

Beyond stacks

K. Rustan M. Leino and Greg Nelson

14 July 1995



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

Finding a methodology guiding the writing of modular, object-oriented programs turns out to be much harder than we expected. A milestone in this quest was the invention of the specification primitive **depends** (see [4]). However, [4] left some open problems. In this note, we present a methodology that addresses those problems. We do not know that the methodology is sound, nor do we know that a mechanical tool can actually check that a program follows the methodology. We have, however, no evidence to the contrary. In the examples we've tried so far, we have found the methodology more versatile than we had hoped for.

This note explains the methodology and provides programming examples that exhibit its features. This note does not provide a soundness proof, but tries to explain why certain restrictions apply and why it's okay to lift some restrictions in certain situations.

Enough advertisement. Now to the real stuff.

0 The programming notation

In this section, we describe the programming notation. We sprinkle small examples through the presentation, and conclude the section with an example program.

0.0 Objects

We consider an object-oriented notation similar to the one in [5]. Variables and values have types. There are *scalar* types (think of integers and booleans) and one *object* type. Collectively, these are called *primitive* types.

The values of scalar types can be directly named in a program (*e.g.*, the boolean *false* or the integer 3), but there is only one object constant, **nil**. Other objects are retrieved by invocations of **new**. For an l-value *t* of an object type, the statement

$$t := \mathbf{new}()$$

assigns to *t* a “new” object, *i.e.*, a non-**nil** object that has never been returned by **new** before. (We can describe this more formally, but won't do so in this note.)

There are also *map* types. Think of a map as an array whose index type is an object type and whose element type is a primitive type. Maps model *fields* of objects.

The declaration

$$\mathbf{var} \ f$$

declares a map *f*. One can declare several maps at the same time by giving a list of identifiers after **var**. For *t* an object, the value of *f* at *t*, *i.e.*, the value of *t*'s *f* field, is denoted $f[t]$. In many programming languages,

this is instead denoted $t.f$, but we choose $f[t]$ here to stress the difference between an object (t) and a part of the *state* of that object ($f[t]$). Objects are seldomly modified—it is usually the state of an object that is modified.

Methods are procedures associated with objects. A method m is declared (and specified) by

method $res := t.m(x)$ **requires** Pre **modifies** w **ensures** $Post$,

where t is the formal name of the object on which the method is invoked, x is a list of formal in-parameters, and res is a list of formal out-parameters. If res is the empty list, the “:=” is omitted from the declaration. What follows is the method’s *specification*, which consists of a *precondition* Pre , a list of variables w (called the *frame*) that the method is allowed to modify, and a *postcondition* $Post$. The postcondition is allowed to refer to the initial state by subscripting a map by 0. res is implicitly part of the frame. If any of the clauses in the specification is omitted, Pre and $Post$ default to *true* and w defaults to the empty list.

Example 0. In the example

```
var f
method old := t.swapF(new)
  modifies f[t]
  ensures old = f0[t] ∧ f[t] = new ,
```

method $swapF$ replaces the value of $f[t]$ with the given parameter new . The method returns the previous value of $f[t]$, *i.e.*, the value of f_0 —the initial value of f — at t . ■

Example 1.

```
var f
method t.dec()
  requires 0 < f[t]
  modifies f[t]
  ensures f[t] = f0[t] - 1
```

In this example, method dec decrements the value of f at t by 1. The method aborts if $f[t]$ is not initially positive. ■

The *implementation* of the above method is declared by

method $res := t.m(x)$ **is** S ,

where S is a *guarded command* [1] over the program state space (the maps) and the variables res , t , and x . There is a proof obligation associated with providing an implementation, *viz.*, that the implementation *meet* its specification. In this note, we will be very informal about what that means.

Example 2. The following declares an implementation of the method $swapF$ in Example 0.

```
method old := t.swapF(new) is
  old, f[t] := f[t], new
```

■

For o an expression of an object type, the statement

```
 $v := o.m(a)$ 
```

invokes method m on o with actual in-parameters a and actual out-parameters v . This requires that o be non-**nil**.

Example 3. If t starts off non-**nil**, then

```
 $f[t] := 6 ; prev := t.swapF(8)$ 
```

results in t being unchanged, $f[t]$ being 8, and $prev$ being 6. ■

0.1 Modules

A program consists of a number of *units*, often known as *interfaces* and *modules*. One unit can refer to entities declared in other units by *importing* those other units. An identifier declared in the foreign unit is then denoted by the name of the foreign unit, followed by a period, followed by the name of the identifier—we say that this identifier is *qualified* by the name of the foreign unit.

A unit U is declared by

```
unit  $U$  imports  $I$  is  
   $declarations$   
end ,
```

where I lists the names of the units being imported. The order of declarations within a module and the order of the names in I bear no significance. If I is the empty list, “**imports**” is also omitted.

Example 4.

```
unit  $A$  is var  $f$  end  
unit  $B$  is var  $g$  end  
unit  $C$  imports  $A, B$  is  
  method  $t.m()$  modifies  $A.f[t], B.g[t]$   
end
```

This example shows how unit C imports units A and B so that it can mention the identifiers f and g declared in those respective units. In C , those identifiers are qualified by “ $A.$ ” and “ $B.$ ”, respectively. ■

As a matter of convenience, a unit name listed in I can be followed by **unqualified**, which allows the identifiers from the foreign unit to be mentioned unqualified in U (provided no name clashes arise). This feature has no bearing on the meat of this note—it simply makes the examples easier to read.

Example 5. Unit C in Example 4 could also have been written, for example, as

```
unit  $C$  imports  $A$  unqualified,  $B$  is  
  method  $t.m()$  modifies  $f[t], B.g[t]$   
end .
```

■

Example 6. The common interface/module paradigm is, in our notation, often written as follows.

```

unit Stack is
  var contents
  method t.push(x) modifies contents[t]
  ...
end

unit StackImpl imports Stack unqualified is
  method t.push(x) is contents[t] := contents[t] ++ x
  ...
end

```

■

Units are verified one at the time. When a unit U is being verified, the only information provided to the verifier is the set of declarations found in the units of U 's *import closure*, i.e., U , U 's imports, their imports, and so on. This is known as *modular verification*.

Example 7. In the program

```

unit A is var f end
unit B is var g end
unit C imports A is var h ... end
unit D imports B, C is ... end ,

```

the import closure of A is A ; of B is B ; of C is A, C ; and of D is A, B, C, D . ■

A declaration is said to be *visible* in a unit whenever it is in the import closure of that unit. The set of units in an import closure is called a *scope*.

Example 8. In Example 7, $A.f$ is visible in A , C , and D , but not in B . The set of units $\{A, C\}$ makes up a scope, and so do the sets $\{B\}$ and $\{A, B, C, D\}$. ■

0.2 Abstraction

Consider the following scenario. A method is declared (and specified) in one unit and implemented in another. The implementing unit imports the declaring unit, but not *vice versa* because of the well-known methodology of *data hiding*. Moreover, the implementing unit is where the fields of the implementation are declared. How, then, is the specification going to say what the method does when the state the method modifies is “hidden” in the implementing unit? The answer is: “by abstraction”.

The idea is that the declaring unit specifies the method *abstractly*. This means it declares some *abstract* (as opposed to *concrete*) variables and then gives the specification in terms of those. (Abstract variables are also called *specification variables* and concrete variables are also called *program variables*.) A specification variable a (a map) is declared by

```

spec var a .

```

The relation between an abstract variable and its concrete representation is called the abstract variable's *representation invariant*. It is given by a **rep** clause, which has the form

$$\mathbf{rep} \ a[t] \ \mathbf{is} \ R \quad ,$$

where t is a dummy that can be used in R , and R is a predicate over the state and t . It is typical that R determines $a[t]$ uniquely.

Example 9. An example **rep** clause is

$$\mathbf{rep} \ a[t] \ \mathbf{is} \ a[t] = c[t]^2 + d[t] \quad ,$$

which states that $a[t]$ represents $c[t]^2 + d[t]$. An increase of $d[t]$ thus results in an equal increase of $a[t]$. Increasing $a[t]$ can be achieved in many ways, each altering $c[t]$ and $d[t]$ accordingly. Note also that changing the sign of $c[t]$ has no effect on the value of $a[t]$. ■

In order for a term $c[t]$ or $c[b[t]]$ to be allowed to be mentioned in R above, it must be a *dependency* of $a[t]$. Dependencies are declared by

$$\mathbf{depends} \ a[t] \ \mathbf{on} \ c[t]$$

and

$$\mathbf{depends} \ a[t] \ \mathbf{on} \ c[b[t]] \quad ,$$

respectively. The t is a dummy denoting any object. The first of these two dependencies of a on c is called a *static* dependency, because $a[t]$ always depends on $c[t]$. The second form is called a *dynamic* dependency, because the index of c on which $a[t]$ depends may change during the course of a program execution. We call b the *pivot* of the dynamic dependency. The right-hand side of the **depends** clause can be a list of terms, allowing multiple dependencies to be declared on the same line.

Example 10. In order to write down the **rep** clause in Example 9, a programmer must provide, in the same scope, the declaration

$$\mathbf{depends} \ a[t] \ \mathbf{on} \ c[t], d[t] \quad .$$

■

The **depends** clauses introduce an ordering among terms. For example, we may think of

$$\mathbf{depends} \ a[t] \ \mathbf{on} \ c[t]$$

as declaring, for all t , $c[t] \leq a[t]$. The *downward (upward) closure* of a term $a[t]$ is the set of terms below (above, respectively) $a[t]$ in the reflexive, transitive closure of the this ordering. In the **rep** clause for $a[t]$, the only terms allowed in R are those in the downward closure of $a[t]$.

Example 11. Given

$$\begin{aligned} &\mathbf{depends} \ a[t] \ \mathbf{on} \ c[t] \\ &\mathbf{depends} \ b[t] \ \mathbf{on} \ c[t] \\ &\mathbf{depends} \ c[t] \ \mathbf{on} \ e[t], f[t] \\ &\mathbf{depends} \ d[t] \ \mathbf{on} \ f[t] \quad , \end{aligned}$$

the downward closure of $a[t]$ is $a[t], c[t], e[t], f[t]$. The downward closure of $e[t]$ is $e[t]$. The upward closure of $c[t]$ is $a[t], b[t]$. The upward closure of the downward closure of $d[t]$ is $a[t], b[t], c[t], d[t], f[t]$. ■

Example 12. Closures of dynamic dependencies are more involved. Given

```
depends a[t] on c[b[t]]
depends c[s] on e[d[s]]    ,
```

the downward closure of $c[s]$ is $c[s], e[d[s]]$, and the downward closure of $a[t]$ is $a[t], c[b[t]], e[d[b[t]]]$. The upward closure of $c[s]$ contains $a[t]$ for every t that satisfies $b[t] = s$. The upward closure of $e[r]$ contains $c[s]$ for every s satisfying $d[s] = r$ and $a[t]$ for every t satisfying $d[b[t]] = r$. ■

We will explain the restrictions guiding the use of **depends** clauses in a later section.

0.3 Program example

Finally, we give a more substantial example (we give several more toward the end of this note). The example is the to many familiar prime sieve of Eratosthenes. We show only the “filter” part. You may notice that the specifications we give are in the “extended static checking” style, rather than the full specification style (see [4]). We provide but a rather shallow explanation of the program here—this example is just to give you a flavor of the kind of program one might write in our notation.

```
unit Filter is
  spec var valid
  spec var state
  method t.Init(val)
    requires val ≠ 0
    modifies valid[t], state[t]
    ensures valid[t]
  method t.Try(n)
    requires valid[t]
    modifies state[t]
  method res := t.Query(n)
    requires valid[t]
end
```

The *valid* / *state* paradigm (where *state* often is a set of several fields, rather than just one as shown here) is quite common in the “extended static checking” style specifications of our programs. The boolean *valid*[*t*] essentially tells us that the *object invariant* of *t* holds (see, e.g., [8]); thus, the *Init* method establishes *valid*[*t*], and all other method require *valid*[*t*] and leave its value unchanged.

```
unit FilterImpl imports Filter unqualified is
  var value
  var next
```

```

depends  $valid[t]$  on  $value[t], next[t], valid[next[t]]$ 
depends  $state[t]$  on  $next[t], state[next[t]]$ 

rep  $valid[t]$  is  $valid[t] \equiv value[t] \neq 0 \wedge (next[t] \neq \mathbf{nil} \Rightarrow valid[next[t]])$ 

method  $t.Init(val)$  is
   $value[t], next[t] := val, \mathbf{nil}$ 

method  $t.Try(n)$  is
  if  $n \bmod value[t] = 0 \rightarrow skip$ 
  ☒  $next[t] = \mathbf{nil} \rightarrow next[t] := \mathbf{new}(); next[t].Init(n)$ 
  ☒  $next[t].Try(n)$ 
  fi

method  $res := t.Query(n)$  is
  if  $n \leq value[t] \rightarrow res := n = value[t]$ 
  ☒  $n \bmod value[t] = 0 \rightarrow res := false$ 
  ☒  $next[t] = \mathbf{nil} \rightarrow res := true$ 
  ☒  $res := next[t].Query(n)$ 
  fi

end

```

The symbol ☒ is pronounced “else” [9]. Note the absence of a **rep** clause for *state*—permitting such absence is a great feature when writing extended static checking specifications.

1 Dependencies

First, we discuss the interpretation of **modifies** clauses in the presence of dependencies, a subject that will introduce the concept of a dependency being *exempt*. Then, we treat the restrictions placed on the use of static and dynamic dependencies, respectively.

1.0 Interpretation of modifies clauses

Proceeding by example, we discuss the effect of dependencies on the interpretation of **modifies** clauses.

If a method m is specified using

```
modifies  $a[t]$  ,
```

then m is allowed to modify the value of a (only) at t . In addition, m is allowed to modify anything in the downward closure of $a[t]$. That is, if $a[t]$ depends on $c[t]$ (i.e., $c[t]$ is in the downward closure of $a[t]$), then m is also allowed to modify $c[t]$.

If a specification allows modification of $a[t]$ as well as t , for some expression t , then it is interpreted as allowing a to be changed at the initial value of t .

Example 13. The stated rule is perhaps most pronounced in the case of dynamic dependencies. Consider the dependencies

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

and the frame given by

modifies $a[t]$.

This allows the values of $a[t]$ and $b[t]$ to be changed. In addition, it permits a change of c at $b_0[t]$, *i.e.*, the value of $b[t]$ at the time the method is invoked. Notice, thus, that the index at which c is allowed to be changed is determined at the time the method is invoked; in particular, the implementation cannot change $b[t]$ arbitrarily and then expect to change c at the new value of $b[t]$. ■

So then, the downward closure of the frame may be modified; so what about its upward closure? Let us divide dependencies into two classes: *exempt* and *nonexempt*. The exempt upward closure is the same as the upward closure, but considers only those dependencies that are exempt. The **modifies** clause

modifies $c[t]$

does grant the method the right to modify the exempt upward closure of (the downward closure of) $c[t]$, but not the rest of $c[t]$'s upward closure. That means the change of the value of $c[t]$ is confined to such changes that do not alter the values of that “rest of” $c[t]$'s upward closure.

Example 14. Here's a larger example.

depends $a[t]$ **on** $c[t]$
depends $b[t]$ **on** $c[t]$
depends $c[t]$ **on** $d[t]$

Let the first dependency be a nonexempt dependency and the second be an exempt dependency. A method m , declared as

method $t.m()$ **modifies** $c[t]$,

is allowed to modify the values of $c[t]$ and $d[t]$ (because they are in the downward closure of $c[t]$) and $b[t]$ (because the dependency of $b[t]$ on $c[t]$ is exempt), but not $a[t]$ (because the dependency of $a[t]$ on $c[t]$ is nonexempt). ■

1.1 Static dependencies

Let us discuss the placement of static dependency declarations in a program. The *visibility requirement* [4] mandates that a dependency of $a[t]$ on $c[t]$ be visible (either directly in the form

depends $a[t]$ **on** $c[t]$

or indirectly by being inferable by transitivity from other **depends** clauses) whenever both a and c are visible. (Note, a and c are visible wherever this dependency is visible—otherwise, the compiler would complain with an “identifier not known” error when processing the **depends** clause.) All static dependencies must adhere to the visibility requirement—not doing so means the verification process is unsound, *i.e.*, the verification process may emit false positives (see [4]).

An easy way to enforce the visibility requirement is to make sure the declaration

depends $a[t]$ **on** $c[t]$

is placed in the same unit as the declaration of a (“near a ”) or in the same unit as the declaration of c (“near c ”).

Another important attribute of a dependency is whether or not it is *authentic* [4]. A dependency of $a[t]$ on $c[t]$ is authentic just when a is visible whenever c is. Stated in conjunction with the visibility requirement, a dependency of $a[t]$ on $c[t]$ declared near c is authentic, whereas declaring it near a and not near c makes the dependency unauthentic.

Example 15. Consider four units:

```

unit A is spec var a end
unit C imports A unqualified is
  spec var c
  depends a[t] on c[t]
end
unit D imports C unqualified is
  var d
  depends c[t] on d[t]
end
unit B imports C unqualified is
  spec var b
  depends b[t] on c[t]
end .

```

All dependencies declared in this example adhere to the visibility requirement. The dependency of a on c is declared near c , so it is authentic, and similarly for the dependency of c on d . The dependency of b on c , however, is not authentic, since it is declared near b and not near c . ■

In order for modular verification to be sound, only authentic dependencies can be nonexempt (*cf.* [4], although it does not use the term “exempt”). Unauthentic dependencies are fine if treated as exempt dependencies. However, we have been unable to find or imagine the use of unauthentic static dependencies; thus, we declare them illegal, a rule given the name *authenticity requirement* [4]. Having decided that, we now define every (legal) static dependency to be nonexempt.

Example 16. Trying to feed the units in Example 15 to a checker that uses our programming methodology would result in an error in unit B . The error message would read something like:

Error: Declaration **depends** $b[t]$ **on** $c[t]$ not near declaration of c .

or maybe:

Error: Declaration **depends** $B.b[t]$ **on** $C.c[t]$ must be placed in unit C .

The checker is able to detect the error because c is not declared in unit B . ■

Here’s a summary of what we just said.

Summary, static dependencies. Static dependencies are nonexempt. This plays a rôle when interpreting **modifies** clauses. Static dependencies must adhere to the visibility and authenticity requirements. This can be achieved by placing

$$\mathbf{depends} \ a[t] \ \mathbf{on} \ c[t]$$

near c . ■

1.2 Dynamic dependencies

A dynamic dependency takes the form

$$\mathbf{depends} \ a[t] \ \mathbf{on} \ c[b[t]] \quad ,$$

where a and c are allowed to coincide (as in

$$\mathbf{depends} \ a[t] \ \mathbf{on} \ a[b[t]] \quad).$$

First, let's discuss the effect of dynamic dependencies on **modifies** clauses.

Example 17. Let us review the using of the downward closure in expanding a frame. Consider the first of the above dependencies. If the **modifies** clause of a method m reads

$$\mathbf{modifies} \ a[t] \quad ,$$

then the method implementation is allowed to modify a at t and c at $b[t]$ (where $b[t]$ is evaluated in the pre-state). ■

We incorporate the following decision into our methodology: dynamic dependencies are exempt.

Example 18. Given the first of the two dependencies shown above, if m is specified with

$$\mathbf{modifies} \ c[s] \quad ,$$

then m is allowed to modify c at s . Moreover, m is allowed to modify a at any index t such that $s = b[t]$, since the dependency of $a[t]$ on $c[b[t]]$, being a dynamic dependency, is exempt. (See also Example 12.) ■

Now, let's explain where dynamic dependencies may be written. If a dynamic dependency declaration adheres to the visibility requirement, no further restrictions apply. However, it will often be convenient (and necessary) to stretch the visibility requirement for dynamic dependencies. Remember, living up to the visibility requirement means placing

$$\mathbf{depends} \ a[t] \ \mathbf{on} \ c[b[t]]$$

near a or near c . More frequently, however, it will be desirable to instead place it near b , in which case we say the **depends** clause *stretches* the visibility requirement. We will allow this, but will restrict the values that $b[t]$ may take on, as explained after the following definitions.

The *state graph* of a program state is a graph whose vertices are the values of all objects. Edges of the state graph are labeled; there is a directed edge labeled b from vertex t to vertex s just when there is a map b in the

state such that $b[t] = s$. We say that a vertex x reaches a vertex y only via a vertex t just when every nonempty path from x to y contains t .

A unit A exports a method m to a unit B exactly when A contains the implementation of m and B contains the specification of m .

The pivot set of a term $d[t]$ is the downward closure of $d[t]$, where every term $e[s]$ is replaced by s .

Example 19. Given

```

depends  $a[t]$  on  $c[b[t]]$ 
depends  $c[s]$  on  $d[s]$ 
depends  $d[s]$  on  $g[f[s]]$ 
depends  $g[u]$  on  $h[u]$  ,

```

the downward closure of $c[b[t]]$ is

```

 $c[b[t]], d[b[t]], g[f[b[t]]], h[f[b[t]]]$  .

```

The pivot set of $c[b[t]]$ is thus

```

 $b[t], b[t], f[b[t]], f[b[t]]$  ,

```

or, since the pivot set is a set,

```

 $b[t], f[b[t]]$  .

```

■

Now, we state the restriction regarding a dependency

```

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

```

that stretches the visibility requirement: For any method m and units A and B , where b is visible in A but not in B , if A exports m to B , then the implementation of m must satisfy the following postconditions:

- For every in-parameter p of m , for every t , and for every x in the pivot set of $c[b[t]]$, $x = \mathbf{nil}$ or p_0 reaches x only via t .
- For every out-parameter q of m , for every t , and for every x in the pivot set of $c[b[t]]$, $x = \mathbf{nil}$ or q reaches x only via t .

The precondition of m is also extended with the first of these bullets (but with p instead of p_0).

The above rule appears rather complicated, and it is. However, in most cases in practice, the pivot set of $c[b[t]]$ consists only of $b[t]$. Therefore, in the jargon, we refer to the above rule simply by saying “ $b[t]$ may not be leaked”.

We have now explained the rules guiding the use of dynamic dependencies. Here’s a summary.

Summary, dynamic dependencies. Dynamic dependencies are exempt. If a dynamic dependency adheres to the visibility requirement, no further restrictions apply. Programmers are allowed to stretch the visibility requirement by instead declaring a dynamic dependency

```

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

```

near b , provided the values of the $b[t]$ ’s are not leaked. ■

Authenticity does not play a rôle in the use of dynamic dependencies.

In a section below, we give some examples of programs that make use of static and dynamic dependencies. As stated earlier, we have not proved that the rules we have stated suffice to make modular verification sound. And, even if they do make modular verification sound, we have not showed that one can actually enforce these rules in practise, *i.e.*, by a mechanical checker. We think, however, that the second of these concerns is simpler than the first.

Regarding the adequacy of the presented proposal, we have found great satisfaction in trying it out on some of our programming examples. While we will try more examples, soundness remains to be proven.

2 Modifying the state of newly allocated objects

We need to describe the relation between **modifies** clauses and **new**. More precisely, we should say something about modifying the state of newly allocated objects. We state (again, informally) the rule as follows: A method is implicitly allowed to modify $c[t]$, where t is an object that was not allocated on entry to the method. We effect this rule for any (abstract or concrete) map c .

Readers familiar with the “**new** scandal” revealed in [3] may be suspicious. After all, we have not defined a notion of “privatizable” here at all, which is what [3] calls for. However, we think we can maintain soundness of modular verification because of two things: (0) Our dependencies are restricted to being of two forms. In particular, there is no dependency of the form

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

(1) We will require the following to be an invariant of the program: For every map c whose element type is an object type, and every unallocated object t , $c[t] = \mathbf{nil}$. Provided this condition holds initially, we can prove that it is invariant by (a) not allowing unallocated objects to be named in the program, and (b) defining the block statement

var x **in** S **end** ,

where x is being declared as a local variable of an object type, to be syntactic sugar for the guarded command

$\llbracket x \bullet x := \mathbf{nil}; S \rrbracket$.

Again, to be sure of our claim, we need to prove it, which we have not yet done.

3 Examples

We have been extremely pleased with the kinds of examples that we easily have been able to specify using the present methodology (perhaps because it is not sound?). In this section, we provide some of those examples.

3.0 Writers that use sequences

Imagine having a class (object type) of “writers”, *i.e.*, of output streams. Such a class may be specified in terms of an abstract variable *target*, which represents, for each writer, the sequence of characters that have been written to

the writer.

```

unit Wr is
  spec var target
  ...
  method wr.PutChar(ch)
    modifies target[wr]
    ensures target[wr] = target0[wr] ++ ch
  end

```

The implementation may want to make use of a buffer, to, for example, buffer some characters before writing them to the disk. The following unit provides an interface to buffers.

```

unit Buffer is
  spec var contents
  ...
  method buf.Extend(ch)
    modifies contents[buf]
    ensures contents[buf] = contents0[buf] ++ ch
  end

```

The writer implementation can now be written along the lines of

```

unit WrImpl imports Wr unqualified, Buffer unqualified is
  spec var flushed (* the flushed portion of the writer's target *)
  var b
  depends target[t] on flushed[t], b[t], contents[b[t]]
  rep target[t] is target[t] = flushed[t] ++ contents[b[t]]
  ...

  method wr.PutChar(ch) is
    if (* buffer full *) → (* flush buffer *) ☒ skip fi ;
    b[wr].Extend(ch)
  end

```

The problem of giving this specification was left open in [4]. Here, it works out because of the dynamic dependency in unit *WrImpl*. The dependency is allowed to be placed in *WrImpl*, despite that *target* is declared in *Wr* and *contents* in *Buffer*, provided that values of *b*[*t*] are not leaked. That condition is met by setting *b*[*t*] once for each writer (in some *Init* method not shown here). The field *b*[*t*] is then set to a newly allocated buffer, and no method in the writer interface gives away that buffer outside the implementation.

3.1 Sieve, revisited

We now offer some further comments on the prime sieve program shown in Section 0.3.

In unit *FilterImpl*, there are two dynamic dependencies of the form

```

depends a[t] on a[next[t]]

```

for *a* := *valid* and *a* := *state*. Consider for a moment what would happen if dynamic dependencies were not exempt, focusing on the *Init* method. Method *Init* is allowed to modify *valid*[*t*] and its downward closure,

but, if dynamic dependencies were not exempt, no part of its upward closure. This is too strict. We must allow $t.Init$ to modify $valid[u]$, for some u , in some cases. For example, consider an invocation of $u.Try(n)$, where $n \bmod value[u] \neq 0$ and $next[u]$ is initially nil . If the statement

$$next[u] := new()$$

sets $next[u]$ to t (assuming t was previously unallocated), then the program momentarily finds itself in a state where $valid[u]$ does not hold, because $next[u] \neq nil$ holds but $valid[next[u]]$ need not. The subsequent invocation of $next[u].Init(n)$ establishes $valid[next[u]]$, thus establishing, once again, $valid[u]$. Hence, the invocation of $Init$ on t did have an effect on its upward closure, *viz.*, changing $valid[u]$ from *false* to *true*. Luckily, dynamic dependencies *are* exempt, so the modification of $valid[u]$ by $t.Init$ is allowed.

The second alternative of the *if* in the *Try* method demonstrates how the treatment of newly allocated values comes into play. The method is allowed to modify the downward closure of $state[t]$, which when $next[t] = nil$ is essentially $state[t]$ and $next[t]$. The assignment to $next[t]$ is thus justified. However, the subsequent invocation of *Init* modifies, for example, $state[next[t]]$, which was not in the downward closure of $state[t]$ at the time the method was invoked. Luckily, this invocation modifies *state* at a newly allocated value, so it *is* allowed.

A story: A quest of a specification

The rest of the text for this example gives a fictional scenario of how a programmer might be guided by the compiler/checker toward writing an appropriate specification for methods *Init* and *Try*.

Let us start with the *Try* method. The checker will inform the programmer about the possibility of a division-by-zero error in the first alternative. The programmer then tries to add $value[t] \neq 0$ as a precondition of *Try*, and runs the checker again. This time, the checker complains that the recursive invocation of *Try* on $next[t]$ might not meet the precondition of $value[next[t]] \neq 0$.

The programmer thus realizes that *Try* can't be invoked on any t , but only on "valid" t 's. He thus introduces a specification variable *valid*, writes the appropriate **rep** clause, and updates the precondition of *Try* to require $valid[t]$. Realizing that *Query*, too, is supposed to only be called with valid filters, the programmer also adds $valid[t]$ as a precondition to *Query*.

If the programmer accidentally places the declaration of *valid* in *FilterImpl* instead of in *Filter*, he will soon realize that, because the compiler will emit an "unknown identifier" for the specification of *Try*. When the programmer runs the checker again, it will complain that the **rep** clause of $valid[t]$ mentions $value[t]$, $next[t]$, and $valid[next[t]]$, none of which $valid[t]$ depends on. The programmer remedies the situation by adding the appropriate **depends** clause.

Now, the checker complains again, because the second alternative in *Try* may violate the conjunct

$$next[t] \neq nil \Rightarrow valid[next[t]]$$

in *valid*'s representation invariant. That would make $valid[t]$ *false* and *Try* is not allowed to modify the value of $valid[t]$. The idea is, of course, that *Init* establish the validity of a filter. The programmer thus writes the specification of *Init*, saying that *Init* will modify $valid[t]$ to ensure that it holds upon exit.

The checker complains once more: *Try* modifies $next[t]$. This modification must be declared in the interface unit *Filter*. So far, there is only one variable in that unit, *viz.*, *valid*. There are two choices at this time. One is to change the specification of *Try* to say that it requires, modifies, and once again establishes $valid[t]$. With such a change in the specification, the programmer would be done. However, as he develops better taste for specifications, the programmer is more likely to introduce another specification variable like *state* and say that *Try* modifies $state[t]$. Then modification of $next[t]$ in *Try* hints that $state[t]$ should depend on $next[t]$, and the recursive

call in *Try* hints that $state[t]$ should also depend on $state[next[t]]$. This will cause the checker to complain that *Init* does not leave $state$ unchanged, upon which the programmer adds $state[t]$ to the frame of *Init*, too. Since the particular value of $state[t]$ does not appear in any postcondition, it is not necessary to furnish a **rep** clause for it.

3.2 Object containers

We were greatly inspired by the ideas of Cliff Jones [2] when thinking about leaking. Jones encountered problems similar to ours in the form of interference in concurrent programs. He introduces the concept of *unique* references (objects). A unique reference may get its value only from **new** and is not allowed to be copied. Albeit strict, these rules guarantee that the values of unique references are not leaked.

Not all of Jones's types can have unique references; in the spirit of the jargon in [3], we may say that a type can have unique references only if it is *unique-able*. A class is unique-able if none of its methods pass references "over" the object on which the method is invoked. That is, no in- or out-parameter may be a reference. Although there are programs one can write that satisfy this restriction, we feel there are many more programs for which the restriction is too stern. Hence, our requirement is weaker.

We now give an example of a program that is not allowed with Jones's rules, but that works nicely using our methodology. The problem is to provide a "container" class. You may think of a container as a set, a bag, a sequence, or something of that nature. For example, one can imagine a container for integers. More interesting is a container of objects. Provided are methods to insert and remove elements of the container. Such methods must take objects as parameters; hence, such a container class is not unique-able in Jones's formalism. However, since the container doesn't do anything with the given objects, other than storing them and later returning them, we think these element objects should be allowed to be leaked into and out of the container. Let's see how we do that.

Here's the interface to the container class. It will be sufficient for this example to restrict the container to keeping just one element at the time.

```

unit Container is
  spec var element
  method c.Set(el)
    modifies element[c]
    ensures element[c] = el
  method el := c.Query()
    ensures el = element[c]
end

```

The implementation looks as follows.

```

unit ContainerImpl imports Container unqualified is
  var elm
  depends element[c] on elm[c]
  rep element[c] is element[c] = elm[c]
  method c.Set(el) is elm[c] := el
  method el := c.Query() is el := elm[c]
end

```

That's all there is to it.

Soundness follows because the **depends** clause lives up to the visibility and authenticity requirements. Methodologically, the argument is that there is nothing the implementation could do with the given elements because there is only a static dependency from $elm[c]$. For example, if *Query* tried to invoke some method on a given element, as in

```
elm[c].m()
```

it can do so only if m requires nothing about the state of object on which it is invoked. If, for example, m is specified along the lines of

```
method e.m() requires good[e] ...
```

then *Query* would not be able to call it, because *Query* cannot show that *good* holds of $elm[c]$. In order for *Query* to do so, the precondition of *Query* must say something about the given element being good. It could do so by introducing a specification variable *valid* and letting $valid[c]$ depend on $good[elm[c]]$. But in order for this dynamic dependency to be placed near elm , the proviso regarding leaking values must be upheld, which it is not.

In the next example, we show what to do if the implementation does need to invoke a method like $elm[c].m()$ above.

3.3 Lexers and readers

The idea in this example is that a “lexer” reads characters from a “reader” to recognize lexical tokens, which it returns one by one. A reader is an input stream, declared in a manner similar to writers.

```
unit Rd is
  spec var good, source
  method rd.Open()
    modifies good[rd], source[rd]
    ensures good[rd]
  method rd.Close()
    requires good[rd]
    modifies good[rd], source[rd]
  method b := rd.AtEof()
    requires good[rd]
    ensures b  $\equiv$  source[rd] =  $\epsilon$ 
  method ch := rd.GetCh()
    requires good[rd]  $\wedge$  source[rd]  $\neq$   $\epsilon$ 
    modifies source[rd]
    ensures ch  $\#$  source[rd] = source0[rd]
end
```

Notice that the *Close* method modifies *good*, but doesn't ensure anything about its value after the call. Thus, clients can't assume anything about the value of *good* after invoking *Close*. In particular, clients cannot assume *good* holds, which effectively means that methods like *GetCh* cannot be invoked (unless *Init* is first invoked again to reestablish the goodness of the reader).

The initialization of a lexer takes a reader. The intention is that the lexer subsequently takes full control over that reader. That is, the reader is leaked into the lexer, but that's supposed to be okay because the lexer client isn't supposed to use the reader directly after that.

To specify the lexer, we need to be able to describe a lexer as being valid. The *Init* method will establish this validity and takes a good reader as a parameter. The *Next* method returns the next token, or returns **nil** if no prefix of the remaining input is a token. Method *Next* may thus modify the source of the reader, but does not alter the validity of the lexer. Hence, we need another specification variable to describe what *Next* modifies. Here is our straw man.

```

unit Lexer imports Rd unqualified is
  spec var valid, state

  method lex.Init(rd)
    requires rd  $\neq$  nil  $\wedge$  good[rd]
    modifies valid[lex], state[lex]
    ensures valid[lex]

  method token := lex.Next()
    requires valid[lex]
    modifies state[lex]
end

```

The implementation needs a field to store the reader of a lexer, call it *r*. A token may consist of one or more characters, so several *GetCh* operations may be required to obtain the next token. Here is the unit implementing lexers.

```

unit LexerImpl imports Lexer unqualified, Rd unqualified is
  var r
  (* dependencies and representation invariants to go here *)

  method lex.Init(rd) is
    r[lex] := rd

  method token := lex.Next() is
    var b, done in
      b, done :=  $\epsilon$ , false ;
      do  $\neg$ done  $\longrightarrow$ 
        if (* b is some token *)  $\longrightarrow$  token, done :=  $\dots$ , true
          ☒ rd.AtEof()  $\longrightarrow$  token, done := nil, true
          ☒ (* b is prefix of some token *)  $\longrightarrow$  b := b ++ rd.GetCh()
          ☒ token, done := nil, true
        fi
      od
    end
end

```

We still need to figure out what to do with dependencies and representation invariants (which is the interesting part anyway). We start with *valid*: A lexer is valid just when its reader is non-**nil** and good.

```

depends valid[lex] on r[lex], good[r[lex]]
rep valid[lex] is valid[lex]  $\equiv$  r[lex]  $\neq$  nil  $\wedge$  good[r[lex]]

```

So what about *state*? An inspection of what *Init* and *Next* modify leads us to the following **depends** clause.

```
depends state[lex] on b[lex], source[r[lex]]
```

As in the case of filters, *state* here is never said to take on any particular value, so it needs no **rep** clause, but only a **depends** clause.

With the above approximation, let's see if we follow the restrictions on the placement of **depends** clauses. The static dependencies—the dependencies on *r*[*lex*] and *b*[*lex*]—are fine. However, the dynamic ones are not—they stretch the visibility requirement, but the parameter of the *Init* method leaks into *r*[*lex*].

Before describing the solution to the specification problem, let's take a look at what can go wrong in the current case. Consider the following client.

```
unit LexerClient imports Lexer unqualified, Rd unqualified is
  method t.SomeMethod()
  method t.SomeMethod() is
    var rd, lex, token in
      rd := new() ; rd.Open() ;
      lex := new() ; lex.Init(rd) ;
      rd.Close() ;
      token := lex.Next()
    end
  end
```

Since the client closes the reader before the invocation of *Next*, the implementation of *Next* will end up invoking *AtEof* on a reader that is not guaranteed to be good. So, we would hope the lexer client would not be able to verify this code. The *Next* method requires only that the lexer be valid, *i.e.*, that *valid*[*lex*] holds. The validity of *lex* is established by the invocation of *Init*, so the question is whether or not the invocation of *Close* has any effect on *valid*[*lex*]. In the scope of *LexerClient*, the program checker knows that *Close* modifies *good*[*rd*] and *source*[*rd*], but does not know of any relation between these fields and *valid*[*lex*]. Hence, without enforcing the restrictions that go with the dynamic dependencies in the *LexerImpl* unit, the modular verification of *LexerClient* would be unsound.

Now for the solution. Since we cannot go around the fact that readers are leaked into *r*[*lex*], we must get rid of the dynamic dependencies that have *r* as their pivot. As another clue, we need to declare in the lexer interface that the validity of a lexer has something to do with the reader given to *Init*.

Picking up on the second of these clues, we introduce a new specification variable *rdr* in *Lexer*, and we declare that *valid* depends on it and its goodness.

```
spec var rdr
depends valid[lex] on rdr[lex], good[rdr[lex]]
```

The specification of *Init* is now extended with

```
ensures rdr[lex] = rd .
```

Similarly, the specification of *Next* is changed to

```
requires valid[lex] modifies state[lex], source[rdr[lex]] .
```

In the implementation unit, *LexerImpl*, we replace the two dynamic dependencies with the one static dependency and its associated **rep** clause

```
depends rdr[lex] on r[lex]
rep rdr[lex] is rdr[lex] = r[lex] .
```

Since *state* is thus no longer used, we remove it from the program.

Let us now reevaluate whether or not the implementation meets the specification and whether or not the rules governing the placement of **depends** clauses are satisfied. The first shall be last and the last shall be first.

We introduced one static and one dynamic dependency in the lexer interface. Both satisfy the visibility requirement, and the static one also satisfies the authenticity requirement. We removed the troubling dynamic dependencies from the implementation. The there newly introduced static dependency satisfies both visibility and authenticity. We conclude that the **depends** clauses are placed properly.

The *Init* method is allowed to modify the downward closure of *valid*[*lex*], which is

```
valid[lex], rdr[lex], good[rdr[lex]], r[lex] .
```

The body of the method modifies *r*[*lex*] so the modifies clause is satisfied. As before, *Init* establishes *valid*[*lex*], and it also establishes the new postcondition conjunct *rdr*[*lex*] = *rd* by setting *r*[*lex*] to *rd* (remember that the **rep** clause tells us *rdr*[*lex*] = *r*[*lex*]).

Method *Next* is allowed to modify *source*[*rdr*[*lex*]], which is its own downward closure. The implementation does live up to this. The invocation of *GetCh* requires that *r*[*lex*] be good, which is guaranteed by the fact that *valid*[*lex*] is required by *Next*.

That's it. Looking back at the faulty lexer client, we see that the checker will complain about that the precondition of *Next* cannot be guaranteed because *Close* modifies *good*[*rd*] on which *valid*[*lex*] depends. This dependency is known on account of the dependency

```
depends valid[lex] on good[rdr[lex]]
```

and *Init*'s postcondition

```
ensures rdr[lex] = rd .
```

Let us make one final observation. This example shows the common case where a client offers an object *x* (here, a reader) to become part of the state of another object, understanding that he gives away his right to some of *x*'s properties in the process.

In this case, the client gives up the right to modify *good* of the reader, something that becomes clear from the fact that *valid* depends on *good* of the given reader. However, the client does not give up the right to any other properties of readers. For example, clients are actually allowed to modify *source* of the reader after initializing a lexer with the reader. In this case, it is doubtful what benefit the client gets from that, since the lexer specification does not say anything about how it modifies *source*; hence, the client could find *source* in any state. Nevertheless, this scenario appears frequently in practice. For example, a parent window may acquire some child windows that it displays (see, e.g., Trestle [7]). Or, an "auto-flush writer" is a writer that periodically calls *flush* on an acquired writer (cf. [0]).

4 Conclusions

We have shown a programming methodology for writing modular, object-oriented programs. The methodology shows how a logical collection of methods can be specified abstractly. Although data abstraction has been a

familiar concept for over two decades, previous approaches assume, to varying degrees, that the implementation of an abstract entity is contained within one isolated scope. In real programs, this is seldomly the case. (This problem is known as “rep exposure” in CLU [6].) Our methodology allows for this, as we have demonstrated in numerous examples.

We do not yet know whether or not our methodology is sound with respect to modular verification. Nor have we explored yet whether or not one can write a mechanical program checker that enforces the restrictions that we have posed, even if the methodology *is* sound. We *have* shown an impressive collection of program examples that can be specified and verified using our methodology.

The big picture is, programmers have written modular programs for decades. Some programs have been correct, others have not. If it is sound, our methodology explains why the correct programs were indeed correct.

5 Acknowledgements

The problems dealt with here were pinpointed by the authors in the summer of 1994 at SRC and led to [4]. Since the end of that summer, what we now have given the name “dynamic dependencies” gave us much grief. This note shows the largest breakthrough since then, and although we haven’t proven our methodology sound, we feel we are much closer to the solution than we’ve ever been. We view the present proposal as a promising candidate solution.

The proposal for static dependencies is the solution in [4]. For dynamic dependencies, two events sparked some indispensable insights. One was sparked by a question made by Marcel van der Goot at Caltech when one of the authors was giving a practice talk. The insight led to the realization that the authenticity requirement is not strictly necessary—but only authentic dependencies can be what we now call “nonexempt”. The other spark came when one of the authors read Cliff Jones’s paper [2]. That paper describes some ideas for dealing with interference in the construction of concurrent programs in an object-oriented setting. Those ideas gave us several new ways to look at our problem, and we eventually weakened the requirements of [2] to something suitable for us (see Section 3.2). We are very grateful to Marcel van der Goot and Cliff Jones for their contributions.

References

- [0] M.R. Brown and G. Nelson. I/O streams: Abstract types, real programs. In G. Nelson, editor, *Systems Programming with Modula-3*, Series in Innovative Technology, pages 130–169. Prentice-Hall, Englewood Cliffs, NJ, 1991.
- [1] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.
- [2] C.B. Jones. Accommodating interference in object-based formal design. To appear.
- [3] K.R.M. Leino. Specifications and private data: A call for privatizable variables. KRML 43, Digital’s Systems Research Center, 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. Object-oriented guarded commands. KRML 50, Digital’s Systems Research Center, 1995.
- [6] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. The MIT Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA, 1986.

- [7] M.S. Manasee and G. Nelson. Trestle reference manual. Research Report 68, DEC SRC, 130 Lytton Ave., Palo Alto, CA 94301, U.S.A., December 1991.
- [8] B. Meyer. *Object-oriented Software Construction*. Series in Computer Science. Prentice-Hall International, New York, 1988.
- [9] G. Nelson. A generalization of Dijkstra's calculus. *ACM Transactions on Programming Languages and Systems*, 11(4):517–561, 1989.