SRC Technical Note
1997 - 001

April 30, 1997

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

K. Rustan M. Leino and Raymie Stata

digital

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

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

Abstract

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.

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

0 Introduction

The Extended Static Checking project [0] is building static program checkers that find errors in object-oriented software. Our checkers take as input an annotated program and produce as output the discovery of errors in the program. One way to view our checkers is as "lightweight program verifiers". They are "lightweight" in the sense of being about the same as type checkers when it comes to performance and annotation effort. The are like program verifiers in being capable of reasoning about specifications and checking assertions in a program.

Well-designed large programs are built in layers: lower levels of abstraction are used in the implementations 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-buffer 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 this paper, we unveil some difficulties in writing procedure specifications strong enough to guarantee to a checker that an object can be assigned to a pivot field without introducing sharing. Overcoming these difficulties is important when designing the annotation language for a programming tool like ESC; otherwise, the checker would produce spurious warnings, which discourage programmers from using it.

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 [1]; 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 the 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 visible to the checker.

The outline of the paper is as follows. Section 1 introduces an example that we will use to motivate and explain the problem. Section 2 describes common programming patterns that cause difficulties for a programmer who hopes the checker will check the initialization of pivot fields. Section 3 describes how we use the concept of *virginity* to solve these difficulties. The remaining brief sections describe our experience using our checker and offer some concluding remarks.

1 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 [2].

A *reader* is an 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 would lead to similar unexpected side effects.

In the axiomatic semantics used by the checker (*cf.* Ecstatic [3]), 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

$$
\begin{aligned}
&\langle \forall\, rd0, rd1 \colon FileRd.T \;\triangleright \\
&\quad rd0.alloc \wedge rd1.alloc \wedge rd0 \neq \mathbf{nil} \wedge rd1 \neq \mathbf{nil} \wedge \\
&\quad rd0.sourceH \neq \mathbf{nil} \wedge rd1.sourceH \neq \mathbf{nil} \\
&\quad\; \Rightarrow\; rd0 = rd1 \vee rd0.sourceH \neq rd1.sourceH \,\rangle
\end{aligned}
$$

placed in the scope that declares *sourceH*. 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 o 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 only objects whose *alloc* field is *true* are reachable from an executing program.

Recording that the range of a pivot field is disjoint from the ranges of other pivot fields requires some help from the checker, since it is not known in one scope what other pivot fields the program contains. Because the details of how that is done are not directly relevant to the current discussion, let us instead focus on the injectivity property for the rest of this paper. The solution we will describe applies equally well to the ensuring that the disjoint ranges property is maintained as a program invariant.

2 Initializing pivot fields

In this section, we describe two common patterns of initializing pivot fields in Modula-3 and show why a checker cannot establish the injectivity property from a naïve specification.

During the initialization of a file reader, a file object is created and assigned to the reader's *sourceH* field. 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.)

The first convention uses a procedure *New* that initializes and returns a newly allocated object. This might be used as follows in the initialization of a file reader:

$$rd.sourceH := File.New(filename) \quad . \tag{0}$$

However, to maintain the injectivity of *sourceH* among all file readers, 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.

The second convention for object creation uses a procedure *Init* that initializes and returns its first parameter. This might be used as follows in the initialization of a file reader:

$$rd.sourceH := File.Init(\mathbf{new}(File.T),\ filename) \qquad . \tag{1}$$

The type $File.T$ is the type of file objects. In this case, to maintain the injectivity of $sourceH$, the specification of $Init$ must imply the preservation of the property that its first parameter is assignable to a pivot field (because procedure $File.Init$ is called before the new object is assigned to $rd.sourceH$). Stated differently, $Init$ 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.

2.0 Ensuring assignability to pivot fields

Let us investigate what the specification of $File.New$ must guarantee in order to satisfy the needs of common clients like (0).

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

$$\begin{array}{l} \mathbf{proc}\ File.New(filename\colon \mathbf{text})\colon File.T \\ \quad \mathbf{requires}\ filename \neq \mathbf{nil} \\ \quad \mathbf{ensures}\ \mathbf{fresh}(\mathbf{result}) \wedge \mathbf{result}.isOpen' \qquad , \end{array} \tag{2}$$

where **text** is the type of text strings, $isOpen$ 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, **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 (0) preserves the injectivity of $sourceH$, using (2) as the specification of $File.Init$, will cause an automatic checker to dream up the following implementation of $File.New$:

$$\begin{array}{l} \mathbf{impl}\ File.New(filename\colon \mathbf{text})\colon File.T\ \mathbf{is} \\ \quad \mathbf{var}\ r\colon FileRd.T\ \mathbf{in} \\ \qquad r := \mathbf{new}(FileRd.T)\ ; \\ \qquad r.sourceH := \mathbf{new}(File.T)\ ; \\ \qquad \mathbf{return}\ File.Init(r.sourceH,\ filename) \\ \quad \mathbf{end} \qquad . \end{array} \tag{3}$$

This implementation meets the specification (2) (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 (2) 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 may not be very likely to write code like (3), the checker has no way of knowing from specification (2) that implementation (3) hasn't been given. Thus, if we are to support modular checking, a different specification is required: with a specification like (2), clients of *File.New* would get spurious warnings when using the checker.

One might try to fix specification (2) by strengthening its **ensures** clause with the conjunct:

$$\langle \forall\, r\colon \mathit{FileRd.T} \;\triangleright\; r.\mathit{alloc} \wedge r \neq \mathbf{nil} \;\Rightarrow\; r.\mathit{sourceH} \neq \mathbf{result} \,\rangle \qquad .$$

This conjunct guarantees that the assignment (0) 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.

We have shown how a naïve specification of *File.New* does not allow its clients to assign the result value to a pivot field. A similar problem exists for parameters: with a naïve specification, one cannot ensure that a parameter to a procedure is assignable to a pivot field. For example, the implementer of a procedure like

```
impl P(file: File.T) is
  var rd: FileRd.T in
    rd := new(FileRd.T) ;
    rd.sourceH := file ;                                (4)
    ⋮
  end     ,
```

has no way of giving a precondition that ensures that it is safe to assign *file* to *sourceH* . This problem could be a real source of errors, so addressing it is important to finding those error, as opposed to just reducing spurious warnings as is the case with *File.New* above.

2.1 Preserving assignability to pivot fields

Let us turn to the use of $File.Init$ to initialize $sourceH$, see (1). The specification of $File.Init$ raises a different issue regarding pivot fields.

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

$$\begin{array}{l}\textbf{proc}\ File.Init(file\colon File.T,\ filename\colon \textbf{text})\colon File.T\\ \quad\textbf{requires}\ file \neq \textbf{nil} \wedge filename \neq \textbf{nil}\\ \quad\textbf{modifies}\ file.isOpen\\ \quad\textbf{ensures}\ \textbf{result}.isOpen' \wedge \textbf{result} = file \qquad .\end{array}$$

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 results and parameters are assignable to pivot fields, the issue here is specifying procedures that *preserve* the property that one of its parameters is assignable to a pivot field.

3 Virginity

We now present a solution to the specification problems revealed 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) and let **new** return fresh, virgin objects. Every program statement

$$x := o \qquad ,$$

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

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

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

$$\langle \forall\, t\colon T \ \triangleright\ t.alloc \wedge t \neq \textbf{nil} \wedge t.f \neq \textbf{nil} \ \Rightarrow\ \neg\, t.f.virgin \,\rangle \tag{5}$$

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 conjoin

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

to the **ensures** clause of $File.New$. This and the axioms about virginity ensure that the result is indeed assignable to a pivot field and rule out implementations like (3). Similarly, for procedure P in (4), we can say

$$\textbf{requires}\ \mathit{file}.virgin$$

to ensure that the parameter file can be assigned to a pivot field.

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

$$\textbf{modifies}\ \mathit{file}.virgin \qquad ,$$

objects passed in as virgins will remain virgin. This ensures that (1) will verify correctly.

3.0 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

$$x := o \qquad ,$$

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

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

In our examples, procedure $File.New$ would be specified to ensure $\textbf{result}.rc' = 0$, and $\mathit{file}.rc$ 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 (0) and (1) are correct. This can be done in one of two ways. One is to provide an axiom like (5), but with $t.f.rc = 0$ instead of $t.f.virgin$. 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 (0) and (1) 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. A specification would need to mention every reference count that a procedure might change in the **modifies** clause of that procedure's specification. 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. This would solve the problem with (1), since *file* is given as a parameter. However, with this proposal, a program fragment like

$$\begin{array}{l} \textbf{var}\ x\colon \mathit{File.T}\ \textbf{in} \\ \quad x := \textbf{new}(\mathit{File.T})\ ; \\ \quad Q()\ ; \\ \quad \mathit{rd.sourceH} := x \\ \textbf{end} \end{array}$$

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 make use of. 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.

3.1 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.

4 Experience

We have introduced virginity into the annotation language used by the Modula-3 Extended Static Checker. 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 is not a necessary ingredient of the annotation language.

5 Conclusion

We have unveiled a specification problem that arises in the context of object-oriented programs that are built in layers: Since it is not desirable to allow sharing of a lower-level object that is part of the implementation of a higher-level object, 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 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.

6 Acknowledgements

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

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] John V. Guttag and James J. Horning. With Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, Texts and Monographs in Computer Science, 1993. ISBN 0–387–94006–5, QA76.6.H66 1993.

[2] 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, U.S.A., December 1993. Available at `http://www.research.digital.com/SRC/publications/src-rr.html`.

[3] 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 at `http://www.cs.indiana.edu/hyplan/pierce/fool/`.