The specification of a consumer

K. Rustan M. Leino

25 July 1994

Specifying procedure variables, whose invocation may be done in a different scope, is by some considered difficult. Giving a specification for a method is considered to be easier, even though it is just a disciplined instance of the former. For a particular example, we expose the specification of procedures, methods, and object types to scrutiny. After presenting a solution, we offer some discussion.

We assume the reader is familiar with a Modula-3-like language and Larch-like specifications.

0 The problem

0.0 Readers and writers

In the realm of readers and writers, streams of characters are of interest. We will use the following definitions of $Rd.T$ and $Wr.T$, the types of reader and writer objects, respectively.

```
type Rd.T = object                type Wr.T = object
    source : seq of char              target : seq of char
  end;                              end;
```

We will also use a procedure $Wr.PutChars$, defined as

```
proc Wr.PutChars(wr : Wr.T; ch : array of char)
  modifies Wr.target[wr]
  requires wr ≠ nil ∧ sup(LL) < wr      ,
```

where LL refers to the locking level of the calling thread, and the following $<$ is the locking-level ordering between shared objects. Regarding locking-level verification and its primitives, the reader of this note only needs to know that $<$ is transitive.

0.1 Consumers

A *consumer* is an object that takes bursts of characters as input and processes these in some potentially desirable fashion. We declare a consumer class as follows.

```
type Consumer = object
  methods
    m(ch : array of char)
  end;
```

We confess that a more sensible choice is to use the **readonly** parameter mode for ch, but such a choice is irrelevant for our discussion.

We may consider a particular consumer class, *Copier*, which consumes its input by passing it to a writer.

```
type Copier = Consumer object
    wr : Wr.T
  overrides
    m := DoCopy
  end;

proc DoCopy(self : Copier;
            ch : array of char)
  is Wr.PutChars(self.wr, ch)
```

One application of a consumer class is in conjunction with a reader. A procedure

```
proc Consume(rd : Rd.T; csum : Consumer; n : cardinal)
```

can be declared with the intention that it read the next n characters of rd, passing them in order to $csum.m$, in arbitrary sized pieces.

A procedure that uses the above is *CopyToWr*, given as

```
proc CopyToWr(rd : Rd.T; wr : Wr.T; n : cardinal)
  modifies Wr.target[wr], Rd.source[rd]
  requires rd ≠ nil ∧ wr ≠ nil ∧ sup(LL) < rd ∧ rd < wr
  is var cp := new(Copier, wr := wr); begin
       Consume(rd, cp, n)
     end      .
```

0.2 Our task

Our task is to find appropriate specifications for classes *Consumer* and *Copier* and procedures *DoCopy* and *Consume*, that will allow the proper verification of procedure *CopyToWr*.

1 A solution

We start our search for the solution from the bottom. That is, we look for what conditions are needed at each stage of the verification, to finally arrive at the specification of the *Consume* procedure. As we don't know how to write the *Consume* specification, we think our approach, although deviating from standard practices, suitable for the task at hand.

1.0 Procedure DoCopy

So, first we look at *DoCopy*. The implementation of *DoCopy* consists of a call to *Wr.PutChars*, whose specification is known. From this we deduce the necessary specification for *DoCopy*, *viz.*,

```
proc DoCopy(self : Copier; ch : array of char)
  modifies Wr.target[Copier.wr[self]]
  requires self ≠ nil ∧ sup(LL) < Copier.wr[self]      .
```

1.1 Method specification

Having done that, we focus on the classes *Copier* and *Consumer*, and their *m* method. The *Consumer* implementation of method *m* involves *self.wr*. We need that this field has an appropriate value at the time the method is invoked. For this purpose, we introduce abstract field *valid* of type **boolean**. So that the original specification of *m* can refer to this field, we place it in the *Consumer* class. For brevity, we reserve the right to in our prose abbreviate *Consumer.valid* as just *valid*.

Now we can write the precondition for method *m*.

requires *Consumer.valid*[*self*]

The concrete representation of *Consumer.valid* for *Copier* objects is the following.

depend *Consumer.valid*[*cp* : *Copier*] → *Copier.wr*[*cp*], *LL*
rep *Consumer.valid*[*cp* : *Copier*] **is** *Consumer.valid*[*cp*] ≡ *sup*(*LL*) < *Copier.wr*[*cp*]

So what about the **modifies** clause? We apply the same trick. First, we introduce abstract field *mmod* in class *Consumer*. As we only intend *mmod* to be part of the **modifies** part of specifications, we give it type **null** —a type that contains one value. This may appear funny, since there is no way to modify a variable of type **null**.

Our *Consumer* class definition now looks like

```
type Consumer = object
    valid : boolean;
    mmod : null
  methods
    m(ch : array of char)
      modifies mmod[self]
      requires valid[self]
  end;      .
```

The **depend** clause for *mmod* in *Copier* looks like

depend *Consumer.mmod*[*cp* : *Copier*] → *Wr.target*[*Copier.wr*[*cp*]] .

The added specifications now allow verifying that *DoCopy* is a refinement of *Copier.m*.

1.2 Procedure Consume

Finally, we embark on writing the specification for procedure *Consume*. The procedure reads from *rd*, so *Rd.source*[*rd*] will be modified. Moreover, the invocation of *csum.m* may modify

Consumer.mmod[*csum*] .

Now for the precondition of *Consume*. Although we didn't say anything specific about the call to some reader procedure from within *Consume*, we now say that *sup*(*LL*) < *rd* is required for this call. The invocation of *csum.m* demands *valid*[*csum*]. Of course, we also need that *rd* and *csum* be non-**nil**.

We write the specification of *Consume* as

proc *Consume*(*rd* : *Rd.T*; *csum* : *Consumer*; *n* : **cardinal**)
modifies *Rd.source*[*rd*], *Consumer.mmod*[*csum*]
requires $rd \neq \textbf{nil} \wedge csum \neq \textbf{nil} \wedge Consumer.valid[csum] \wedge sup(LL) < rd$.

This specification is strong enough for its implementation, and is also weak enough to be called from *CopyToWr* . Hence, we have achieved our goal.

2 Discussion

We conclude by offering some discussion of the solution.

2.0 Subtype-specific depend clauses

Note that proving the call to *Consume* from the implementation of *CopyToWr* requires that *cp* be of type *Copier* —otherwise, *valid*[*cp*] cannot not be established, because the **rep** clause for *valid*[*cp*] is not be visible.

Other than this, there is no difference in applying **depend** clauses to preconditions *vs.* applying them to **modifies** clauses, for which they were first designed. So, for example, a concretization of a particular slice of an abstract field can only be done once.

2.1 Scope

We didn't mention much about scopes in this note. There is, however, only one place where the issues of scopes may play a rôle in this excursion, *viz.*, mentioning *LL* in the **depend** clause for *valid*[*cp* : *Copier*] . We can easily imagine class *Copier* and procedures *DoCopy* and *CopyToWr* to be hidden in a module. Then, successive invocations of *csum.m* from within one call to *Consume* may inadvertently alter the variables on which *valid*[*csum*] depends.

The rule for where a **depend** clause may appear requires that

depend $a \rightarrow c$

be visible everywhere both *a* and *c* are. Since *LL* is global, the declaration

depend $Consumer.valid[cp : Copier] \rightarrow LL$

must be visible everywhere *valid* is. Note that it is not sufficient to only require it to be visible where also *Copier* is visible, since *Copier* objects may be used outside the scope of the *Copier* type declaration.

So, as stated, this **depend** clause must appear at the declaration of the *Consumer* class. But then the clause cannot be as specific as giving the dependency only for the *Copier* subtype, since *Copier* is not in scope. Instead, the only choice is to write it as

depend $Consumer.valid \rightarrow LL$,

which means that *valid* may depend on *LL* for any subtype of *Consumer* .

Moreover, if the original design of *Consumer* did not foresee the need for *valid* to depend on *LL*, the interface where *Consumer* is found must be modified. This requires that the designer of *Copier* have access to modify the said interface. Furthermore, the implementation of procedure *Consume* needs to be reverified. In fact, depending on how the old implementation of *Consume* used *LL*, a new implementation may need to be provided.

Both of these objections can be handled by arguing it is either the intention of the *Consumer* class designer that the validity of consumers may indeed depend on *LL*, or that the subtype *Copier* is invalid.