KRML 59 -0

The specification of a concurrent consumer

K. Rustan M. Leino and Greg Nelson and James B. Saxe
11 September 1995

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

A novel specification of a *consumer* is given in [2]. That specification can be written because of the use of *representation dependencies* (**depends** clauses) [4]. Due to the state of knowledge of dependencies at the time it was written, [2] does not consider different scopes. Nor does it gracefully handle dependencies on global variables. In particular, dependencies of the forms

```
    depends $a[t]$ on $c[b[t]]$
    depends $a[t]$ on $g$
```

give problems. We refer to the first of these kinds of dependencies as a *dynamic* dependency, and the latter as a dependency on a global. Given the advanced knowledge of dependencies as recorded in [5], we now know how to deal with the dynamic dependencies. In the present note, we give two specifications of the consumer, both of which use dynamic dependencies and both of which avoid dependencies on globals.

0 The cast

Readers and writers

A *reader* is an input stream and a *writer* is an output stream. The only things we need to know about readers and writers are the following (we use a notation similar or identical to that used in the references).

```
unit $Rd$ is
  type $T$;
  spec var $valid : T \to$ bool;
  spec var $state : T \to$ any;
  proc $ch$ : char := $GetChar(rd : T)$
    requires $valid[rd] \land sup.LL = rd$
    modifies $state[rd]$
end
```

```
unit $Wr$ is
  type $T$;
  spec var $valid : T \to$ bool;
  spec var $state : T \to$ any;
  proc $PutChar(wr : T; ch :$ char$)$
    requires $valid[wr] \land sup.LL = wr$
    modifies $state[wr]$
end
```

An $Rd.T$ is a reader and a $Wr.T$ is a writer. Both readers and writers have a $valid$ field and a $state$ field. The type $\mathbf{any}$ essentially denotes that we don't care what the exact type of the field is. The $GetChar$ and $PutChar$ procedures require as a precondition that $valid$ hold of the reader or writer, respectively, and each modifies only the $state$ of that reader or writer (*i.e.*, validity is maintained by the procedures). We assume there is a special character $\mathbf{eof}$ that $GetChar$ returns when invoked on a reader positioned at end-of-file.

Locks

An object can be in a *locked* or *unlocked* state. If an object is intended to be shared among threads, its fields should be accessed only if the object is locked. LL denotes the set of locked objects held by the current thread, and $sup.LL$ is the supremum (lowest upper bound) of that set with respect to some partial order on objects $<$.

An object m is locked with the programming construct

$$\mathbf{lock}\ m\ \mathbf{do}\ S\ \mathbf{end} \quad ,$$

where S is a program statement. A precondition of this construct is

$$sup.LL < m \quad .$$

The statement acquires the lock on m, executes S, and then releases the lock on m. Acquiring m means (waiting until m is not held by any other thread, and then immediately) adding m to LL. Releasing m means removing m from LL. Thus, LL is the same before and after the $\mathbf{lock}$ statement. Since $sup.LL$ is less than m before the $\mathbf{lock}$, we have

$$sup.LL = m$$

as a precondition of S. Moreover, because the $\mathbf{lock}$ statement is the only way to change LL, an invariant of any thread is that LL is totally ordered. Fields protected by m can be accessed only when $m \in LL$. Since LL is totally ordered, we have

$$sup.LL = m \Rightarrow m \in LL \quad ,$$

and thus the fields protected by m can be accessed if $sup.LL = m$.

Each of the above procedures requires that the given reader or writer be locked. (The actual code from which we draw our example [1] works a bit differently; we illustrate the problem this way to more directly get to the point.) More precisely, the procedures require that $sup.LL$ equal the reader or writer given as a parameter.

Consumers

A *consumer* is an object that features a method to which one can pass characters.

```
unit $Csum$ imports $Rd$ is
  type $T$;
  method $m(csum : T; ch : \textbf{char})$
    $(*$ specification of $m$ to go here $*)$
  ⋮
end
```

Consumers are used in conjunction with readers, so the $Csum$ unit provides the following procedure.

```
proc $Consume(rd : Rd.T; csum : T)$
    $(*$ specification of $Consume$ to go here $*)$

impl $Consume(rd : Rd.T; csum : T)$ is
  lock $rd$ do
    var $ch$ : char in
      $ch := Rd.GetChar(rd)$;
      while $ch \neq$ eof do
        $m(csum, ch)$;
        $ch := Rd.GetChar(rd)$
      end
    end
  end
```

Copier consumers

A *copier* is a particular consumer that consumes its given characters by writing them to a writer. An interface to copiers declares a procedure which takes a reader and a writer.

```
unit $Cp$ imports $Rd$, $Wr$ is
  proc $Copy(rd : Rd.T; wr : Wr.T)$
    requires $Rd.valid[rd] \land sup.LL < rd \land Wr.valid[wr] \land rd < wr$
    modifies $Rd.state[rd], Wr.state[wr]$
```

As for the given specification, the procedure must require that both rd and wr be valid and that

$$sup.LL < rd \land sup.LL < wr \qquad .$$

However, either $rd < wr$ or $wr < rd$ are possible. We chose one of these.

Procedure $Cp.Copy$ is implemented in module $CpImpl$.

```
unit CpImpl imports Cp, Csum, Rd, Wr is
  impl Cp.Copy(rd : Rd.T; wr : Wr.T)
    var cp : T in
      cp := new(T);
      dest[cp] := wr;
      Csum.Consume(rd, cp)
    end
  type T <: Csum.T;
  var dest : T → Wr.T;
  impl Csum.m(cp : T; ch : char) is
    lock dest[cp] do Wr.PutChar(dest[cp], ch) end
end
```

This unit declares a new consumer type, T, a subtype of $Csum.T$. It declares a field $dest$ for such consumers, and provides an implementation for the $Csum.m$ method for such consumers. This implementation simply locks the writer stored in the $dest$ field and calls $Wr.PutChar$ to consume the given character.

Remark. Had we chosen $wr < rd$ is the specification of $Cp.Copy$, that procedure would lock $dest[cp]$ before calling $Csum.Consume$, and $Cp.T$'s implementation of method $Csum.m$ would not lock $dest[cp]$.

1 Act I

Having introduced the players, the problem is now to write specifications for $Csum.m$, $Csum.Consume$, and $Cp.Copy$, in such a way that one can prove that every procedure and method call meets the specified precondition, and that every procedure and method implementation meets its specification.

We use the ideas from [2] and declare in unit $Csum$ the following abstract data fields.

```
spec var valid : T → bool;
spec var state : T → any;
```

Having done that, [2] gives the following specifications in unit Csum.

$$\begin{array}{l} \textbf{method}\ m(\mathit{csum} : T; \mathit{ch} : \textbf{char}) \\ \quad \textbf{requires}\ \mathit{valid}[\mathit{csum}] \\ \quad \textbf{modifies}\ \mathit{state}[\mathit{csum}]; \end{array}$$

$$\begin{array}{l} \textbf{proc}\ \mathit{Consume}(\mathit{rd} : \mathit{Rd}.T; \mathit{csum} : T) \\ \quad \textbf{requires}\ \mathit{Rd}.\mathit{valid}[\mathit{rd}] \wedge \mathit{sup}.\mathit{LL} < \mathit{rd} \wedge \mathit{valid}[\mathit{csum}] \\ \quad \textbf{modifies}\ \mathit{Rd}.\mathit{state}[\mathit{rd}], \mathit{state}[\mathit{csum}]; \end{array}$$

These specifications are indeed sufficient to prove the correctness of the implementation of $\mathit{Csum}.\mathit{Consume}$.

Now for the copier. We need to give the representation invariant for $\mathit{Csum}.\mathit{valid}$ and the appropriate dependencies for $\mathit{Csum}.\mathit{valid}$ and $\mathit{Csum}.\mathit{state}$. Following the approach in [2], we write the following in unit CpImpl.

$$\textbf{rep}\ \mathit{Csum}.\mathit{valid}[\mathit{cp} : T] \equiv \mathit{Wr}.\mathit{valid}[\mathit{dest}[\mathit{cp}]] \wedge \mathit{sup}.\mathit{LL} < \mathit{dest}[\mathit{cp}]$$

$$\textbf{depends}\ \mathit{Csum}.\mathit{state}[\mathit{cp} : T]\ \textbf{on}\ \mathit{Wr}.\mathit{state}[\mathit{dest}[\mathit{cp}]]$$

This is good enough to verify $\mathit{Cp}.T$'s $\mathit{Csum}.m$ method. However, because of the variables this $\textbf{rep}$ clause mentions, we must also declare the following dependencies.

$$\begin{array}{l} \textbf{depends}\ \mathit{Csum}.\mathit{valid}[\mathit{cp} : T]\ \textbf{on}\ \mathit{dest}[\mathit{cp}] \\ \textbf{depends}\ \mathit{Csum}.\mathit{valid}[\mathit{cp} : T]\ \textbf{on}\ \mathit{Wr}.\mathit{state}[\mathit{dest}[\mathit{cp}]] \\ \textbf{depends}\ \mathit{Csum}.\mathit{valid}[\mathit{cp} : T]\ \textbf{on}\ \mathit{LL} \end{array}$$

The first of these dependencies is a *static* one. That's fine. The second is a *dynamic* dependency. That's fine, too (see [5]). The third, however, has the form

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

i.e., a dependency on a global, and that's not allowed ([3] explains the problem with this kind of dependency, and [5] solves the problem by simply ruling out the use of dependencies on globals).

So, what to do? We must write a new specification that does not involve dependencies on globals.

2 Intermission

Popcorn, anyone?

3 Act II

The problem with the above attempt at specifying the consumer is that looking at the consumer specification itself, there is no mention of locking levels. Thus, consumer subclasses are left in the dark as to how to go about locking something inside the consumer or even locking something outside the consumer.

We thus start afresh. As before, we introduce in unit $Csum$ the following abstract data fields.

```
spec var $valid : T \to \mathbf{bool}$;
spec var $state : T \to \mathbf{any}$;
```

We also keep the specification of procedure $Csum.Consume$ as before.

```
proc $Consume(rd : Rd.T; csum : T)$
  requires $Rd.valid[rd] \wedge sup.LL < rd \wedge valid[csum]$
  modifies $Rd.state[rd], state[csum]$;
```

In Act I, the specification of $Csum.m$ does not mention locking levels. We will alter that specification so that it will mention locking levels. As it turns out, there is more than one way to do that. We take a Clue approach and offer more than one finale.

Finale A: Lower bound

We consider giving $Csum.m$ a specification that mentions a lower bound on the locking level. Looking at the implementation of $Csum.Consume$, we see that $sup.LL = rd$ holds at the time of the invocation of $Csum.m$. But writing

```
method $m(csum : T; ch : \mathbf{char})$
  requires $valid[csum] \wedge sup.LL = rd$
  modifies $state[csum]$;
```

is but nonsense, because there is no rd in scope here!

We could try adding rd as a parameter, but won't be sufficient: Then the validity of a subclass object cannot make use of this fact, because there is no guarantee that the subclass knows anything about the given reader. Stated differently, the reader would now be in the scope of the $Csum.m$ specification, but it is not in scope when writing a **rep** clause for $Csum.valid$.

Introducing the reader into the scope of $Csum.valid$ **rep** clauses is done by introducing a new data field for consumers. In unit $Csum$, we write

```
var $source : T \to Rd.T$        .
```

This allows us to write the specification of $Csum.m$ as follows.

$$\begin{array}{l}\textbf{method}\ m(csum : T; ch : \textbf{char}) \\ \quad \textbf{requires}\ valid[csum] \wedge sup.LL = source[csum] \\ \quad \textbf{modifies}\ state[csum];\end{array}$$

We need one more thing: We need to permit subclasses to mention $source[csum]$ in their **rep** clauses. To that end, we introduce in unit $Csum$ the dependency

$$\textbf{depends}\ valid[csum : T]\ \textbf{on}\ source[csum] \qquad .$$

We're getting close. Let's look at how the implementation of $Csum.Consume$ would meet the precondition of its invocation of $Csum.m$. As written, $sup.LL = rd$ holds at the time of the invocation, but we need to show $sup.LL = source[csum]$. Thus, we need $rd = source[csum]$, a condition we can guarantee if $rd = source[csum]$ is a precondition of $Csum.Consume$. So be it.

Only the copier left. In unit $CpImpl$, we place the following declarations.

$$\begin{array}{l}\textbf{depends}\ Csum.valid[cp : T]\ \textbf{on}\ dest[cp], Wr.state[dest[cp]] \\ \textbf{rep}\ Csum.valid[cp : T] \equiv Wr.valid[dest[cp]] \wedge source[cp] < dest[cp] \\ \textbf{depends}\ Csum.state[cp : T]\ \textbf{on}\ Wr.state[dest[cp]]\end{array}$$

Notice how this **rep** differs from the one given in Act I only in that $sup.LL$ is replaced by $source[cp]$. Hence, we have replaced a dependency on a global by a simple static dependency.

To meet the new precondition of $Csum.Consume$, we need the assignment

$$source[cp] := rd;$$

before the call of $Csum.Consume$ in the implementation of $Cp.Copy$.

That's a solution, but we can clean it up a little. It seems silly that $Csum.Consume$ requires $rd = source[csum]$. Why not let the implementation of $Csum.Consume$ set $source[csum]$ to rd or remove the rd parameter? The first of these suggestions won't work, because if the implementation of $Csum.Consume$ modifies the value of $source[csum]$, it may inadvertently alter the validity of $csum$, since $valid[csum]$ depends on $source[csum]$ and no **rep** of $valid[csum]$ is in scope. The other suggestion, however —removing the parameter rd from $Csum.Consume$ — appears to be a good idea.

Finale B: Upper bound

Instead of specifying a lower bound on the locking level as a precondition of $Csum.m$, we can also consider specifying an upper bound. We introduce, in unit $Csum$, an abstract data field whose purpose is to indicate the upper bound.

$$\textbf{spec var } upperbound : T \to \textbf{object}$$

The type $\textbf{object}$ denotes the type of any object—it is a supertype of every object type.

The specification of $Csum.m$ is now written as

```
$\textbf{method } m(csum : T; ch : \textbf{char})$
  $\textbf{requires } valid[csum] \wedge sup.LL < upperbound[csum]$
  $\textbf{modifies } state[csum];$        .
```

This allows the method implementation to lock $upperbound[csum]$.

Procedure $Csum.Consume$ is then specified as follows.

```
$\textbf{proc } Consume(rd : Rd.T; csum : T)$
  $\textbf{requires } Rd.valid[rd] \wedge sup.LL < rd \wedge valid[csum] \wedge rd < upperbound[csum]$
  $\textbf{modifies } Rd.state[rd], state[csum];$
```

The implementation of $Csum.Consume$ is then allowed to lock rd, and because of

$$rd < upperbound[csum] \qquad ,$$

it can meet the precondition

$$sup.LL < upperbound[csum]$$

for the invocation of $Csum.m$, since $sup.LL = rd$ holds at the time of the invocation.

For the copier, $upperbound[cp]$ equals $dest[cp]$, since it is $dest[cp]$ that the copier implementation of $Csum.m$ locks. Thus, unit $CpImpl$ provides the following declarations.

```
$\textbf{depends } Csum.valid[cp : T] \textbf{ on } Wr.valid[dest[cp]]$
$\textbf{rep } Csum.valid[cp : T] \equiv Wr.valid[dest[cp]]$

$\textbf{depends } Csum.upperbound[cp : T] \textbf{ on } dest[cp]$
$\textbf{rep } Csum.upperbound[cp : T] = dest[cp]$

$\textbf{depends } Csum.state[cp : T] \textbf{ on } Wr.state[dest[cp]]$
```

That's all.

Having seen how nicely this works out for copiers, we may want to think for a moment about $upperbound$ for consumers that don't lock anything. Let C be a class of such consumers. Calls to $Csum.Consume$ on C objects must still prove $rd < upperbound[csum]$, so $upperbound[csum]$ must be given a **rep** for C consumers. This is easy to achieve if there is a constant object value, ∞ say, whose locking level exceeds that of all other objects. Then, the **rep** clause for C consumers is given as follows.

$$\mathbf{rep}\ Csum.upperbound[csum : C] = \infty$$

4 Reviews

We presented the problem of specifying a consumer abstraction. Inspired by the ideas from [2], we presented two solutions for the problem. These solutions make use of dynamic dependencies [5], which were not understood at the time [2] was written. Moreover, the present solutions do not make use of dependencies on globals. Such dependencies were shown to be a source of trouble in [3]. Subsequently, [5] rules out such dependencies, a measure that may seem rather brutal. However, two specification problems whose solutions at one time appeared to require dependencies on globals have now been shown to have solutions that do not need them. One of those problems is described and solved in [0], and the other in the present note.

References

[0] D.L. Detlefs and K.R.M. Leino. Specifying the state of modules. KRML 57, Digital's Systems Research Center, September 1995.

[1] J.J. Horning, B. Kalsow, P. McJones, and G. Nelson. Some useful modula-3 interfaces. Research Report 113, DEC SRC, 130 Lytton Ave., Palo Alto, CA 94301, U.S.A., December 1993.

[2] K.R.M. Leino. The specification of a consumer. KRML 39, July 1994.

[3] K.R.M. Leino. Specifications and private data: A call for privatizable variables. KRML 43, September 1994.

[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.