Specifying the state of modules

David L. Detlefs and K. Rustan M. Leino
5 September 1995

Digital's Systems Research Center
130 Lytton Ave., Palo Alto, CA 94301, U.S.A.
{detlefs,rustan}@pa.dec.com

0 Introduction

Programs written in languages like Modula-3 [6], Ada [0], and to some extent also C++ [3], consist of a number of interfaces and modules. The variables declared at the outermost level in a module are called *global* variables. The procedures found in a module can operate directly on the global variables declared in the same module. Each module contains a special procedure called the *module body*. The rôle of the module body is to initialize the global variables to satisfy what is thought of as the *module invariant*. In that process, a module body may invoke procedures, which may be implemented in other modules.

This situation leads to several questions. For example, how does one (in a module's interface) specify procedures that modify the state of global variables (since the global variables are not visible in the interface)? How does one specify procedures whose correctness requires that the global variables satisfy the "module invariant"? How does one specify the "module invariant"? And, since the initialization of one module can call the procedures found in other modules, how is the module initialization order determined or specified? (Note that the module initialization orders prescribed by Modula-3 and Ada are not strong enough to ensure correctness, and that the order prescribed by C++ is so strong it cannot be implemented.) In this note, we treat these problems and present their solutions.

1 Example: Text module

Let us begin by showing an example. Consider a $Text$ interface and its associated module $TextImpl$. For the purpose of specifications, the distinction between interfaces and modules are irrelevant, so we will refer to either as a *unit*. We will, however, keep "interface" and "module" in our jargon, just as a motivational device.

Interface $Text$ declares a type T, whose values represent text strings. One attribute of a T is worth revealing in the $Text$ interface, *viz.*, $valid$. For any t a T, $valid[t]$ holds just when t

properly represents a text string. One may say that $valid[t]$ holds just when the *class invariant* holds for t. Attribute $valid$ is an *abstract* property, *i.e.*, it is represented by (*concrete*) program variables in the module. Hence, $valid$ is declared to be a *specification* variable. So far, the unit looks like:

```
unit Text is
  type T ;
  spec var valid: T → bool ;
  ⋮
end      .
```

Just to provide some flavor of the specification of a typical procedure of this interface, we introduce $Length$. This procedure takes a valid T object t and returns the length of the text string that t represents.

```
proc len: nat := Length(t: T)
  requires valid[t]
```

The **requires** clause gives a precondition of the procedure. There could also be an **ensures** clause, which gives the postcondition. If either clause is absent, it defaults to $true$. The specification can also mention a **modifies** clause, which lists what the procedure is allowed to change. In this case, nothing is changed and thus the **modifies** clause is omitted. As it turns out, T objects are never changed after they are created.

Now to a procedure that will evince the problem. Procedure $FromChar$ returns a valid text object from a given single character.

```
proc result: T := Length(ch: char)
  ensures valid[result]
```

The implementation of this procedure could simply create a new text object for the given character. However, as far as particular text strings go, one-character text strings are rather common. Since there are a small number (maybe 256) of such text strings, the implementation may consider caching each one-character string that $FromChar$ returns. Thus, before creating a new text string, $FromChar$ first consults the cache.

We will consider such an implementation, of which we now show a portion.

```
unit TextImpl imports Text is
  var cache: char → T ;
  impl result: T := Length(ch: char) is
    if cache[ch] = nil then
      (* create new one-character text string *)
      cache[ch] := ...
    fi ;
    result := cache[ch] ;
  ⋮
end
```

The phrase "**imports** *Text*" makes the declarations of *Text* visible in *TextImpl*. The *TextImpl* module declares an array of *T* objects, indexed by a character. This array is then used by *FromChar*.

Notice that the implementation of *FromChar* assumes *cache*[*ch*] is either **nil** or a valid text string. That condition is the *TextImpl* "module invariant", which is initially established by the module body, *TextImpl_Init*:

```
body TextImpl_Init() is
  for i in char do cache[i] := nil .
```

The problem

Finally, we have arrived at a place where we can examine the problem. Some might (and have) imagine(d) attempting to solve the problem by introducing a specification construct like

```
module invariant Inv .
```

The idea would be to verify that the module body establishes *Inv* and to verify that every procedure within the module maintains the invariant. That, however, is not good enough. We list some problems.

P0 The first question that comes to mind is, exactly when is the "module invariant" supposed to hold? If it is something like $a = b$, then do updates of a and b need to be atomic? Or does the "module invariant" perhaps hold only at procedure boundaries? Isn't that, also, too strict?

P1 The proposal does not address the problem of module initialization order. For example, what if *TextImpl_Init* calls a procedure declared in another interface? Can *TextImpl_Init* then assume that the other module has been properly initialized?

P2 The suggested rule seems to imply that *no* procedure implemented in the module can be called until the module has been initialized. At first, this consequence appears somewhat strict (*e.g.*, why could *Length* above not be called until after *TextImpl_Init* has been executed?), but after due consideration, one finds that it is actually far too strict to be tolerable. The work performed by a module body can often be done by procedures that already exist in the module, so surely we should not ban the module body's use of such procedures.

P3 What precisely is it that every procedure in the module and the module body are allowed to modify? For example, a procedure can modify $cache[ch]$, but can it also modify other attributes of that text object?

Enough gripes. Let's get to a real solution. The lovely thing is that our solution will address and solve all of these problems at once.

2 A solution

Our solution is much inspired by Dave Detlefs and Greg Nelson's solution [1], but is more explicit, thereby permitting greater flexibility at the expense of verbosity. The present approach uses *dependencies* [4, 5], but we leave those details until the next section.

Let's return to the *Text* example, look at the meat and potatoes, and derive a different solution. Procedure *FromChar* requires of the cache to be in a "good" state (as described above). Thus, we'd like to add the precondition

requires *cache* is good

to the *FromChar* specification. However, the specification of *FromChar* lives in interface *Text* where *cache* is not visible. How can we write this condition without referring directly to *cache*? Why, through abstraction, as usual. We introduce, in interface *Text*, a specification variable *good*,

spec var $good$: **bool** ,

and present, in module *TextImpl*, the *representation* of *good*,

rep *good* **is** $good \equiv \langle\, \forall\, ch\colon \mathbf{char} \triangleright\ cache[ch] = \mathbf{nil} \vee valid[cache[ch]] \,\rangle$.

(Had we presented more attributes of text objects, like, *e.g.*, length and contents, we might have written the second disjunct above as something like

$$valid[cache[ch]] \wedge length[cache[ch]] = 1 \wedge contents[cache[ch]][0] = ch \quad ,$$

but such details are orthogonal to the point of this note.) Now, we write the precondition of $FromChar$ as

requires $good$.

Now to the $TextImpl$ module body, which gets the specification

modifies $good$
ensures $good$.

Let's take a look at a $Text$ client that calls $FromChar$, procedure P, say, with an implementation of the form

impl $P()$ **is**
$\ldots$; $t := FromChar(ch)$; $\ldots$.

From the specification of $FromChar$, we have that $good$ must hold just prior to the call to $FromChar$ from P. The implementor of P can discharge that proof obligation by simply requiring $good$.

proc $P()$
requires $good$

This turns out to be a bit undesirable, however, because it suggests that clients of P get to know that the implementation of P makes use of something that requires $good$. More concretely, let's say P is declared in an interface U and implemented in a module $UImpl$. $UImpl$ needs to import $Text$ because the implementation of P calls $FromChar$, whereas U needs to import $Text$ in order to write the specification of P.

Let's explore how interface U can be written without importing $Text$. We use abstraction and declare a specification variable $ugood$ in interface U.

spec var $ugood$: **bool**

We change the specification of P to

requires $ugood$,

and then give the representation of $ugood$ in $UImpl$.

rep $ugood$ **is** $ugood \equiv good \wedge \ldots$

The ellipsis in the previous line is meant to denote the rest of the "module invariant" of *UImpl*, which might be just *true*.

The module body of *UImpl* is to establish *ugood*. To do so, it must ensure the " . . . " above, but it must also ensure *good*. But *UImpl_Init* does not know how to establish *good*, so the only way out is for *UImpl_Init* to require *good*. We thus write the module body of *UImpl* as follows.

```
body UImpl_Init()
  requires good
  modifies ugood
  ensures ugood
  is
    (* code that establishes the "..." goes here *)
```

We have now reached the relevance of the module initialization order. Since *UImpl_Init* requires *good*, *good* must be established before *UImpl_Init* is called. The calls to module bodies are carefully arranged by the linker, which will use the module body specifications to determine the proper order of these calls. A module initialization order can be derived from the specifications of the module bodies (details are described later).

Summary

We summarize what we have seen so far. We have described the first approximation of the solution in terms of ordinary abstraction (specification variables and **rep** clauses) and usual specifications.

Module bodies, too, get specifications, and the linker uses these specifications to produce a proper sequence of calls to initialize all modules. Although the concept of a "module invariant" may be useful conceptually, it is not essential to and plays no rôle in what we have presented.

We have taken care of problems *P0*, *P1*, *P2*, and *P3*. For *P0*, since a "module invariant" holds just when some variable like *good* is *true*, we need to know when *good* needs to hold, and that follows directly from procedure specifications. For example, if a procedure *Q* does not require *good*, it is okay to change *good*, call *Q*, and then restore *good* to its previous value.

For *P1*, a module body declares explicitly what it requires and ensures, and the linker attempts to find a sequence of calls to module bodies such that the precondition of each module body is met at the time the module body is called. (We will worry about the efficiency of finding such a sequence later.)

For *P2*, each procedure is given its own specification. Thus, only those procedures that rely on *good* will list *good* in their precondition. Consequently, the module body is free to call

any procedure whose precondition it can meet—just like verifying any ol' program.

For *P3*, every procedure and module body has a **modifies** clause which declares precisely what the implementation is allowed to modify. Hence, the solution of *P3*, like the solutions of *P0*, *P1*, and *P2*, boils down to abstraction and specifications.

So, are we done? No, not really. We're interested in modular verification, and we know that modular verification (with abstraction) is sound only if representation dependencies between variables are declared in the proper places [4]. We treat that next.

3 Soundness of modular verification

Our above example shows two **rep** clauses, one for *good* and one for *ugood*. The **rep** clause of a variable *a* can be written down only if every variable mentioned in the right-hand side of the **rep** clause is a dependency of *a* [4]. Thus, we need to add the following dependency declarations to our program.

depends *good* **on** *cache* ;
depends *good* **on** $valid[cache['A']], valid[cache['B']], \ldots, valid[cache['Z']]$;
depends *ugood* **on** *good* ;
depends *ugood* **on** ...

Delving into the rules that govern where **depends** clauses can be placed (see [4, 5]), we find several things. We will concentrate on just one of those here, *viz.*,

depends *ugood* **on** *good* .

The rules that ensure soundness of modular verification state that this dependency must be placed in the same unit that declares *good*. That, of course, would be silly, because that means that *Text* needs to anticipate all clients of *FromChar* and then also import them so that the dependencies could be listed in interface *Text*. Ludicrous.

> **Remark.** It is worth noting that had we indeed been able to satisfy the rules of **depends** clause placement, we really would have been done now.

At this time, we find ourselves looking up our sleeves for aces. The intended use of variables like *good* and *ugood* is to have a module body establish them. After that, they remain *true* for the rest of the program execution (with the exception of sometimes possibly being changed temporarily). Stated differently and more precisely, every procedure and module body either leaves *good* unmodified or changes *good* to *true*. Having spotted the intended monotonicity of variables like *good*, we hope to resolve the problem by imposing a tighter discipline of the use of such variables.

Initialization variables

We introduce a special kind of specification variable called an *initialization variable*. Examples of such variables are $good$ and $ugood$ as seen above. An initialization variable $good$ is declared as follows.

init-var $good$: **bool**

Every initialization variable must have type **bool**. A few additional restrictions apply to initialization variables. Every procedure that modifies an initialization variable must also ensure it. Initialization variables are allowed in preconditions only as "positive" conjuncts. For example,

requires $good \wedge \ldots$

is allowed, whereas

requires $\neg good \wedge \ldots$

is not.

Similarly, an initialization variable is allowed to depend on other initialization variables, but only in "positive" ways. More precisely, if initialization variable a depends on initialization variable b, then the **rep** of a should imply

$$a \Rightarrow b \quad .$$

(This condition can be checked only at link-time.)

We could apply the rules about the use of initialization variables in procedure specifications also to their use in module body specifications. However, the methodology suggests a stronger rule, and we adopt the stronger rule to match the methodology and to make the job of the linker easier. We thus restrict the specifications of module bodies in the following way. The **modifies** clause must list only initialization variables, and the **ensures** clause must be the conjunction of the variables in the **modifies** clause. The precondition must be a (possibly empty) conjunction of initialization variables, each one of which is a proper dependency of some variable listed in the **modifies** clause.

With this restriction, the linker builds a graph whose vertices are the initialization variables. There's an edge from a to b just when b is declared to depend on a. If this graph represents a partial order (*i.e.*, there are no cycles), then we know from, *e.g.*, lattice theory that there exists a total order that is consistent with the partial order. If there are cycles, the linker rejects the program. If there are no cycles, the linker generates the calls to according to some (any) aforementioned total order.

4 Commentary

Several remarks are worth making. Foremost, we do not know whether or not the suggested approach yields sound modular verification. Also, we did indeed sweep several of the dependencies of $good$ under the rug. It appears that those dependencies are similar to dynamic dependencies [5], but they require further attention.

A little history

In [4], the soundness proof deals only with dependencies of the form

$$\textbf{depends } a \textbf{ on } c \quad .$$

In contrast, [5] does not mention such dependencies, but instead deals exclusively with dependencies of the forms

$$\textbf{depends } a[t] \textbf{ on } c[t]$$
$$\textbf{depends } a[t] \textbf{ on } c[b[t]] \quad .$$

In fact, in studying real programs, the authors of [5] didn't find very compelling reasons to include global variables (other than "maps", or "heap variables"), and thus did not include any dependencies like

$$\textbf{depends } a \textbf{ on } c \quad .$$

Around the same time, it was noted at a SRC Sparta meeting that the SRC Modula-3 $Text$ package contains a global variable, $cache$, whose use has been described in this note. It was then thought that a dependency of the form

$$\textbf{depends } a[t] \textbf{ on } g$$

were needed. When the authors tried to make sense of this kind of dependency, we discovered that it was not needed at all. What the situation really called for was a more explicit treatment of module state and "module invariants", which lead to the present note.

Interjection. Up until recently, another dependency of the form

$$\textbf{depends } a[t] \textbf{ on } g$$

has seemed necessary, *viz.*, when g is the set of locks held by the current thread, as is used in locking-level verification [2]. However, the compelling example where that **depends** clause would be used now has a much nicer (and understandable!) solution (stayed tuned for a forthcoming KRML note).

The present note somewhat surprisingly suggests the need for a different kind of dependency, *viz.*,

depends g **on** $a[c]$.

(In our case, c takes an unusual form, as in seen above.) Regarding this dependency and soundness of modular verification, we can remark only that it seems reminiscent of dynamic dependencies. More work is necessary to be more concrete.

Programming languages with modules

As a final observation, we note that our approach does not say anything about the placement of initialization variables *vs.* the placement of the module bodies that establish them. In fact, what we have called module bodies can be treated a special kinds of procedures that only the linker can call. Thus, one unit can actually contain any number of "module bodies".

We have also not put any restrictions on the multiplicity of module bodies that establish a particular initialization variable. We just need that every initialization variable is established by at least one module body.

References

[0] American National Standards Institute, Inc. *The Programming Language Ada Reference Manual*, volume 155 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Germany, 1983. ANSI/MIL-STD-1815A-1983.

[1] D.L. Detlefs. Mod init order. Message 409 posted to `src.sparta`, Digital's Systems Research Center, February 1995.

[2] D.L. Detlefs and G. Nelson. Locking-level verification. Ongoing research, Digital's Systems Research Center.

[3] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley Publishing Company, 1990.

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

[5] K.R.M. Leino and G. Nelson. Beyond stacks. KRML 54, Digital's Systems Research Center, July 1995.

[6] G. Nelson, editor. *Systems Programming with Modula-3*. Series in Innovative Technology. Prentice-Hall, Englewood Cliffs, NJ, 1991.