# Virginity: A contribution to the specification of object-oriented software

K. Rustan M. Leino Raymie Stata

*Compaq Systems Research Center,*
*130 Lytton Ave., Palo Alto, CA 94301, USA*

In object-oriented programs built in layers, an object at a higher level of abstraction is implemented by objects at lower levels of abstraction. It is usually crucial to correctness that a lower-level object not be shared among several higher-level objects. This paper unveils some difficulties in writing procedure specifications strong enough to guarantee that a lower-level object can be used in the implementation of another object at a higher level of abstraction. To overcome these difficulties, the paper presents *virginity*, a convenient way of specifying that an object is not globally reachable and thus can safely be used in the implementation of a higher-level abstraction.

*Key words:* Program specification, object-oriented programming, program verification, specification languages, formal semantics, static program checking.

## 1 Introduction

The Extended Static Checking project [6,1] is building static program checkers that take annotated programs and produce messages about possible errors. Our checkers are "lightweight program verifiers". Like all program verifiers, they are capable of reasoning about specifications and checking assertions in a program. They are "lightweight" in being similar to type checkers when it comes to performance and annotation effort. We achieve "lightweight" by checking for less than full, functional correctness; our checkers are intended for less ambitious checking, such as checking for the absence of null pointer dereferences.

In building static program checkers, an important goal is to reduce the number of spurious warning messages. Spurious messages obfuscate valid messages, greatly diminishing the value of running the checker. In the context of checkers based on program verification, one way to reduce spurious messages is to make assumptions that are not checked. However, this approach sacrifices

soundness. When soundness cannot be sacrificed, one needs annotations that can be checked easily yet are not onerous to programmers. This issue is particularly pronounced in large programs.

Well-designed large programs are built in layers: lower levels of abstraction are used in the implementation of higher levels of abstraction. A *pivot field* is an object field in the implementation of a higher-level object that refers to a lower-level object, where the state of the lower-level object is an integral part of the state of the higher-level object. For example, a stack object may have a pivot field pointing to an unbounded array object that contains the contents of the stack.

While sharing of objects is an important feature of modern programming languages, sharing of objects in pivot fields is dangerous and seldom desired. In practice, few programming errors involve undesired sharing of objects in pivot fields, but writing specifications strong enough to guarantee to a checker that such sharing does not occur can be difficult. Overcoming this difficulty is important when designing the annotation language for a programming tool like the Extended Static Checker; otherwise, the checker would produce many spurious warnings.

Before continuing, let us say a little more about the kind of program checker we have in mind. The checker's annotation language includes the declarations of program invariants and procedure specifications. We use a Larch-like notation for procedure specifications [7]; a procedure specification like

**requires** *pre* **modifies** $m$ **ensures** *post*

says that, if the procedure is started in a state satisfying precondition *pre*, the procedure terminates in a state satisfying the postcondition *post* (if it terminates at all), having modified only the values of the designator expressions listed in $m$.

We're interested in modular checking. This means that one should be able to check the implementation of a procedure using only the declarations that are in scope at the implementation. That is, declarations given in other scopes —in particular, the implementations of procedures called by the procedure being checked— should not be used to the checker.

The outline of the paper is as follows. Section 2 introduces an example that illustrates pivot fields and the danger in sharing their contents. Section 3 describes common programming patterns that cause difficulties for a programmer who hopes the checker will check the initialization of pivot fields. Section 4 describes how the concept of *virginity* can overcome these difficulties. The remaining brief sections describe our experience using our checker, some related

work, and offer some concluding remarks.

## 2 Sharing of pivots

To motivate and explain the problem of sharing the values of pivot fields, we show a program example taken from the Modula-3 library [9].

A *reader* is a buffered input stream that gives access to a source of characters. A *file reader* is a reader whose source is a file in a file system. The implementation of a file reader contains a pointer to a *file* object. A file object can be thought of as a file handle capable of communicating with the underlying file system. A file reader satisfies its reader requests by calling appropriate operations on its associated file object. The pointer to the file object is stored in the object field *sourceH*, so *sourceH* is a pivot field. We write *rd.sourceH* to denote the *sourceH* field of a file reader *rd*.

It is an invariant that the *sourceH* field of an open file reader points to an open file object. Closing a file reader closes the associated file object, which means that read operations can no longer be applied to it. It would be a bad idea for two file readers to share the same file object, since closing one of these readers would end up closing the file object of the open reader as well, falsifying our invariant that open readers are associated with open files. It would also be a bad idea if the file object underlying some file reader were part of the implementation of some other higher-level object, since this may lead to similar unexpected side effects.

In the axiomatic semantics used by the checker (*cf.* Ecstatic [10]), a field $f$ is treated as a map mapping each object to the value of $f$ for that object. If one views pivot fields in this way, that is, as maps from higher-level objects to lower-level objects, then the lack of sharing among pivot fields corresponds to injectivity within a map and disjoint ranges across maps. These properties can be described by program invariants. A program invariant is an assertion that is checked to hold in the initial program state and on every procedure boundary. For example, to record the design decision that *sourceH* is injective, a program may contain the program invariant

$$
\begin{array}{l}
(\,\forall\, rd0, rd1 : FileRd.T \;\bullet \\
\quad rd0.alloc \;\wedge\; rd1.alloc \;\wedge \\
\quad rd0.sourceH \neq \mathbf{nil} \;\wedge\; rd1.sourceH \neq \mathbf{nil} \\
\quad\; \Rightarrow\; rd0 = rd1 \;\vee\; rd0.sourceH \neq rd1.sourceH \;)
\end{array}
$$

(Here and throughout, we assume that quantified expressions do not range

over **nil**.) The type *FileRd.T* is the type of file readers (*FileRd* is the name of the file reader module and *T* is the name of the principal type exported by that module). The special object field *alloc* is used to model which objects have been allocated and which have not. In a program, objects are allocated by calling **new**. From the axiomatic semantics point of view, **new** selects an object whose *alloc* field is *false* and returns it after setting its *alloc* field to *true*. The only way to directly change *alloc* is to invoke **new**, so the language guarantees that an object is reachable from an executing program only if its *alloc* field is *true*.

Injectivity is only one aspect of preventing sharing of pivots. More generally, one may be interested in properties like disjoint ranges of pivot fields. For simplicity, we focus on injectivity in this paper. The solution we will describe applies equally well to the ensuring that the disjoint ranges property is maintained as a program invariant.

## 3 Initializing pivot fields

In this section, we show three common patterns in which a file reader initialization procedure may initialize the *sourceH* field of a file reader. We show that a checker cannot establish the injectivity property from naïve specifications, leading to spurious warning messages.

### *3.1 Requiring assignability to pivot fields*

One common pattern of initialization is suggested by the following procedure *FileRd.Init*, which initializes a given file reader *rd* to use the given *file* as its underlying file object:

```
impl FileRd.Init(rd: FileRd.T, file: File.T): FileRd.T is
  rd.sourceH := file ;
  ⋮
  (* initialization of other fields of rd *)
  ⋮
  return rd
```
(1)

The type *File.T* is the type of file objects. To prove that the assignment to *rd.sourceH* maintains the injectivity of *sourceH* among all file readers, the specification of *FileRd.Init* must require that *file* not be the underlying file

object of any other file reader. Such a specification can be written as the precondition:

$$\textbf{requires } (\forall\, r\colon \mathit{FileRd.T} \bullet\ r.\mathit{alloc} \Rightarrow\ r = \mathit{rd} \vee\ r.\mathit{sourceH} \neq \mathit{file}\ )$$

(The precondition should also require *file* to be open.) However, this specification would seem to breach data hiding, since it mentions the field *sourceH*, which is to be private to the implementation of file readers. So then how does one give a sufficiently strong precondition for *FileRd.Init*?

### *3.2 Ensuring assignability to pivot fields*

In the previous example, *FileRd.Init* took as a parameter an already allocated and initialized file object. Another common pattern of initialization is to let *FileRd.Init* create the underlying file object.

There are two conventions for object creation in the Modula-3 library. (Unlike some other object-oriented programming languages, Modula-3 does not feature object constructors that are automatically invoked when a new object is allocated. However, the problem we are about to describe arises in the presence of such constructors, too.) One of these conventions uses a procedure *New* that initializes and returns a newly allocated object. This might be used as follows in a file reader initialization procedure:

```
impl FileRd.Init(rd: FileRd.T, filename: text): FileRd.T is
  rd.sourceH := File.New(filename) ;
  ⋮
  return rd
```
(2)

where **text** is the type of text strings. However, to maintain the injectivity of *sourceH*, the specification of *New* must ensure that its result is assignable to a pivot field. Stated differently, *New* must ensure that its result is an object suitable for use in the implementation of a higher-level abstraction.

One might propose the following specification for *File.New*:

```
proc File.New(filename: text): File.T
  requires filename ≠ nil
  ensures fresh(result) ∧ result.isOpen′
```
(3)

where *isOpen* is the field that is *true* exactly for those file objects that are

open, *isOpen′* refers to the value of the *isOpen* field in the post-state of the procedure (an unprimed occurrence of *isOpen* would refer to the value in the pre-state), **result** refers to the value returned by the procedure, and **fresh** is the macro defined, for any object $o$, as

$$\mathbf{fresh}(o) \;\equiv\; \neg\, o.alloc \wedge o.alloc'$$

and used to indicate in postconditions that an object is newly allocated.

Unfortunately, freshness is not sufficient to ensure that an object can be assigned to a pivot field: trying to check that the assignment to *rd.sourceH* in (2) preserves the injectivity of *sourceH*, using (3) as the specification of *File.New*, will cause an automatic checker to dream up the following implementation of *File.New*:

```
impl File.New(filename: text): File.T is
  var r: FileRd.T in
    r := new(FileRd.T) ;
    r.sourceH := new(File.T) ;
    return File.Init(r.sourceH, filename)
  end
```
(4)

This implementation meets the specification (3) (in our methodology, any procedure is implicitly allowed to allocate new objects and modify their fields). The implementation also maintains the program invariant that *sourceH* is injective. However, it returns a file object that, despite being fresh, is already used as the *sourceH* field of another file reader. Hence, one cannot infer from specification (3) that the result of procedure *File.New* is appropriate for assignment to the *sourceH* field of an object. In other words, the specification of *File.New* fails to ensure that, in the post-state, **result** is not the value of the *sourceH* field of any allocated file reader.

While programmers are not likely to write code like (4), the checker has no way of knowing from specification (3) that implementation (4) hasn't been given. Thus, if we are to support modular checking, a different specification is required.

One might try to fix specification (3) by strengthening its **ensures** clause with the conjunct:

$$(\,\forall\, r\colon FileRd.T \;\bullet\; r.alloc \;\Rightarrow\; r.sourceH \neq \mathbf{result}\,)$$

This conjunct guarantees that the assignment in (2) keeps *sourceH* injective,

but it is not modular: the specification of *File.New* is found in an interface where, in the interest of data hiding, *sourceH* is not and should not be visible.

### *3.3 Preserving assignability to pivot fields*

The third common pattern for initialization is to let *FileRd.Init* create the underlying file object, but this time using a different convention for object creation in the Modula-3 library. This other convention for object creation uses a procedure *Init* that initializes and returns its first parameter (just like *FileRd.Init* itself). Using this convention, *FileRd.Init* might be implemented as follows:

```
impl FileRd.Init(rd: FileRd.T, filename: text): FileRd.T is
  rd.sourceH := File.Init(new(File.T), filename) ;
  ⋮
  return rd
```
(5)

In this case, to maintain the injectivity of *sourceH*, the specification of *File.Init* must imply the preservation of the property that its first parameter is assignable to a pivot field. Stated differently, *File.Init* must ensure that it does not introduce any sharing of its parameter that would make the object unsuitable for use in the implementation of a higher-level abstraction.

One might propose the following specification for *File.Init*:

```
proc File.Init(file: File.T, filename: text): File.T
  requires file ≠ nil ∧ filename ≠ nil
  modifies file.isOpen
  ensures result.isOpen′ ∧ result = file
```

The problem is that this specification allows the implementation to allocate a new file reader $r$ and assign *file* to $r.sourceH$, similar to what the problematic implementation of *File.New* does above. Where the issue above was ensuring that parameters and results are assignable to pivot fields, the issue here is specifying a procedure that *preserves* the property that one of its parameters is assignable to a pivot field.

## 4 Virginity

We now present a solution to the specification problems described in the previous section.

A *global location* is a global variable or an object field. The *reference count* of an object is the number of allocated global locations that reference the object. An object is said to be *virgin* just when its reference count is and has always been 0.

We now show how to extend a given proof system to an equivalent one in which one can reason about virginity. First, we introduce a special field *virgin* (some may call it a ghost variable or auxiliary variable, see [13] among many others) and let **new** return fresh, virgin objects. Every program statement

$$x := o$$

where $x$ denotes a global location and $o$ an object, is treated as

$$x := o \ ; \ x.virgin := false$$

Other than through such assignments, a program cannot directly change *virgin*. This approach ensures that, for any object type $T$ with a field $f$, the property

$$(\forall\, t\colon T \bullet \ t.alloc \wedge t.f \neq \mathbf{nil} \Rightarrow \neg\, t.f.virgin\ ) \tag{6}$$

is always true, and similarly for other global locations. Hence, we allow any verification to make use of this property as a given axiom.

To use virginity in our examples, we use

**requires** *file.virgin*

as a precondition of *FileRd.Init* in (1). Combined with the axioms about virginity, this precondition allows a checker to prove that the injectivity of *sourceH* is maintained. Since this version of *FileRd.Init* assigns parameter *file* to global location *rd.sourceH*, its specification must also include

**modifies** *file.virgin*

Similarly, to specify that the result of *File.New* can be assigned to a pivot

field, like in (2), we conjoin

$$\textbf{result}.\mathit{virgin}'$$

to the **ensures** clause of *File.New*. Such a specification outlaws implementations of *File.New* like (4).

The specification of *File.Init* remains as before. Because of the lack of a clause like

**modifies** *file.virgin*

objects passed in as virgins will remain virgin. This ensures that (5) will verify correctly.

### *4.1 Ease of specification*

It is worth comparing virginity to using reference counts directly, a solution we rejected. To model reference counts, one can introduce a special field *rc*. Invocations of **new** would guarantee that *rc* is 0 for the object returned. Every statement

$$x := o$$

where $x$ denotes a global location and $o$ an object, is treated as

$$x.rc := x.rc - 1 \ ; \ x := o \ ; \ x.rc := x.rc + 1$$

In our examples, procedure *File.New* would be specified to ensure $\textbf{result}.rc' = 0$, and *file.rc* would be absent from the **modifies** clause of *File.Init*.

Reference counts are not an attractive solution for several reasons. First, we need a way to deduce that client programs (1), (2), and (5) are correct. This can be done in one of two ways. One is to provide an axiom like (6), but with $t.f.rc = 0$ instead of $t.f.\mathit{virgin}$. This allows reasoning about the reference count being 0, but says nothing about other reference counts. Writing axioms for reference counts greater than 0 is much more complicated. If one chooses not to provide such complicated axioms, it seems that one gives up the precision that reference counts provide. Another way to deduce the correctness of client programs like (1), (2), and (5) is to treat pivot fields in a special way and require $u.rc = 0$ for any $u$ assigned into a pivot field. This

is a simple approach, but it does not make use of the precision provided by reference counts—only a reference count of 0 can be used.

A more disturbing problem of reference counts is that they place too much of a burden on specifiers. The specification of a procedure would need to mention in its **modifies** clause every reference count that the procedure might change. With several levels of abstraction, each changing *rc* at its constituent objects, this proposal becomes unmanageable. Alternatively, one may consider granting procedures the right to modify *rc* for many objects without having to be explicit about all such modifications. A sample proposal would be to let a procedure modify *rc* for any object, provided modifications of *rc* for parameters are explicitly given. However, with this proposal, a program fragment like

```
var x: File.T in
  x := new(File.T) ;
  Q( ) ;
  rd.sourceH := x
end
```

would fail to verify, because the specification of $Q$ does not guarantee that $x.rc$ is unchanged (despite the fact that $Q$ cannot possibly reach the value $x$).

In conclusion, keeping track of reference counts provides more precision than keeping track of virginity. However, we have argued that not only does specifying what reference counts a procedure might change become infeasible, the extra precision of reference counts is hard to use. Virginity, on the other hand, lacks this precision, yet it is strong enough to solve the problems we have described. In addition, the beauty of the virginity proposal is that the burden on specifiers is reduced to a minimum, as we argue next.

### *4.2 Virginity in specifications*

As we showed above, a problem with reference counts is that they change so often that they become a burden to the specifier. Virginity is certainly better, because an object's virginity state changes at most once. However, virginity has another property that makes it even less of a burden to the specifier: If $o$ denotes a global location, then $o.virgin$ is, and will remain, *false*, so mentioning $o.virgin$ in a specification is not particularly interesting. For that reason, $o.virgin$ is usually mentioned in a specification only when $o$ denotes a parameter or a result value.

## 5 Experience

We have introduced virginity into the annotation language used by the Extended Static Checker for Modula-3. In our experience, virginity has allowed us to annotate and check programs for which the checker would otherwise have produced spurious warnings. The burden of annotation goes mostly unnoticed in the checking of programs where virginity plays no role.

## 6 Related work

Various disciplines have been proposed for specifying and controlling the sharing of objects in object-oriented programs. These include Islands [8], linear objects [3], unique pointers [11], Balloon types [2], Flexible Alias Protection [12], and the "promises" **unique** and **limited** [4].

These other disciplines are based on type systems or similar analyses, whereas ours is based on program specification and verification. The specification framework has the advantage that it permits a wide range of checking to be done using a single style of annotation. For example, in addition to checking injectivity, the Extended Static Checker has been used to check array index bounds errors, **nil** dereference errors, and deadlocks and race conditions. A downside to the verification approach is the unpredictable and possibly low performance of the underlying theorem-prover. Fortunately, in our experience, the intractability of theorem-proving has not been an unsurmountable problem (see [6]).

The methodology of virginity has also been a useful springboard in tackling the problem of rep exposure, as described in a separate report [5]. It is unclear how these other disciplines could be extended to handle rep exposure in realistic programs.

## 7 Conclusion

We have unveiled a specification problem that arises in the context of object-oriented programs that are built in layers. It is not desirable to allow sharing of a lower-level object that is part of the implementation of a higher-level object. To avoid spurious warnings, procedure specifications must be strong enough to say whether an object is shared or can be used as an integral part of another. To solve this problem, we defined virginity for objects: an object is virgin if it is not, and has never been, reachable from a global location.

To reason about virginity, we introduced a special field *virgin*, which can be mentioned in ordinary procedure specifications and which is automatically updated with every assignment of an object to a global location. We argued, and our experience with a static checker confirms, that the use of virginity in specifications reduces spurious warnings while placing a minimal burden on specifiers.

**Acknowledgements** We are grateful to Greg Nelson for comments on a previous version of this write-up. Dave Detlefs implemented virginity in the Extended Static Checker for Modula-3. An anonymous reviewer helped improve the paper.

## References

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

[2] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97—Object-Oriented Programming, 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*, pages 32–59. Springer, June 1997.

[3] Henry G. Baker. 'Use-once' variables and linear object—Storage management, reflection and multi-threading. *ACM SIGPLAN Notices*, 30(1):45–52, January 1995.

[4] Edwin C. Chan, John T. Boyland, and William L. Scherlis. Promises: Limited specifications for analysis and manipulation. In *Proceedings of the IEEE International Conference on Software Engineering (ICSE'98)*, pages 167–176. IEEE Computer Society, April 1998.

[5] David L. Detlefs, K. Rustan M. Leino, and Greg Nelson. Wrestling with rep exposure. Research Report 156, Digital Equipment Corporation Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, July 1998. Available from `http://www.research.digital.com/SRC/publications/src-rr.html`.

[6] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, December 1998. Available from `http://www.research.digital.com/SRC/publications/src-rr.html`.

[7] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag, 1993. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing.

[8] John Hogg. Islands: Aliasing protection in object-oriented languages. In Andreas Paepcke, editor, *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'91)*, pages 271–285. ACM Press, October 1991.

[9] Jim Horning, Bill Kalsow, Paul McJones, and Greg Nelson. Some useful Modula-3 interfaces. Research Report 113, Digital Equipment Corporation Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301, December 1993. Available from `http://www.research.digital.com/SRC/publications/src-rr.html`.

[10] K. Rustan M. Leino. Ecstatic: An object-oriented programming language with an axiomatic semantics. In *The Fourth International Workshop on Foundations of Object-Oriented Languages*, January 1997. Proceedings available from `http://www.cs.williams.edu/~kim/FOOL/FOOL4.html`.

[11] Naftaly H. Minsky. Towards alias-free pointers. In Pierre Cointe, editor, *ECOOP'96—Object-Oriented Programming, 10th European Conference*, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209. Springer, July 1996.

[12] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In Eric Jul, editor, *ECOOP'98—Object-Oriented Programming, 12th European Conference*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer, July 1998.

[13] Susan Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Communications of the ACM*, 19(5):279–285, May 1976.