authoring of TH).

If $ARITH$ precedes TH in the validity ordering, then we can solve the specification problem on account of program invariant J0. The method in class $Theory$ then declares the precondition

$$\textbf{requires } TH.si = tvalid \ ;$$

which by J0 implies $ARITH.si = tvalid$, as needed in the method override. In other words, a caller of method $assertLiteral$, which may not even know about the existence of $ARITH$ but may nevertheless hold a reference to an object of allocated type $LATheory$, must establish the t-validity of TH at the time of call, which gives the implementation of $LATheory$ enough information to determine that $ARITH$ is t-valid, too.

To allow module $ARITH$ to define the edge $ARITH \leftarrow TH$ in the validity ordering, we introduce a variant of the imports relation. We call the variant *extends*, and it is used as following in the example:

$$\textbf{module } ARITH \textbf{ extends } TH \ \{ \ \ldots \ \} \qquad (1)$$

The extends relation is just like the imports relation in that it allows a module to mention entities declared in the modules it extends. The difference is that the validity-ordering edge introduced by **extends** goes in the other direction from that introduced by **imports**. In other words, the validity ordering includes the converse of the extends relation on modules. For example, the declaration (1) allows the class declaration for $LATheory$ in module $ARITH$ to mention $TH.Theory$, and it "inserts" $ARITH$ as a predecessor of TH in the validity ordering.

In general, we allow a module to import and extend any number of other modules, but the resulting validity order must be acyclic.

Viewed from a different perspective, if a module A needs to mention the name of another module B, then A must be declared to either import or extend B. The declaration A **imports** B gives rise to $B \leftarrow A$, which means that t-validity of A implies t-validity of B; in contrast, A **extends** B gives rise to $A \leftarrow B$, which means that t-validity of B implies t-validity of A. (One could also consider an "oblivious import", which has no effect on the validity ordering.) A programmer must choose between **imports** and **extends** in such a way as to keep the validity ordering acyclic.

Note, by the way, that we do allow cyclic references between modules. For example, A and B can each refer to entities in the other if A imports B and B extends A, or vice versa (and cyclic references would also be possible with oblivious imports).

1.3 Module initialization

A module's invariant is first established by the module's initializer, a designated block of code that is invoked exactly once. Module initializers are invoked by the runtime system, so as to orchestrate the initialization of multiple modules. To enable modular reasoning, module loading and initialization adhere to the following policy:

0. Modules can be loaded and initialized only by a special program statement **fetch**. This statement is executed at program start before invocation of the $Main$ procedure and it can also be used by programs to load modules dynamically.

1. The **fetch** statement loads a specified module and all transitively imported and extended modules, except for those modules that have been loaded during an earlier **fetch**.

```
module ARITH extends TH imports STRING {
  STRING.String version ;
  invariant ARITH.version ≠ null ;
  initializer {
    STRING.String v := STRING.natToString(3) ;
    ARITH.version := v ;
  }
  ...
}
```

Figure 2: An example of a module initializer.

2. Once loading has completed, the **fetch** statement initializes all newly loaded modules by invoking the module initializers. The order of initialization follows the validity ordering: a module A is initialized after the initialization of all modules that precede A in the validity ordering.

Requirements 0 and 2 enforce an eager initialization of modules, which provides stronger guarantees for both the initializer and the clients of a module than the lazy initialization used, for instance, in Java [12].

Requirement 1 guarantees that the set of loaded modules is closed under the imports and extends relations. That is, loaded modules do not have any unresolved references to other modules.

Before invoking the initializer for a module A, **fetch** sets $A.si$ to $mutable$, which allows the initializer to assign to the module's variables. Upon return from the initializer, **fetch** sets $A.si$ to $tvalid$. Since **fetch** initializes modules according to a specified partial order (Requirement 2), the initializers of a module A can assume all predecessors of A to be t-valid. This is in particular necessary to allow A's initializer to access variables and procedures of A's imported modules. In return for being able to assume the precondition

$$A.si = mutable \ \wedge \ (\forall\, B \bullet \ B \leftarrow A \ \Rightarrow \ B.si = tvalid \ )$$

(where, here and throughout, quantifications over module names range over loaded modules), the initializer is responsible for making sure the following implicit assertion holds on exit:

$$\textbf{assert}\ ModuleInv(A)\ ;$$

For example, consider module $ARITH$ in Fig. 2. Because $STRING$ precedes $ARITH$, the second conjunct of the precondition implies $STRING.si = tvalid$; therefore, the initializer can meet the precondition of $natToString$. Because of the first conjunct of the precondition, the assignment to $ARITH.version$ is permitted. Note, by the way, that the $ARITH$ initializer cannot assume TH to be t-valid, since TH does not precede $ARITH$. Provided $natToString$ returns a non-null value, the implicit assertion at the end of the initializer body will hold.

```
Program    ::= Module*
ModuleDecl ::= module Id [ extends IdList ]
                 [ imports IdList ] {
               VarDecl*
               invariant Expr ;
               initializer { Stmt }
               ProcDecl*
               ClassDecl*
               }
VarDecl    ::= Type Id ;
ClassDecl  ::= class Id [ extends Id ]
               { ClassMemberDecl* }
```

Figure 3: The grammar of modules and classes. Square brackets indicate optional components, and * indicates zero or more occurrences.

2 Formalization

In this section, we formalize the notions introduced in the previous section. We do so by defining an object-oriented language with modules and then prescribing the operational semantics of this object-oriented language as a mapping to a conventional language.

2.0 Programming language

We consider a language where a program consists of a set of module declarations. A module declares a set of module variables, a module invariant, a set of procedures, an initializer, and a set of classes, see Fig. 3. The set of admissible invariants is described below. Every module B named in a module A must explicitly be declared as being imported or extended by A. We define the validity ordering to be the transitive closure of the union of the imports relation and the converse of the extends relation. The declarations in a program must ensure that the validity ordering is acyclic. Class members include fields (instance variables) and methods (dynamically-dispatched instance procedures), for which we use a Java-like syntax.

The statement language is given in Fig. 4. It includes assignments to local variables, module variables, and fields. A new local variable is introduced by giving its type and an initial assignment. A new instance of a class, an object, is allocated in a special assignment statement. The fields of the new object have zero-equivalent values. A custom initialization can be achieved by the invocation of a method. Our language thus separates allocation from initialization, which will be convenient for our presentation and which can model languages like Java and C# where the two are combined into instance *constructors*.

The assert statement causes program execution to abort if the given expression evaluates to **false**. References to procedures, module variables, and classes use fully qualified names; except, we sometimes omit the module name

Stmt	::=	[*Type*] $x := Expr$	local variable
	|	$A.g := Expr$;	module variable
	|	$x.f := Expr$;	field update
	|	$x :=$ **new** $A.T$;	object allocation
	|	**assert** *Expr*;	
	|	[$y :=$] $A.P(Expr^*)$;	procedure call
	|	[$y :=$] $x.M(Expr^*)$;	method call
	|	**expose** A { *Stmt* }	
	|	**fetch** *Expr*;	dynamic class load
	|	*Stmt Stmt*	sequential composition
	|	**if** (*Expr*) { *Stmt* } [**else** { *Stmt* }]	
	|	**while** (*Expr*) { *Stmt* }	

Figure 4: The grammar of statements. To suggest the types of expressions and identifiers, we write T for classes, x and y for variables, and A for modules.

in our examples when the module is clear. Calls to procedures and dynamically dispatched methods are standard, as are sequential, conditional, and iterative composition. The **expose** and **fetch** statements were described in Sections 1.0 and 1.3, respectively.

We omit the exact grammar for expressions, which include literals, local and module variables, access expressions (written $E.f$, to refer to the f field of the object denoted by E), and usual operators.

Program execution starts from a procedure named $Main$ in a given module, say $START$. Before $START.Main$ is called, module $START$ and its transitively imported and extended modules are loaded and initialized. This bootstrapping process is thus:

$$\textbf{fetch } \text{“}START\text{”}; START.Main() ;$$

2.1 Admissible invariants

An important consideration for any methodology that uses invariants is the set of variables and fields an invariant can refer to [17, 21]. The invariant of a module A may only refer to module variables declared in A and fields declared in classes in A.

Definition 0 (Admissible simple invariant) *The module invariant J of a module A is* admissible *if every access expression in J has one of the following forms:*

0. *a module variable $A.g$, or*
1. *$(A.g).h_1.\cdots.h_n.f$ where $n \geqslant 0$ and f is a field declared in a class in A.*

The variable g must not be the predefined variable si.

Note that each prefix $(A.g).h_1.\cdots.h_m$ $(0 \leq m \leq n)$ of an access expression $(A.g).h_1.\cdots.h_n.f$ is again an access expression. Therefore, for an access expressions of Case 1, g must be declared in A, and all fields h_i $(1 \leq i \leq n)$ must be declared in classes in A.

OpSem[$A.g := E$] $\equiv$
  assert $A.si = mutable$;
  $A.g := E$
OpSem[$x.f := E$] $\equiv$
  assert $x \neq$ **null** ;
  foreach access expression $(A.g).h_1.\cdots.h_n.f$
      mentioned in the module invariant of A {
    assert $x = (A.g).h_1.\cdots.h_n \Rightarrow A.si = mutable$;
  }
  $x.f := E$
OpSem[**expose** A { S }] $\equiv$
  assert $A.si \neq mutable$;
  let $Q = \{ C \mid A \leftarrow C \wedge C.si = tvalid \}$;
  [**foreach** $C \in Q$ { $C.si := valid$; }]
  $A.si := mutable$;
  OpSem[S] ;
  assert $ModuleInv(A)$;
  [**foreach** $C \in \{A\} \cup Q$ { $C.si :=$ **old**$(C.si)$; }]

Figure 5: The operational semantics of the object-oriented language is given as a mapping to a conventional language.

2.2 Operational semantics

We give the operational semantics of our language by a mapping into a conventional language. This operational-semantics language includes the statements of our object-oriented language in Fig. 4, with the following exceptions: the **assert** statement is the only statement that can cause the program to abort (all other statements are total); the **fetch** and **expose** statements are not present; and the language allows a statement S to be marked with *atomicity* brackets $[S]$, which have no semantic meaning but are used in the soundness proof to treat a number of statements as one.

At the very beginning of program execution, no modules have been loaded and no objects have been allocated. That is, the following condition holds:

$$loadedModules = \emptyset \wedge (\forall\, o \bullet \textbf{false})$$

where $loadedModules$ is an operational-semantics variable that keeps track of the set of loaded modules and, here and throughout, quantifications over objects range over non-null allocated objects.

The operational semantics of the language is given in Figs. 5 and 6. Trivial cases, where OpSem is the identity or simply distributes over all sub-statements, are omitted.

The operational semantics for an assignment to a module variable g declared in a module A reflects the decision in the methodology to abort program execution if A is not mutable. Note that we require A to be mutable even if the update actually would maintain the module invariant of A; this centralizes the checking of module invariants to the end of **expose** blocks (*cf.* [0]).

Updating a field f declared in a class in a module A requires the receiver of the update, x, to be non-null. It

also requires A to be mutable whenever A's invariant depends on $x.f$, that is, when $(A.g).h_1.\cdots.h_n.f$ is some subexpression of the module invariant of A and the reference x equals the value of $(A.g).h_1.\cdots.h_n$. For example, suppose the invariant of a module A mentions $(A.g).f$, where f is a field declared in a class T in A. Then, in the program fragment

$$x := \textbf{new}\ A.T\ ;\ y := A.g\ ;$$
$$x.f := 12\ ;\ y.f := 14\ ;$$

the operational-semantics asserts for the first field update always hold, whereas the asserts for the second field update hold iff A is mutable.

The non-reentrancy of the **expose** statement requires that the module A to be exposed is not already exposed. Before executing the sub-statement S, the **expose** statement makes A mutable and changes all $tvalid$ successors of A to be just $valid$. After executing S, the module invariant of A is checked and the initial values of all changed si variables are restored.

The semantics of the **fetch** statement is presented in Fig. 6. For simplicity, we forbid recursive **fetch** operations (not shown in Fig. 6). That is, **fetch** must not be invoked during module initialization.

The first half of the **fetch** statement performs the loading as an atomic operation. It requires that all predecessors of a newly loaded module are either uninitialized (which is the case if they have been loaded during the same **fetch** operation) or t-valid. The responsibility for establishing this condition lies with the callers of **fetch**, as is indicated by the operational-semantics assert. The si variable of all newly loaded modules is set to $uninitialized$.

Because modules can be extended, a **fetch** may load new predecessors for previously loaded and initialized modules. If this were to happen, program invariant J0 would be at stake. Therefore, after loading the new modules, **fetch** changes any such t-valid successor modules (*i.e.,* those in Q_1) to be just valid.

The second half of the **fetch** statement initializes the newly loaded modules and gradually restores the t-validity of the modules in Q_1. A module C is initialized by setting $C.si$ to $mutable$, calling C's module initializer, and finally setting $C.si$ to $tvalid$.

For example, if $ARITH$ and TH of Fig. 2 are loaded by the same **fetch**, then TH's initializer is invoked after $ARITH$'s initializer. If TH is loaded by an earlier **fetch**, then either TH is not t-valid at the later **fetch** or the later **fetch** changes $TH.si$ from $tvalid$ to just $valid$ until after $ARITH$ has been initialized.

A loop invariant of the initialization loop is that each predecessor B of a module C in Q is either itself in Q or is t-valid. This loop invariant holds on entry to the initialization loop, because a loop invariant of the first loop is that each predecessor of a t-valid or uninitialized modules is either t-valid or uninitialized, and because between the two loops a module's si field is changed only for modules in Q. The loop invariant is preserved by the initialization loop, because modules removed from Q become t-valid. Since the validity ordering is a partial order, this loop invariant guarantees that there is a module C that can be chosen by the **choose** operation.

```
OpSem[ fetch "A" ] ≡
  toBeLoaded := {A} ;
  [ while toBeLoaded ≠ ∅ {
      choose C ∈ toBeLoaded ;
      toBeLoaded := toBeLoaded \ {C} ;
      if C ∉ loadedModules {
        load C ;
        assert ( ∀ B • B ← C ⇒
          B.si = uninitialized ∨ B.si = tvalid ) ;
        let R be the set of modules imported or extended by C ;
        toBeLoaded := toBeLoaded ∪ R ;
        set all module variables of C to zero-equivalent values ;
        C.si := uninitialized ;
        loadedModules := loadedModules ∪ {C} ;
      }
    }
    let Q0 = { C | C.si = uninitialized } ;
    let Q1 = { C | C.si = tvalid ∧
                   ( ∃ D • D ∈ Q0 ∧ C ← D ) } ;
    foreach C ∈ Q1 { C.si := valid ; } ]
  var Q := Q0 ∪ Q1 ;
  while Q ≠ ∅ {
    choose C | C ∈ Q ∧
      ( ∀ B • B ← C ⇒ B.si = tvalid ) ;
    Q := Q \ {C} ;
    if C.si = uninitialized {
      C.si := mutable ;
      C.init( ) ;
    }
    C.si := tvalid ;
  }
```

Figure 6: Pseudo code for the **fetch** statement.

Since the **fetch** statement is the only operation that sets an si field to $uninitialized$ and since all uninitialized modules are initialized before **fetch** terminates, the statement guarantees that all modules are initialized in all program states in which there is no **fetch** in progress. That is, treating **fetch** as atomic operation, the following is a program invariant: $(\forall\, C \bullet\ C.si \neq uninitialized\ )$. The proof of this program invariant is straightforward and, therefore, omitted.

3 Soundness

For our methodology, soundness means that the si variable of each module A correctly reflects whether the module invariants of A and A's predecessors can be expected to hold.

In this section, we formalize and prove this property for well-formed programs. A program **P** is well-formed if **P** is syntactically correct, type correct, **P**'s invariants are admissible (see Def. 0), and the validity ordering induced by the declarations in **P** is a partial order.

Theorem 0 (Soundness) *Properties J0 and J1 (see Section 1.1) are program invariants, that is, they hold in every reachable execution state of a well-formed program.*

3.0 Auxiliary lemma

The soundness proof makes use of the following auxiliary lemma: a statement S has no net effect on the si variable of modules that were loaded before the execution of S, and all modules that are loaded during its execution are t-valid after the execution of S.

Lemma 1 (Auxiliary lemma) *Each statement S guarantees the following postconditions*

(i) $(\forall\, C \bullet C \in \mathbf{old}(loadedModules) \Rightarrow C.si{=}\mathbf{old}(C.si)\ )$

(ii) $(\forall\, C \bullet C \notin \mathbf{old}(loadedModules) \Rightarrow C.si{=}tvalid\ )$

where $\mathbf{old}(E)$ denotes the value of expression E in the state where execution of S begins.

Proof. The lemma is proved by rule induction. All cases except **expose** and **fetch** are trivial since they do not modify the si variable of loaded modules.

PART (I). The **expose** statement (Fig. 5) modifies the si variable of modules, then invokes the sub-statement (which, by the induction hypothesis does not have a net effect on the si variable of loaded modules), and finally restores the initial values of the si variables it had changed.

The **fetch** statement (Fig. 6) modifies the si variable of modules in $Q_0 \cup Q_1$, but the modules in Q_0 were not previously loaded and the initial value of the si variable of each module in Q_1 is restored after some number of calls to module initializers (each one of which has no net effect on the si variables of loaded modules, by the induction hypothesis).

PART (II). By the induction hypothesis, we know that the sub-statement of an **expose** statement satisfies the postcondition. In the subsequent instructions, si variables are not changed from $tvalid$ to other values. Therefore, the property is preserved.

The **fetch** statement loads a number of modules, each one of which it will eventually assign to be t-valid. The calls to module initializers don't interfere with the si variables of loaded modules (by Part (i)) and only produce newly loaded modules that are t-valid (by induction hypothesis). Therefore, any module loaded during the **fetch** will indeed by t-valid on exit. □

3.1 Proof of program invariant J0

The program invariant J0 holds in the initial state of an execution of a program **P** because no modules are loaded. We show by rule induction that each statement of **P** preserves J0. For the proof, only the statements that modify the state or load new modules are interesting: module variable update and field update for the base case, as well as **expose** and **fetch** for the induction step. We omit all other cases for brevity.

Module-variable and field update. Updates neither change the value of si variables nor load new modules.

expose statement. Consider the statement **expose** A { S }. We assume that J0 holds in the state before execution of this statement and prove that it is preserved by each of the instructions in the operational semantics of **expose** (Fig. 5).

0. **assert** and **let** instructions do not change the state.
1. The first **foreach** statement sets all $tvalid$ successors of A to $valid$, which preserves J0.
2. Setting $A.si$ to $mutable$ preserves J0 because the si variables of A's successors are different from $tvalid$.
3. By the induction hypothesis, S preserves J0.
4. To show that the last **foreach** statement preserves the invariant, we consider two modules B and C, where $B \leftarrow C$ and prove $C.si = tvalid \Rightarrow B.si = tvalid$. We continue by case distinction:

   (a) If both B and C have been loaded before execution of the **expose** statement, the property holds after the **foreach**, which is the end of the **expose** statment, because J0 holds before the execution of the **expose** and the execution of an **expose** statement does not have a net effect on si variables of loaded modules (Lemma 1 (i)).

   (b) If B is loaded during the execution of the **expose**, we apply Lemma 1 (ii) to show that B is $tvalid$ after execution of S. Since $B \notin \{A\} \cup Q$, $B.si$ is unchanged by the **foreach**, and thus $C.si = tvalid \Rightarrow B.si = tvalid$ is preserved.

   (c) If B has been loaded before and C during the execution of the **expose**, we apply Lemma 1 (ii) to show that C is $tvalid$ after execution of S. Since J0 holds in the poststate of S, we get that all predecessors of C, in particular B, are $tvalid$. Since A and all modules in Q are not $tvalid$ in this state (Lemma 1 (i)), we conclude that the **foreach** instruction does not change $B.si$ and $C.si$.

fetch statement. Consider the statement **fetch** "A". We assume that J0 holds in the state before execution of this statement and prove that it is preserved by each of the atomic instructions in the operational semantics of **fetch** (Fig. 6).

0. The atomic loading statement (that is, the group of statements within the atomicity brackets) sets the si variable of newly loaded modules to $uninitialized$ and then changes any t-valid successor of an uninitialized module to just valid (precisely for the purpose of maintaining J0).

1. The assignment that changes $C.si$ from $uninitialized$ to $mutable$ preserves J0.

2. By the induction hypothesis, execution of C's initializer preserves J0.

3. The assignment $C.si := tvalid;$ preserves J0 because C has been chosen such that all predecessors of C are $tvalid$. This property is not changed by executing C's initializer (Lemma 1 (i) and (ii)). □

3.2 Proof of program invariant J1

The proof for J1 is analogous to J0. We present the same cases of the induction base and step.

Module-variable update. Consider a module-variable update $A.g := E$. By the definition of admissible invariants (Def. 0), only A's module invariant can be violated by this update. The assertion in the semantics for module-variable updates guarantees that A is $mutable$, and thus J1 is preserved.

Field update. Consider a field update $x.f := E$ where f is declared in a class in a module A. Again, only A's module invariant can be violated by this update, namely if it contains an access expression $(A.g).h_1.\cdots.h_n.f$, where $x = (A.g).h_1.\cdots.h_n$. In this case, the assertion in the semantics for field updates guarantees that A is $mutable$, and thus J1 is preserved.

expose statement. Consider the statement **expose** A { S }. We assume that J1 holds in the state before execution of this statement and prove that it is preserved by each of the instructions in the operational semantics of **expose** (Fig. 5).

0. The instructions before the execution of S preserve J1 because they only change si for the modules in Q from $tvalid$ to $valid$, and $A.si$ to $mutable$. Since module invariants must not mention si (Def. 0), module invariants are not affected by these modifications. Consequently, J1 is preserved.

1. By the induction hypothesis, S preserves J1.

2. The last **foreach** statement sets $A.si$ to its initial value after asserting $ModuleInv(A)$. By Lemma 1 (i), S does not change si for modules in Q; thus, for modules in Q, the **foreach** statement changes si from $valid$ to $tvalid$, which does not affect J1. Therefore, this statement preserves J1.

fetch statement. Consider the statement **fetch** "A". We assume that J1 holds in the state before execution of this statement and prove that it is preserved by each of the atomic instructions in the operational semantics of **fetch** (Fig. 6).

0. The atomic loading statement sets some si variables to $uninitialized$ and changes some si variables from $tvalid$ to $valid$. These operations preserve J1.

1. The assignment $C.si := mutable;$ trivially preserves J1.

2. By the induction hypothesis, execution of C's initializer preserves J1.

3. The initializer for C ends with the implicit assertion $ModuleInv(C)$. Therefore, the assignment $C.si := tvalid;$ preserves J1 if $C \in Q_0$. If $C \in Q_1$, the change of $C.si$ from $valid$ (Lemma 1 (i)) to $tvalid$ also preserves J1. □

4 Ownership-based invariants

So far, the invariant of a module A may depend only on module variables and fields declared in A. Therefore, assignment to a field or module variable can only violate the module invariant of the enclosing module. This restriction allows us to guard such an assignment by the assertion that the module is mutable.

However, preventing module invariants from depending on variables and fields declared in different modules is too restrictive for many interesting programs. For instance, a module may want to use a global cache data structure, implemented by an imported collection class, and impose certain requirements on the elements stored in the collection. Fig. 7 shows such a program. A simple $Cell$ is used to represent the cache, and $CLIENT$'s module invariant states that only natural numbers are stored in the cache.

The reason the methodology introduced so far cannot handle such programs is illustrated by the field update in method set: this update potentially violates $CLIENT$'s module invariant. However, since this invariant is contained in a different module, it is not possible to determine modularly that the update has to be guarded by an assertion that $CLIENT$ is mutable.

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

4.0 Ownership

Ownership organizes objects into a hierarchy of *contexts*, where the objects in each context have a common owner

```
module CELL {
  class Cell {
    int c ;
    method set(int p) { this.c := p ; }
  }
}
module CLIENT imports CELL {
  Cell cache ;
  invariant CLIENT.cache ≠ null ∧
    (CLIENT.cache).c ⩾ 0 ;
  initializer { CLIENT.cache := new Cell ; }
  ...
}
```

Figure 7: A simplified implementation of a module with a global cache. Without ownership, the invariant of module $CLIENT$ is not admissible, because it refers to a field c that is declared in a different module.

(see, *e.g.*, [2, 4, 21]). In our methodology, an owner is either a module or a pair consisting of an object reference and a class name. Other than ownership, we do not restrict object references; an object may have non-owning references to other objects.

Following our work on object invariants [16], we encode ownership by a special field $owner$ for every object. The value of $owner$ is either a module or a pair $[obj, typ]$. It is set when an object is created. We extend the object allocation statement so that an owner can be indicated: $x := \mathbf{new}\ \langle A \rangle\, T$ creates a new object of class T owned by module A. Analogously, the object created by $x := \mathbf{new}\ \langle [o, U] \rangle\, T$ is owned by the pair of object o and type U. In this paper, we assume $owner$ to be immutable after object creation. We described how to handle a mutable $owner$ field (ownership transfer) in a previous paper [16].

We say that an object X is *(transitively) owned* by a module A if $X.owner = A$ or if $X.owner = [Y, T]$ and Y is (transitively) owned by A. In this paper, we are only interested in whether an object is owned by a module or another object. The type component of an owner is only used to handle inheritance in our methodology for object invariants and can, therefore, safely be ignored here. Consequently, we say that an object X is owned by object Y if $X.owner = [Y, T]$ for some type T.

The $owner$ field can be mentioned in module and object invariants. To simplify the notation and to check the definition of admissible invariants syntactically, any module variable and field can be declared with the modifier $\mathbf{rep}$. This modifier gives rise to an implicit invariant. For the declaration $\mathbf{rep}\ S\ g;$ of a module variable in a module A, A contains the implicit module invariant

$$A.g \neq \mathbf{null}\ \Rightarrow\ (A.g).owner = A\ ;$$

Analogously, a field f declared $\mathbf{rep}$ in a class T gives rise to the implicit object invariant

$$\mathbf{this}.f \neq \mathbf{null}\ \Rightarrow\ \mathbf{this}.f.owner = [\mathbf{this}, T]$$

4.1 Admissible invariants

Ownership allows us to support more module invariants. In addition to the module variables and fields permitted by Def. 0, the invariant of a module A may depend on fields of objects that are (transitively) owned by A. This leads to the following refined definition of admissible invariants.

Definition 1 (Admissible ownership-based invariant) *The module invariant J of a module A is* admissible *if every access expression in J has one of the following forms:*

0. *a module variable $A.g$, or*
1. $(A.g).h_1.\cdots.h_n.f$ *where $n \geqslant 0$ and f is a field declared in a class in A, or*
2. $(A.g).h_1.\cdots.h_n.f$ *where $n \geqslant 0$ and $A.g$ as well as each h_i is declared* $\mathbf{rep}$*, or*
3. $x.f$ *where x is bound by a universal quantification of the form* $(\forall\ T\ x \bullet\ x.owner = A\ \Rightarrow\ \ldots\ x.f\ \ldots\ )$.

The variable g must not be the predefined variable si.

The Cases 0 and 1 are identical to simple invariants (Def. 0). Access expressions in Case 2 allow module invariants to depend on fields of owned objects. The fact that these objects are owned by A can be derived from the fact that they are reachable via a chain of $\mathbf{rep}$ references. Case 3 allows module invariants to quantify over all objects directly owned by the module, even if the objects are not reachable from a module variable.

By declaring $CLIENT.cache$ in Fig. 7 as $\mathbf{rep}$, $CLIENT$'s invariant would be admissible according to this refined definition. The access expressions $CLIENT.cache$ and $(CLIENT.cache).c$ would then meet the requirements of Cases 0 and 2, respectively.

4.2 Mutability of owned objects

Allowing module invariants to depend on fields of (transitively) owned objects introduces a connection between the methodologies for module invariants and object invariants, for instance because we make use of the implicit object invariants introduced by $\mathbf{rep}$ modifiers in field declarations. Our methodology for object invariants is based on ownership, some special fields for objects, and statements to expose and unexpose objects. In this subsection, we summarize those aspects of the methodology that are necessary to understand our treatment of module invariants. For a detailed presentation, see our previous paper [16].

Mutability of objects is handled analogously to modules. Each object has a special field $committed$. Only fields of objects that are *not* committed can be updated. That is, a field update $x.f := E$ is guarded by the additional assertion

$$\textbf{assert } \neg x.committed \ ;$$

Consider a module A which (transitively) owns x. A's invariant may depend on $x.f$ even if f is declared in another module. Consequently, an update of $x.f$ may violate A's invariant. Our methodology handles this situation by the following rule: a module is mutable whenever one of the objects it owns is not committed. That is, from $\neg x.committed$, we can conclude that A is mutable. Therefore, A's invariant is allowed to be violated by the update of $x.f$.

The $mutable$ and $committed$ fields can only be manipulated according to a strict protocol. This protocol guarantees that an object can be uncommitted only if all of its owner objects are uncommitted and, consequently, the owning module, if any, is mutable. To enforce this protocol, $committed$ can only be manipulated in a restricted way by the statements for object creation, **expose**, **fetch**, and two special statements to expose objects. We describe the effects of these statements on $committed$ in the following.

Object creation. The $committed$ field is set to **false** when an object is created. The creation statements $x := \textbf{new } \langle A \rangle\, T$ and $x := \textbf{new } \langle [o, U] \rangle\, T$ are guarded by the assertions $A.si = mutable$ and $\neg o.committed$, respectively, to ensure that the direct owner is mutable (if it is a module) or uncommitted (if it is an object).

expose statement. The statement **expose** $A \{ S \}$ modifies the $committed$ field of objects that are directly owned by A. It sets $committed$ to **false** when $A.si$ is set to $mutable$. Analogously, $committed$ is set to **true** when the original value of $A.si$ is restored at the end of the **expose** statement. To preserve the property that an object Y can be committed only if all objects transitively owned by $[Y, T]$ are committed, this operation requires an additional assertion that objects that are transitively, but not directly owned by A are committed. Fig. 8 shows the refined semantics of the **expose** statement.

fetch statement. After initialization of a newly loaded module C, the **fetch** statement sets the $committed$ field of objects directly owned by C to **true**. That is, in the semantics of **fetch** (Fig. 6), the assignment $C.si := tvalid$; is replaced by the following atomic operation:

$$\left[\begin{array}{l} C.si := tvalid \ ; \\ \textbf{foreach } X \mid X.owner = C \ \{ \ X.committed := \textbf{true} \ ; \ \} \end{array}\right]$$

Analogously to the new assertion in the **expose** statement, this operation leads to an additional implicit assertion at the end of C's module initializer. Objects that are transitively, but not directly, owned by C are committed:

$$\begin{array}{l} \textbf{assert } (\forall X, Y, T \bullet \ X.owner = [Y, T] \wedge \\ \quad Y.owner = C \ \Rightarrow \ X.committed \ ) \ ; \end{array}$$

$$\begin{array}{l} \textsc{OpSem}[\ \textbf{expose } A \ \{ \ S \ \} \ ] \equiv \\ \quad \textbf{assert } A.si \neq mutable \ ; \\ \quad \textbf{let } Q = \{ \ C \mid A \leftarrow C \wedge C.si = tvalid \ \} \ ; \\ \quad [\,\textbf{foreach } C \in Q \ \{ \ C.si := valid \ ; \ \}\,] \\ \quad \left[\begin{array}{l} A.si := mutable \ ; \\ \textbf{foreach } X \mid X.owner = A \ \{ \ X.committed := \textbf{false} \ ; \ \} \end{array}\right] \\ \quad \textsc{OpSem}[\ S \ ] \ ; \\ \quad \textbf{assert } ModuleInv(A) \ ; \\ \quad \textbf{assert } (\forall X, Y, T \bullet \ X.owner = [Y, T] \wedge \\ \qquad Y.owner = A \ \Rightarrow \ X.committed \ ) \ ; \\ \quad \left[\begin{array}{l} \textbf{foreach } C \in \{A\} \cup Q \ \{ \ C.si := \textbf{old}(C.si) \ ; \ \} \\ \textbf{foreach } X \mid X.owner = A \ \{ \ X.committed := \textbf{true} \ ; \ \} \end{array}\right] \end{array}$$

Figure 8: Pseudo code for the ownership-aware **expose** statement. The statement modifies the $committed$ field of objects directly owned by A.

unpack and pack statements. For objects that are owned by other objetcs, our methodology for object invariants provides two additional statements, **unpack** and **pack**, analogous in functionality to **expose**. When applied to an uncommitted object X, **unpack** sets the $committed$ field of all objects owned by $[X, T]$ to **false**, whereas **pack** sets it to **true**. Neither statement modifies fields other than $committed$. For a detailed description including a formal semantics of these, see [0, 16].

We formalize two program invariants about the relation between the si and $committed$ fields in Section 5.

4.3 Example

The implementation of a web server in Fig. 9 illustrates the expressiveness of ownership-based module invariants. Although arrays are not covered by the formalization presented in this paper, we use arrays in this example to show that our methodology can handle them. Array elements behave like public fields declared in class $Object$. That is, every procedure or method that has a reference to an array object can modify its elements. Reading and updating array elements is handled analogously to field read and update.

Module $SERVER$ maintains a global cache of web pages, which is represented by the array $cache$. As indicated by the **rep** keyword, this array object is owned by module $SERVER$.

The module invariant requires $cache$ to be different from **null**. Moreover, different slots of the array have to store different references or **null**. This invariant is not admissible according to Def. 0, because it refers to fields such as $length$ and the array elements, which are not declared in classes in $SERVER$. However, the invariant is an admissible ownership-based invariant, since these fields belong to objects owned by $SERVER$. $SERVER$'s module invariant as well as the implicit invariant about ownership is established by the module initializer.

```
module WEBPAGE {
  class WebPage { rep String theURL ; ... }
}

module SERVER imports WEBPAGE, STRING, RANDOM {
  rep Webpage[ ] cache ;
  invariant SERVER.cache ≠ null ∧
    (∀ i, j • 0 ≤ i < j < (SERVER.cache).length ∧
      SERVER.cache[i] ≠ null ⇒
        SERVER.cache[i] ≠ SERVER.cache[j] ) ;
  initializer {
    SERVER.cache := new ⟨SERVER⟩ Webpage[10] ;
  }
  class WebServer {
    Webpage Request(String url)
      requires SERVER.si = tvalid ∧ url ≠ null ;
    {
      int i := 0 ;
      result := null ;
      while (i < (SERVER.cache).length ∧ result = null) {
        if (url.Equals(SERVER.cache[i].theURL)
          { result := SERVER.cache[i] ; }
        i := i + 1 ;
      }
      if (result = null) {
        result := ...   //retrieve webpage
        i := RANDOM.Generate( ) ;
        expose SERVER { SERVER.cache[i] := result ; }
      }
    }
  }
}
```

Figure 9: An implementation of a simple web server. The module invariant of *SERVER* refers to the state of the *cache* array owned by *SERVER*. The module has to be exposed before updating the cache.

Method *Request* requires *SERVER* to be *tvalid*. From this precondition, we can conclude that the imported modules *STRING* and *RANDOM* satisfy their module invariants (program invariants J0 amd J1). This property is necessary to meet the preconditions of the methods and procedures called in *Request*, such as *String.Equals* and *RANDOM.Generate*. (The specifications of these methods are not shown in the code.)

The update $SERVER.cache[i] :=$ **result**; in method *Request* illustrates how ownership-based invariants are handled. To satisfy the assertion $\neg(SERVER.cache).committed$ of the update, *SERVER* has to be exposed. The **expose** statement sets $SERVER.si$ to *mutable* and allows the array update to temporarily violate *SERVER*'s module invariant as long as the invariant is reestablished before the **expose** block ends. In our example, the invariant is not violated by the update because a reference is stored in *cache* only if the preceding loop does not find the reference in the array.

5 Soundness in the presence of ownership

In addition to the program invariants J0 and J1, the extended methodology guarantees that an object X can be committed only if all objects owned by $[X, T]$ are committed, and that an object directly owned by a module A is committed if and only if A is not mutable.

In the extended methodology, a program **P** is well-formed if **P** is syntactically correct, type correct, **P**'s invariants are admissible according to Def. 1, and the validity ordering induced by the declarations in **P** is a partial order.

Theorem 2 (Soundness in presence of ownership) *In each reachable execution state of a well-formed program, the program invariants J0 and J1 (see Lemma 0) as well as the following program invariants hold:*

$$\textit{J2:}\ (\forall\, x, o, T \bullet\ x.owner = [o, T] \wedge o.committed \Rightarrow x.committed\ )$$

$$\textit{J3:}\ (\forall\, x, A \bullet\ x.owner = A \Rightarrow (A.si \neq mutable\ \equiv\ x.committed)\ )$$

where x and o range over non-null allocated objects.

In this section, we show that the program invariants J0 and J1 are still valid in the presence of ownership-based invariants. We omit the proofs for J2 and J3 for brevity. They are analogous to the proofs for J0 and J1. The interesting cases are those statements that manipulate the *si* variable or the *committed* field: object creation, **expose**, **fetch**, **unpack**, and **pack**.

5.0 Proof of program invariant J0

The program invariant J0 neither refers to *committed* nor depends on the definition of admissible invariants. Therefore, its proof is not affected by the ownership extensions, since all modifications in the semantics of statements are either assertions or manipulate *committed*. □

5.1 Proof of program invariant J1

For program invariant J1, we have to adapt the proof (see Section 3.2) to cover the additional cases in the definition of admissible invariants (Def. 1).

The proof for Case 2 of Def. 1 relies on properties about object invariants, in particular the implicit object invariants about *owner* stemming from **rep** declarations. These properties are guaranteed by our methodology for object invariants [16]. However, for self-containedness, we do not want to use these properties here. Therefore, we present the proof of a restricted part of the theorem: for Case 2 of Def. 1, we assume that $n = 0$. That is, the invariant of a module A may

depend on fields of objects *directly* referenced by $A.g$, but not of transitively referenced objects. Consequently, these objects are directly owned by A. We have proved that J1 holds also in the general case.

The only proof case that is affected by the refined definition of admissible invariants is the case for field updates. Consider a field update $x.f := E$. We have to show that if the module invariant of a module A depends on the value of $x.f$, then A is neither $valid$ nor $tvalid$. Following the cases in Def. 1:

0. The access expression does not depend on any field.
1. f is declared in a class in module A. This case is covered by the proof in Section 3.2.
2. A's module invariant contains an access expression $(A.g).f$, where $A.g$ is declared **rep** and $A.g = x$. We show by contradiction that A is neither $valid$ nor $tvalid$.

   Assume that A is $valid$ or $tvalid$. By the induction hypothesis, we have that J1 holds before the update. Therefore, A's module invariants hold. From the implicit invariant for the **rep** variable $A.g$ and $A.g = x$, we know $x.owner = A$. From the assertion guarding the update, we get $\neg x.committed$, which, by program invariant J3, implies $A.si = mutable$. However, this is a contradiction to the assumption that A is $valid$ or $tvalid$.
3. A's module invariant contains an access expression $o.f$ where o is bound by a universal quantification, $o.owner = A$, and $o = x$. Analogously to Case 2, we have $x.owner = A$, $\neg x.committed$, and, therefore, $A.si = mutable$. □

6 Discussion

6.0 More invariants

An interesting extension of the methodology we've presented expresses relations between a module variable $A.g$ and fields of objects that are not necessarily reachable from A's module variables. For instance, one might specify that for all T objects x, $x.theValue$ is bounded by a global maximum value:

$$x.theValue \leqslant A.theMaximum$$

Basically, such properties can be specified as a module invariant in A or as an object invariant of x. The methodology we present in this paper can express such properties only for the objects x owned by A (Case 3 of Def. 1). If such an invariant has to be expressed for *all* T objects x, independently of x's owner, then it has to be specified as an object invariant. We have extended our methodology for object invariants [16] to allow object invariants to mention module variables. However, these extensions are beyond the scope of this paper.

6.1 Static verification

Soundness of our methodology is essentially guaranteed by the assertions in the semantics of statements. For instance, if the module invariant of module A does not hold at the end of the body of an **expose** A statement, the program aborts. However, it is possible to statically verify that a program does not abort due to a violated assertion. To do that, each assertion is turned into a proof obligation.

Such proof obligations are introduced for the assertions in module-variable and field updates, **expose** statements, and object creation statements as well as for the implicit assertions in module initializers. Technically, these statements are replaced by their pseudo code including assertions. One can then use an appropriate program logic to show that the assertions hold (*cf.* [24, 10]).

The argument to **fetch** is an expression denoting a string, whose characters name the module to be fetched. At compile time, there is no information about the set of modules imported or extended by this module to be fetched. Therefore, it is in general not possible to prove the assertion in the **fetch** statement (Fig. 6) statically. For this reason, the assertion in **fetch** has to be checked at run time. Note, however, that this assertion can never abort for the initial **fetch** that is performed at the start of a program execution.

A crucial element of modular static verification is reasoning about which variables a call may modify. Analogous to the interpretation of *modifies specifications* for fields [0], we envision using a policy that allows any procedure or method to have a net effect on the module variables of modules that are valid or t-valid, without explicitly having to mention such modifications in the procedure or method's modifies declaration.

6.2 Hidden imports

In this paper, the imports and extends relations for modules have always been visible to the clients of a module. This has the advantage that the acyclicity requirement for the validity ordering can be checked at compile time. However, for the sake of information hiding, it may be desirable to hide portions of the imports or extends relations. In such a setting, the acyclicity test may have to be done later (for example, at link time or load time), and a failing acyclicity test may then come as a surprise to the programmer.

7 Related work

7.0 Module and object invariants

Our treatment of module invariants is based on our methodology for reasoning about object invariants [0, 16]. Analogously to the work presented in this paper, this methodology uses a special field to represent explicitly when an object invariant is allowed to be violated. This field can be manipulated only by the special statements **unpack** and **pack**, which are analogous to the **expose** statement we use here, and field updates require an object to be unpacked (exposed) before it can be modified.

There are significant differences between module invariants and object invariants. Our methodology for object invariants uses hierarchical ownership as an abstraction mechanism, where a client object exclusively owns the objects of its internal representation. Since modules are not owned by one owner, but shared by all clients in a program, this paper uses a different abstraction mechanism, namely an acyclic ordering on the modules. However, ownership is still used to allow module invariants to depend on object fields. The ownership encoding is identical in both methodologies. However, our earlier paper enables dynamic ownership transfer, whereas here we use a fixed owner relation to simplify the formalization. An extension to dynamically changing owners is straightforward.

Barnett and Naumann [1] extend our methodology for so-called visibility-based object invariants [16]. Such an invariant of an object X may depend on fields of objects that are not owned by X. Visibility-based invariants are expressive, but make it difficult to determine all objects that have to be unpacked before a field update. However, due to the static nature of modules, the modules that are potentially affected by a field update can easily be determined.

Pierik *et al.* [23] add so-called *creational guards* to Barnett and Naumann's work to allow invariants to quantify over all objects of a class, for instance, to specify that a singleton object is the only instance of a class. Pierik *et al.* do not address either the abstraction problem for class invariants or the initialization-order problem for classes.

Müller's thesis [21] uses a visible state semantics for object invariants, which requires invariants of relevant objects to hold in pre- and postconditions of all exported methods, whereas our methodology allows invariants to be violated as long as such violations are made explicit by the si field. Müller's thesis supports invariants over so-called abstract fields in a sound way, which we consider future work for the methodology presented here.

Leino and Nelson [15, 17] developed a sound modular treatment of object invariants over abstract fields. However, their work is not based on the notion of ownership, which makes the soundness proof difficult. The Extended Static Checker for Modula-3 [8] uses the technique of Leino and Nelson to reason about validity of object structures by defining a boolean abstract field $valid$ to represent validity. Usage of this field in specifications is similar to our si field. Leino and Nelson treat some aspects of module invariants and module initialization order, but neither Müller's nor Leino and Nelson's work fully supports module invariants.

The Extended Static Checker for Java [10] and ESC/Java2 [6] use heuristics to determine which object invariants to check for method invocations. Described in detail in the ESC/Java User's Manual [18], these heuristics are a compromise between flexibility and likelihood of errors and do not guarantee soundness.

JML [13, 14] provides static invariants to express properties of static fields and objects referenced from static fields. Static invariants correspond to our module invariants. Analogously to our methodology, the static invariant of a class C has to be established by C's static initializer. However, 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.

7.1 Global data

Our realization of global data as module variables is similar to static fields in Java and C#. Directly supporting static class invariants would require programmers explicitly to specify an acyclic validity ordering among classes as a means of abstraction. To avoid this overhead, we use modules and their imports relation, with module variables and procedures instead of static fields and static methods. This solution does not restrict the generality of our methodology. Every Java and C# program can easily be mapped to our notation.

In Eiffel [20], global data is implemented by **once** methods. The body of these methods is executed only when the method is called for the first time. For further calls, the cached result of the first execution is returned. That is, global data is essentially realized by objects that are shared by all objects that have to access the global data. However, since a shared object must be accessed from several other objects, it cannot, in general, be owned exclusively by another object. Therefore, it is unclear how to express invariants such as the module invariant in module $SERVER$ (Fig. 9) for such global objects.

7.2 Loading and initialization

Dynamic class loading and linking has been studied intensely in the context of type safety [5, 9, 19, 25]. We build on this work by assuming that the load operation applies a type safe load and link mechanism. However, we abstract from the details of loading, resolution, and bytecode verification in the semantics of **fetch**.

Java uses lazy class initialization. That is, classes are

initialized before they are first used. Although lazy intialization can be formalized in operational semantics and Hoare logics [22], it entails several problems. Kozen and Stillerman [12] illustrate that modular reasoning is extremely difficult in the presence of lazy initialization because the initialization order of classes and, thus, the initial values of static fields, depend on their clients. The focus of Kozen and Stillerman's paper is on a static analysis for Java bytecode to determine class initialization dependencies. Such an analysis is not needed in our approach since the initialization order is given explicitly by an acyclic imports relation. Börger and Schulte [3] report on several problems of Java's lazy initialization mechanism related to portability, concurrency, and compiler optimizations. These problems are avoided by the eager initialization used in our paper.

8 Conclusions

We presented the first modular verification methodology for global module invariants in object-oriented programs. These invariants can express properties of global data, that is, values stored in module variables. In addition, ownership allows module invariants to refer to fields of objects or object structures such as global caches. The methodology is proved to be sound.

The methodology for module invariants complements our previous treatment of object invariants [16]. Both methodologies are based on the same formal model, for instance on the same encoding of ownership. Therefore, they can easily be combined.

As future work, we will extend our methodology to allow method calls in invariants. Moreover, we plan to implement our methodology as part of the .NET program checker Boogie at Microsoft Research and use this implementation for non-trivial case studies.

References

[0] Mike Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6), 2004. `www.jot.fm`.

[1] Mike Barnett and David Naumann. Friends need a bit more: Maintaining invariants over shared state. In Dexter Kozen, editor, *Mathematics of Program Construction*, Lecture Notes in Computer Science, pages 54–84. Springer, July 2004.

[2] Boris Bokowski and Jan Vitek. Confined types. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '99)*, volume 34, number 10 in *SIGPLAN Notices*, pages 82–96. ACM, October 1999.

[3] Egon Börger and Wolfram Schulte. Initialization problems for Java. *Software—Concepts & Tools*, 20(4), 1999.

[4] Chandrasekhar Boyapati, Robert Lee, and Martin Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002*, volume 37, number 11 in *SIGPLAN Notices*, pages 211–230. ACM, November 2002.

[5] Luca Cardelli. Program fragments, linking, and modularization. In *Principles of programming languages (POPL)*, pages 266–277. ACM Press, 1997.

[6] David Cok and Joe Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an Internet voting tally system. In *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, 2004. (submitted).

[7] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. Technical Report HPL-2003-148, HP Labs, July 2003.

[8] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, December 1998.

[9] Sophia Drossopoulou, Giovanni Lagorio, and Susan Eisenbach. Flexible models for dynamic linking. In Pierpaolo Degano, editor, *European Symposium on Programming (ESOP)*, volume 2618 of *Lecture Notes in Computer Science*, pages 38–53. Springer-Verlag, 2003.

[10] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 37, number 5 in *SIGPLAN Notices*, pages 234–245. ACM, May 2002.

[11] C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1(4):271–281, 1972.

[12] Dexter Kozen and Matthew Stillerman. Eager class initialization for Java. In Werner Damm and Ernst-Rüdiger Olderog, editors, *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, volume 2469 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[13] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[14] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06v, Iowa State University, Department of Computer Science, May 2003. See `www.jmlspecs.org`.

[15] K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995.

[16] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In Martin Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.

[17] K. Rustan M. Leino and Greg Nelson. Data abstraction and information hiding. *ACM Transactions on Programming Languages and Systems*, 24(5):491–553, September 2002.

[18] K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical Note 2000-002, Compaq Systems Research Center, October 2000.

[19] Sheng Liang and Gilad Bracha. Dynamic class loading in the Java virtual machine. In *Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 36–44. ACM Press, 1998.

[20] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[21] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. PhD thesis, FernUniversität Hagen.

[22] David von Oheimb. *Analyzing Java in Isabelle/HOL: Formalization, Type Safety and Hoare Logic*. PhD thesis, Technische Universität München, 2001. `www4.in.tum.de/~oheimb/diss/`.

[23] Cees Pierik, Dave Clarke, and Frank S. de Boer. Creational invariants. In Erik Poll, editor, *Formal Techniques for Java-like Programs*, pages 78–85, 2004. Available from `www.cs.kun.nl/~erikpoll/ftfjp/2004/CreationalInvariants.pdf`.

[24] Arnd Poetzsch-Heffter and Peter Müller. A programming logic for sequential Java. In S. Doaitse Swierstra, editor, *European Symposium on Programming Languages and Systems (ESOP)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1999.

[25] Zhenyu Qian, Allen Goldberg, and Alessandro Coglio. A formal specification of Java class loading. In *Object-oriented programming, systems, languages, and applications (OOPSLA)*, pages 325–336. ACM Press, 2000.