SRC Technical Note
1997 - 007

2 January 1997

Checking object invariants

K. Rustan M. Leino and Raymie Stata

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

http://www.research.digital.com/SRC/

Abstract

When writing computer programs, programmers make assumptions about the relations among variables. In object-oriented programs, these assumptions include relations among the instance variables of a single object, relations often referred to as *object invariants*. It is a good idea to explicitly annotate a program with these assumptions. Then, a static program-analysis tool can inspect the annotated program to check that routines preserve object invariants. This paper considers two issues that affect what object invariants a program analysis tool can check: object construction and modular checking. The paper suggests some programming idioms and program annotations that widen the range of object invariants that a static program checker can check. The paper also suggests a simple extension to the Java programming language that makes the language more amenable to object-invariant checking.

KRML 74 -0

Checking object invariants

K. Rustan M. Leino and Raymie Stata
2 January 1997

Digital Equipment Corporation Systems Research Center
130 Lytton Ave., Palo Alto, CA 94301, U.S.A.
{rustan, stata}@pa.dec.com

Abstract. When writing computer programs, programmers make assumptions about the relations among variables. In object-oriented programs, these assumptions include relations among the instance variables of a single object, relations often referred to as *object invariants*. It is a good idea to explicitly annotate a program with these assumptions. Then, a static program-analysis tool can inspect the annotated program to check that routines preserve object invariants. This paper considers two issues that affect what object invariants a program analysis tool can check: object construction and modular checking. The paper suggests some programming idioms and program annotations that widen the range of object invariants that a static program checker can check. The paper also suggests a simple extension to the Java programming language that makes the language more amenable to object-invariant checking.

0 Introduction

When writing programs, programmers make assumptions about the relations among variables. In object-oriented programs, these assumptions include relations among the instance variables of a single object, relations often referred to as *object invariants*. As a simple example, an object's invariant might state that a given instance variable is never **null**. A routine accessing this variable can assume the variable is not **null** and thus can safely dereference it, but the routine must take care not to set the variable to **null**.

Unfortunately, preserving object invariants is often more tedious than simply taking care that a single instance variable is not **null**. It is all too easy to write buggy code that mistakenly breaks object invariants. These bugs are hard to find because they manifest themselves far from the actual coding errors. To avoid these bugs, a useful static program checker would inspect a program to check that routines preserve object invariants. Such a checker would take as input a program annotated with the invariants to be checked and perhaps other information. The checker would compare the code of routines against the annotations and report when the code fails to preserve invariants. Such a checker must be capable of statically checking program assertions and thus would be more like a program verifier than a `lint`-like tool. However, unlike a verifier, such a checker need not check for full, functional correctness, making it more viable for practical use. The Extended Static Checking (ESC) project has shown that such checkers are feasible and useful for everyday programs [0].

This paper discusses two issues that arise when integrating object-invariant checking into an ESC-like tool. The first issue is checking that invariants are established at object creation, which depends heavily on the semantics of the language being checked. The paper looks at the object-construction semantics of three languages, Modula-3, Java, and Theta, and describes and compares the checking rules for all three languages. It also describes a small addition to Java's constructors that enhances the utility of checked object invariants.

The second issue discussed by this paper is modular checking of object invariants. Modular checking in this context means that a class can be checked once when it is implemented and need not be re-checked in every program that instantiates it or subclasses from it. Modular checking requires placing limitations on where object invariants can be declared. The paper shows that a naïve set of limitations is overly restrictive and shows how to relax these restrictions by placing limitations on where instance variables can be updated.

Section 1 describes object invariants in a little more detail. Section 2 looks at invariants of newly-constructed objects. Section 3 looks at the requirements of modular checking. Section 4 discusses some related work, in particular *run-time assertion checking*, found in Eiffel and Anna, and *validity variables*, an alternative approach to static checking of object invariants. Section 5 presents some concluding remarks.

1 Object invariants

An object invariant is a relation among the values of the instance variables of a single object. Consider the following example.

class C {
  int f, g;
  /* **invariant** $f \leq g$; */
    ⋮
}

This program fragment declares a class C with two integer instance variables, f and g. The class is annotated with an object invariant $f \leq g$.

Object invariants are to hold at all routine boundaries, that is, on entry and exit to all routines. This rule is enforced by an object-invariant checker. For example, suppose class C declares a method m:

$$\begin{array}{l} \textbf{method}\ m()\ \{ \\ \quad f := f + 5\ ; \\ \quad g := g + 5\ ; \\ \quad n()\ ; \\ \quad \textbf{if}\ f \neq g\ \textbf{then}\ f := f + 1\ \textbf{end}\ ; \\ \}\quad . \end{array}$$

The checker assumes that the object invariant $f \leq g$ holds on entry to method m, and checks that the invariant holds at the next routine boundary, the call to method n. The checker assumes that the object invariant holds upon return from n, and checks that it holds at the following routine boundary, the end of m's method body. Note that it is possible that the object invariant does not hold between the increments of f and g; the requirement is only that it hold on routine boundaries. (It is possible to relax the requirement that invariants hold at *all* boundaries, but for simplicity we avoid this generalization.)

A method can have additional annotations that an object-invariant checker will take into consideration. For example, consider the following additional method of class C:

$$\begin{array}{l} \textbf{method}\ k(\textbf{int}\ d)\quad /\!*\ \textbf{requires}\ 0 \leq d\ *\!/\quad \{ \\ \quad g := g + d \\ \}\quad . \end{array}$$

In showing that method k maintains the object invariant, the checker assumes the precondition $0 \leq d$, and in showing that clients of C don't mess up C's object invariant, the checker enforces this precondition. We assume that any object-invariant checker under consideration can handle method annotations like preconditions, but we don't focus on them in the paper.

A subclass inherits the instance variables and object invariants declared in its superclasses. A subclass can declare additional instance variables and provide object invariants for them. For example,

class D **extends** C {
  $T\ x$;
  /* **invariant** $x \neq$ **null** ; */
  $\vdots$
}

declares D to be a subclass of class C. Class D introduces a new instance variable x of some object type T, and declares as an object invariant that x is non-**null**. The object invariant for an object of type D is the conjunction of the "local" object invariants $f \leq g$ and $x \neq$ **null** declared in classes C and D, respectively.

2 Establishing object invariants initially

Once a new object is allocated, the object's invariant must be established initially. The process of ensuring that an object's invariant holds after the object has been initialized poses some challenges to an object-invariant checker. The following Modula-3 fragment illustrates:

$$t := \mathbf{new}(T)\ ;\quad \vdots \quad P() \quad . \tag{0}$$

The construct $\mathbf{new}(T)$ allocates storage for an object of class T and returns a reference to that storage. Before returning, the call to **new** sets each instance variable of the object to some arbitrary value of the variable's type. Code fragment (0) ends with a call to some procedure P. At this procedure boundary, the invariant of the newly allocated object t is required to hold. However, establishing this object invariant is tricker than it may seem, because the code fragment may appear in a scope where not all of the object's instance variables can be accessed.

Languages differ greatly in their mechanisms for object creation and initialization, so the challenges of ensuring that object invariants hold after initialization differ in different languages. Further, from the perspective of checking invariants at object initialization time, some languages are better than others. This is because different initialization mechanisms require the checker to enforce different rules, and some rules allow more invariants to be checked than do other rules.

This section looks at the object creation mechanisms of three object-oriented languages. For each, it explains what rules an object-invariant checker would enforce to ensure that object invariants are properly established during initialization. The section also points out how some of these rules are more flexible than others.

Modula-3. In Modula-3, a new instance of a class T is allocated by calling $\mathbf{new}(T)$. A program can associate with each instance variable a constant expression called the variable's *default value*. The call to **new** initializes each instance variable of the new object to its default value if the variable has an associated default value, and to an arbitrary value of the variable's type otherwise.

In Modula-3, a call to $\mathbf{new}(T)$ may be placed in a scope where not all instance variables of T are visible. The code that immediately follows the call to **new** is then unable to affect the values of those variables. A programmer's only hope is that the default values of those variables satisfy the appropriate object invariants. Thus, rather than checking each call to **new**, an object-invariant checker performs a check for each object-invariant declaration. The check enforces that the values assigned to instance variables by **new** satisfy the object invariant. Code that invokes **new** can then simply assume the new object to satisfy its object invariant.

The downside to default values is that they rule out many useful invariants. This is especially true for instance variables that are references to other objects: the only constant value for references is **null**, so an invariant saying that an instance variable is not **null** can never be satisfied by default values. To support checking of more interesting invariants, a mechanism other than default values is needed.

Java. Java has *constructors*, code responsible for initializing newly allocated objects. Constructors give class implementors more control over the initial values of objects than do default values. However, details of a language's constructor mechanism can greatly affect the rules enforced by an object-invariant checker.

A class in Java declares a superclass, a list of instance variables, a set of methods, and a set of constructors. In addition, instance variables can be associated with *initializers*, pieces of executable code that compute initial values for the variables. A class is allowed to have several constructors taking different parameters; a client specifies which one of the constructors to invoke by providing a list of parameters whose types match those of the desired constructor. There is a syntactic restriction on the executable code given by constructors: a constructor must start by calling a constructor of the superclass, or by calling another constructor of the same class. The second of these is provided by the language so that two constructors can share code conveniently by letting one constructor call the other; the rest of this section focuses only on constructors of the first kind.

A new instance of a class T is allocated by the expression

new T(... parameters to constructor ...) .

Executing such an expression allocates space for the new object, sets every instance variable of the new object to a *zero-equivalent constant* (that is, 0 for integers, *false* for booleans, **null** for objects, *etc.*), executes the constructor specified by the types of the constructor parameters, and finally returns a reference to the object. A constructor is executed as follows. First, the superclass constructor specified in the constructor's body is executed. Then, for each instance variable that the class declares, the associated initializer is evaluated and the result is assigned to the instance variable. Finally, the rest of the constructor body is executed.

In languages with constructors, it makes sense to require that the constructor of a class take responsibility for establishing any object invariant that the class declares. This rule is easy for programmers to remember, easy for an ESC-like invariant checker to enforce, and ensures that all of an object's invariants are established on return from **new**.

The task of defining an object-invariant checker for Java is complicated somewhat by liberal rules regarding the use of "**this**" in initializer code and in constructor bodies. In initializer code and constructor bodies, the special variable **this** refers to the object being constructed. However, during the execution of superclass constructors, neither the initializer code nor the constructor bodies of subclasses have run yet, so instance variables of **this** defined by subclasses have zero-equivalent constants. These constants might not satisfy the invariants of subclasses, making any use of **this** in an initializer or constructor body unsafe from the perspective of an object invariant checker. For example, if a superclass constructor invokes a method that is overridden by a subclass, the subclass's code for the method will start executing before the subclass's constructor has had a chance to establish the subclass's invariants. (In an attempt to avoid this problem, C++ temporarily changes the method suite of **this** during the call to the superclass constructor so that method calls do not dispatch to subclass code. This does not solve the problem completely in C++, and it also introduces new difficulties.)

An object-invariant checker can take two approaches to solving the problems posed by uses of **this** during object construction. First, it can take the approach described for Modula-3 above: restrict checkable object invariants to those that are true for the default values of instance variables. However, in Java, the default values of instance variables are the zero-equivalent constants, not programmer-defined constants. This restricts checkable object invariants even more severely than in Modula-3, so severely that very little useful checking is possible.

The second approach is to outlaw most uses of **this** in initializers and constructor bodies. In particular, no methods of **this** can be invoked, and **this** cannot be assigned to global locations or passed as a parameter. The only way **this** can be used in these contexts is to read and write its instance variables. The checker will check that the object invariant for **this** holds on exit from constructors, but, because of the severe restrictions on **this** in the constructor body, it need not check that the invariant holds before then.

Theta. Like Java, the programming language Theta [1] has constructors. However, constructors in Theta were designed with object invariants in mind, avoiding the problems with uses of **this** found in Java (and other languages).

In Theta, constructors are terminated by **make** statements, that is, like **return** statements in ordinary methods, **make** statements terminate the invocations of constructors. In addition to terminating the constructor, **make** statements initialize a new object.

The form of a **make** statement is

make {*inits* ; *supercons*} **then**
⋮
end ,

where *inits* is a sequence of assignments to the instance variables defined locally by the class, *supercons* is a call to a superclass constructor, and the "**then**-block" is a block of code, like any block of code found in method bodies. When a **make** statement is executed, *inits* is executed to initialize the class's local instance variables, the superclass constructor is called to initialize the superclass's instance variables, the **then**-block is executed, and finally the constructor returns to its caller. Thus, object construction occurs in two phases: the first phase goes from subclasses to superclasses initializing instance variables, and the second phase goes from superclasses to subclasses executing **then**-blocks. In a Theta constructor, the special variable "**self**" can be used to refer to the object being constructed, but only inside **then**-blocks.

An object-invariant checker for Theta would ensure that *inits* of **make** statements establish invariants on the class's own instance variables, and that **then**-blocks, like method bodies, preserve (not establish) the class's invariant. This design of constructors is well suited for an object-invariant checker, since **self** cannot be used until the instance variables of the object have been initialized and the object satisfies its object invariant.

One may ask why **then**-blocks are included in constructors—after all, the constructor has already established the object invariant by the time any **then**-block is executed, so what else should a constructor do? Object construction often involves more than establishing the object invariant. For example, sometimes it is desirable to add the new object to some global list of objects. In such cases, it is convenient to have a built-in second phase of the construction, which is what **then** -blocks facilitate. Note that Java's constructors feature only one construction phase. Next, we explain how to extend Java to be better suited for object-invariant checking; with this extension, the constructor body emerges as a second-phase constructor.

Java extension. We can improve Java's constructors from the perspective of checking object invariants by borrowing some syntax from C++ and some semantics from Theta. In constructor implementations, before the constructor body, initializers could be given for instance variables after a colon. For example, the constructor of a class C with instance variables f and g might look like

C(... parameters to constructor ...)
   : $f(E)$, $g(E')$
{
   ... call to superclass constructor goes here ...
   ... rest of body of constructor goes here ...
}   ,

where E and E' are expressions that are allowed to refer to the parameters of the constructor but not to **this**. The semantics of such a constructor would be that f and g are first initialized to the results of evaluating E and E', then the superclass constructor is called, then the constructor body is executed. As in Theta, subclasses are given the opportunity to initialize their instance variables before the superclass constructor is called, so an object-invariant checker can allow arbitrary uses of **this** inside the constructor body.

These new initializers for instance variables obviate the need for Java's old initializers that are associated with the instance variables themselves and are executed after the superclass constructor is called. The new initializers have two advantages over the old ones. First, by executing before the superclass constructor rather than after, they avoid problems with using **this** in the constructor body. Second, by associating them with constructor definitions rather than with instance-variable definitions, they can initialize instance variables to values that depend on the constructor's parameters. (The two kinds of initializers do not interfere with one another, so Java's old initializers could be kept in the language for backward compatibility.)

Supporting the new initializers in the Java virtual machine would require a change to the byte-code verifier. Currently, the verifier ensures that the byte-code of an object constructor does not do anything to its **this** parameter until after the constructor has called a superclass constructor. To support the new initializers, the verifier has to be relaxed to allow writing to instance variables of **this** while still disallowing other uses of **this**. This change is simple, and it is backward-compatible with the current verifier.

3 Modularity and the placement of object invariants

To check that a program maintains its specified object invariants, an object-invariant checker generates *verification conditions* from the program and the object invariants. A verification condition is a logical formula that is valid only if the program maintains its object invariants. The verification condition is passed to a theorem prover to see if it is valid.

The information available in a given verification scope affects the generation of verification conditions in that scope. For example, if an instance variable f is constrained by the object invariant $f > 0$ and is not visible in some verification scope, then the verification conditions generated in that scope mentions neither f nor its invariant. We call this *modular* checking, since it allows scopes to be checked without having full information about the program. Modular checking is said to be *sound* if the truth of a modularly-generated verification condition implies the truth of the condition that would have been generated in the presence of the full program [4]. To achieve soundness, each scope must contain enough information to generate sufficiently strong verification conditions. As an example of what goes wrong when not enough information is available, if a scope contained the instance variable f but not the object invariant $f > 0$, then the verification conditions generated in that scope would not check that updates to f maintain the invariant.

The placement of object-invariant declarations relative to the placement of instance-variable definitions is crucial to the soundness of modular checking. Essentially, an object invariant must be visible in any scope where any variable it mentions is visible. We call this condition the *visibility requirement* (*cf.* [4]). Unless the visibility requirement is satisfied, updates of a variable in some scopes are not guaranteed to be constrained by every invariant that mentions the variable.

As an example, consider a class *Reader* declared as

class *Reader* { **int** *lo*, *cur*, *hi*; ... } ,

and one of *Reader*'s subclasses, *BlankReader*, declared elsewhere as

class *BlankReader* **extends** *Reader* { **int** *max*; ... } .

According to the visibility requirement, it would be legal to add the object invariant

$$lo \leq cur \;\wedge\; cur \leq hi$$

to class *Reader*, and it would also be legal to add the object invariant

$$max \text{ is a power of } 2$$

to class *BlankReader*.

For some programs, the visibility requirement can be overly restrictive, ruling out useful invariants. Consider, for example, the object invariant

$$hi \leq max \qquad . \tag{1}$$

This invariant cannot be written in either class *Reader* or *BlankReader*. It cannot be written in class *Reader* because the instance variable *max* is not visible there. It cannot be written in class *BlankReader* because there are scopes where *Reader*, and thus *hi*, are visible but where *BlankReader* and *max* are not, violating the visibility requirement. Indeed, the methods of *Reader* are in such a scope, so placing invariant (1) in *BlankReader* would mean that a method of class *Reader* might inadvertently increase *hi* beyond *max*.

Invariant (1) is taken from a real piece of code, the Modula-3 input streams library [2]. Instance variable *cur* is the index of the next character to be returned by the reader. Readers are buffered, and *lo* and *hi* are indices that bracket the characters that are stored in the buffer. *BlankReader* is a simple subclass of *Reader*. A *BlankReader* is an input stream of length *max* (which is specified during object construction) and its contents is all blank characters. Even though object invariant (1) violates the visibility requirement, this library is believed to be correct because the *Reader* implementation of the methods does not modify the instance variable *hi*. This example indicates that the visibility requirement can be relaxed for an invariant if the checker has a mechanism for "write protecting" instance variables in the invariant. Such a mechanism should ensure that code in scopes where the invariant is not visible does not modify any of the invariant's variables.

Relaxing visibility with write protection. Simple annotations can provide just such a write-protect mechanism. We describe these annotations in the context of Java, although they are applicable also to other object-oriented languages.

Like many object-oriented languages, Java features a slew of *access control modifiers*, such as **private** and **public**, that can be part of the declaration of instance variables. Ordinarily, the modifiers determine which scopes are allowed to read and write each instance variable. We propose that an object-invariant checker enforce slightly different rules. Under this proposal, the current access control modifiers are used to control the reading of instance variables only, and the writing of instance variables is controlled by additional *write modifiers*.

For each read modifier, like **private**, we introduce an analogous write modifier with a similar name, like **writable-private**. These write modifiers are included in comments after the read modifier; if an instance variable declaration does not mention a write modifier, the write modifier defaults to the one analogous to the given read modifier. For example, a simple use of write modifiers is

class *C* { **public** /* **writable-private** */ **int** *f*; ... } .

This declares a class C with an instance variable f that is publicly readable and only privately writable. (This is a common programming idiom, usually realized by declaring f as private and introducing a public method *getF* that returns the value of f.)

In addition to write modifiers that mirror existing read modifiers, we introduce a special write modifier, **writable-deferred**, which designates that write access to the instance variable is subclass-dependent. In particular, if an instance variable f is declared in a superclass S as **writable-deferred**, then a subclass T is allowed to set the write modifier of f, provided that no other superclass of T has already done so. If a variable is **writable-deferred** to a scope, then that scope is not allowed to write the variable; stated differently, **writable-deferred** does not imply the privilege to write a variable.

The special write modifier **writable-deferred** allows us to safely handle the annotation of classes *Reader* and *BlankReader* described earlier. With write modifiers, the classes can be declared as

class *Reader* {
  protected /* **writable-deferred** */ **int** *lo*, *hi*;
  protected int *cur*;
  ⋮
} ,

and

class *BlankReader* **extends** *Reader* {
  /* **writable-private** *Reader.lo*, *Reader.hi*; */
  private int *max*;
  ⋮
} .

This declares that methods of class *Reader* can read instance variables *lo*, *cur*, and *hi*, but can write only *cur*. Subclasses of *Reader* can read these instance variables too, can write *cur*, and have the opportunity to define the write modifiers for *lo* and *hi*. The subclass *BlankReader* defines the write modifiers of *lo* and *hi* in order that its methods be able to write the instance variables. Class *BlankReader* also introduces the instance variable *max*, whose read and write accesses are restricted to methods of class *BlankReader*.

With write modifiers, the visibility requirement for object invariants can be relaxed as follows: an object invariant must be visible in any scope where any variable it mentions *can be written*. Applied to the *Reader* and *BlankReader* example, this relaxed visibility requirement allows class *Reader* to declare the invariant

$$lo \le cur \;\wedge\; cur \le hi$$

and allows class *BlankReader* to declare the invariant

$$hi \le max \quad .$$

The relaxed rule is sound because every scope that can update a variable is aware of the invariants constraining the values of that variable.

Write protection and object initialization. Write protection adds a wrinkle to object initialization. We said earlier that the constructors of a class are obliged to establish any object invariant that the class declares. However, with **writable-deferred** instance variables, a class might not have permission to write to all the instance variables in its invariants, making it hard for its constructors to establish its invariants. For example, constructors for the class *Reader* must establish the condition

$$lo \le cur \;\wedge\; cur \le hi \quad ,$$

but by declaring *lo* and *hi* as **writable-deferred**, class *Reader* can write only *cur* and has given up the right to write *lo* and *hi*.

One way to deal with this wrinkle is an idiom in which constructors declare appropriate preconditions on variables like *lo* and *hi*, and leave it to subclasses to establish those conditions. In the case of class *Reader*, a precondition that suffices is

$$lo \le hi \quad .$$

This allows the class *Reader* constructor to establish its invariant, for example by setting *cur* to *lo*. The precondition dictates that the constructor for *BlankReader* initialize the instance variables *lo* and *hi* accordingly before the *Reader* constructor is called.

Unfortunately, this idiom does not work in languages like Java in which subclass constructors do not have a chance to initialize their instance variables until after superclass constructors are finished. (This is another reason to extend Java as suggested in Section 2.) The idiom also does not work unless the subclass is allowed to provide an initializer for instance variables declared (as **writable-deferred**) in a superclass.

If the idiom above cannot be used with the language at hand, one can instead change the semantics of **writable-deferred** somewhat to allow the constructors to assign initial values to those instance variables that the class declares as **writable-deferred**. Programmers can then use the following idiom to establish invariants initially: a subclass constructor passes the initial values of **writable-deferred** instance variables as parameters to its superclass constructor. For example, the *Reader* constructor would take two parameters, say *low* and *high*, and would specify the precondition $low \leq high$ (which enables *Reader* to establish the object invariant it declares) and postcondition $lo = low \wedge hi = high$ (which enables *BlankReader* to establish the object invariant it declares).

Further relaxing visibility. Even the relaxed visibility requirement is stricter than we would like. In particular, either form of the visibility requirement makes it difficult to check invariants that mention instance variables of instance variables. For example, consider the following class:

> **class** *V* {
>   **int** *x*, *y*, *cx*, *cy*;
>   **int** *xCenter*, *yCenter*;
>   **invariant** $x \leq xCenter \leq x + cx \ \wedge \ y \leq yCenter \leq y + cy$;
>   ⋮
> }   .

The invariant declared in this class satisfies the visibility requirement, because all of the variables it mentions are declared in the same class as the invariant itself. However, suppose the first four fields were replaced by a single rectangle object of type *Rect*:

$$
\begin{array}{l}
\textbf{class } V' \; \{ \\
\quad Rect\ r; \\
\quad \textbf{int } xCenter, yCenter; \\
\quad \textbf{invariant } r \neq \textbf{null} \\
\qquad\qquad\quad \wedge \;\; r.x \leq xCenter \leq r.x + r.cx \\
\qquad\qquad\quad \wedge \;\; r.y \leq yCenter \leq r.y + r.cy \; ; \\
\qquad \vdots \\
\} \qquad .
\end{array}
$$

This object invariant is not allowed because it violates the (relaxed) visibility requirement: the *Rect* instance variables x, y, cx, and cy are not declared in the same scope as the invariant. This is potentially a real problem, because there can be code that modifies these fields of a *Rect* object without knowing anything about *xCenter*, *yCenter*, or the object invariant of class V'.

Still, many programmers may find this a useful object invariant. A possible avenue toward solving this problem is to introduce the notion of *inlined* objects. The following example illustrates the idea.

$$
\begin{array}{l}
\textbf{class } V'' \; \{ \\
\quad /\!*\ \textbf{inlined}\ *\!/\ Rect\ r; \\
\quad \textbf{int } xCenter, yCenter; \\
\quad \textbf{invariant } r \neq \textbf{null} \\
\qquad\qquad\quad \wedge \;\; r.x \leq xCenter \leq r.x + r.cx \\
\qquad\qquad\quad \wedge \;\; r.y \leq yCenter \leq r.y + r.cy \; ; \\
\\
\qquad \vdots \\
\quad \textbf{method } moveX(\textbf{int } dx) \; \{ \\
\qquad r.transpose(dx, 0) \; ; \\
\qquad xCenter := xCenter + dx \; ; \\
\quad \} \\
\}
\end{array}
$$

Here, instance variable r is declared to contain only inlined objects. That declaration allows class V'' to declare an object invariant that mentions the instance variables of r, despite the fact that those are declared in class *Rect*. Since the implementation of class *Rect* couldn't reasonably be expected to maintain the V'' object invariant, the invocation of, for example, the *Rect* method *transpose* in method *moveX* above cannot be assumed to return in state where the V'' object invariant holds. Instead, the checker will expect the implementation of *moveX* to reestablish the V'' object invariant before the next routine boundary.

In order for this approach to be sound, one needs to worry about the *leaking* of r, that is, roughly, the possibility that the object referenced by r is accessible from outside the implementation of class V'. This is a complicated problem which we will not attempt to solve here. Instead, we simply note that the problem is quite related to the leaking problems that arise in the context of *dynamic dependencies* [5], a partial solution for which is described in KRML 68 [3].

4 Related work

An alternative to checking object invariants statically is to check them dynamically via run-time assertions. Programming systems often provide an **assert** pragma or macro with which a programmer can manually insert run-time assertions at routine boundaries. There are also more automated approaches; for example, invariants declared in Anna [7] or Eiffel [8] compile into run-time checks at appropriate boundaries. Historically, static checkers for object invariants have not been available, so all checking of object invariants has been done dynamically.

The relative merits of static versus dynamic checking are well understood, and this understanding applies equally to the checking of object invariants. Just as dynamic typing provides flexibility over static typing, dynamic checking of object invariants can check a larger variety of invariants than can static checking. For example, dynamic checking does not involve restrictions like those discussed for static checking in Section 3. The major advantage of static checking is that it can find errors earlier in the development cycle than can dynamic checking. A further advantage of static checking is that it checks the program for all inputs. In contrast, the effectiveness of dynamic checking depends on the selection of a good set of test cases, an arduous process that's often ignored.

The rest of this section discusses the object-invariant checking proposed in this paper with the approach taken in ESC. ESC does not have a built-in notion of object invariants. Instead, it takes a more general approach, encoding object invariants using the data abstraction features of the ESC tool. The idiom used in ESC is to introduce an abstract field *valid* for each object. The concrete representation of *valid* reveals a condition under which the object is "valid", that is, a condition describing the object invariant. An advantage of this approach is that no special treatment of object invariants is needed; reasoning about the program is done in terms of abstract variables, one of which is *valid* . A second advantage of this *validity approach*, as we shall call it, is that programmers get full control in specifying when an object is assumed to be valid and when it is not. Among other things, this second advantage allows classes to have "close" methods that destroy the validity of object, perhaps by freeing some expensive system resources that the object needs in order to be valid.

Building the concept of object invariants into the static checker requires special mechanisms in the checker and does not allow as much flexibility when it comes to, for example, allowing objects to be invalidated during their lifetime. The advantage of built-in invariants is some simplicity over the validity approach:

- Fewer abstract variables. Since object invariants hold implicitly at every routine boundary, there is no need to introduce an abstract variable *valid* .
- Simpler public interfaces. The object invariant is implicitly part of every routine's pre- and postconditions. Thus, for example, one doesn't need to manually specify that every method requires validity.
- More manageable with recursive structures. Experience with ESC indicates that there are problems with the performance of the underlying mechanical theorem prover when one attempts to reason about programs that contain recursive data structures such as trees and linked lists. Because an object invariant is local to one object, rather than being a function of the validity of the "next" object, the same performance problems do not arise with built-in object invariants.

The validity approach uses ESC's data abstraction facility, so it is not surprising that the issues that arose in designing that facility also arise when designing a checker that checks directly for object invariants. For example, data abstraction requires a visibility requirement similar to the one outlined in this paper [4]. Also, leaking is a problem for data abstraction just as it is for object invariants [5]. Finally, the need to write protect instance variables also arises in the validity approach. In the validity approach, write protection is achieved through a specification idiom called *read-only by specification* [6].

5 Conclusions

In this paper, we have outlined how the checking of object invariants can be built into an ESC-like, static checker. We focused on two issues that arise when designing such a checker: how object invariants are established initially, and in what scopes object invariants are allowed to be declared.

For the first focus, we showed how the details of a language's design influence how a checker ensures that invariants are established upon object creation. We pointed out that careful language design can enable the checking of a wider range of invariants, and suggested a backward-compatible extension to Java that improves the language in this regard.

For the second focus, we showed that an object-invariant checker can allow a larger variety of object invariants if the scopes in which instance variables can be updated is limited. We showed a methodology of write protecting instance variables, introducing a special write modifier called **writable-deferred**. This methodology enables the checking of a wider range of invariants.

References

[0] Extended Static Checking home page, Digital Equipment Corporation, Systems Research Center. On the Web at `http://www.research.digital.com/SRC/esc/Esc.html`.

[1] Theta. On the Web at `http://clef.lcs.mit.edu/Theta.html`.

[2] Mark R. Brown and Greg Nelson. I/O streams: Abstract types, real programs. In Greg Nelson, editor, *Systems Programming with Modula-3*, Series in Innovative Technology, pages 130–169. Prentice-Hall, Englewood Cliffs, NJ, 1991.

[3] David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. KRML 68, Digital Equipment Corporation Systems Research Center, July 1996.

[4] K. Rustan M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995. Available as Technical Report Caltech-CS-TR-95-03.

[5] K. Rustan M. Leino and Greg Nelson. Beyond stacks. KRML 54, Digital Equipment Corporation Systems Research Center, July 1995.

[6] K. Rustan M. Leino and Greg Nelson. Read-only by specification. KRML 58, Digital Equipment Corporation Systems Research Center, September 1995.

[7] David C. Luckham. *Programming with Specifications: An Introduction to ANNA, a Language for Specifying Ada Programs*. Texts and Monographs in Computer Science. Springer-Verlag, 1990.

[8] Bertrand Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, New York, 1988.