Data abstraction and information hiding

K. RUSTAN M. LEINO and GREG NELSON
Compaq Systems Research Center

This paper describes an approach for verifying programs in the presence of data abstraction and information hiding, which are key features of modern programming languages with objects and modules. The paper draws on our experience building and using an automatic program checker, and focuses on the property of *modular soundness*: that is, the property that the separate verifications of the individual modules of a program suffice to ensure the correctness of the composite program. We found this desirable property surprisingly difficult to achieve. A key feature of our methodology for modular soundness is a new specification construct: the *abstraction dependency*, which reveals which concrete variables appear in the representation of a given abstract variable, without revealing the abstraction function itself. This paper discusses in detail two varieties of abstraction dependencies: static and dynamic. The paper also presents a new technical definition of modular soundness as a monotonicity property of verifiability with respect to scope and uses this technical definition to formally prove the modular soundness of a programming discipline for static dependencies.

Categories and Subject Descriptors: F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.2.4 [**Software Engineering**]: Program Verification; D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Modules, packages*

General Terms: Verification, Languages

Additional Key Words and Phrases: Abstraction dependencies, abstract variables, extended static checking, modifies clauses, modular verification, object-oriented programming, specifications

Contents

Authors' address: Compaq System Research Center, 130 Lytton Ave., Palo Alto, CA 94301, USA; email: rustan.leino@compaq.com and gnelson@compaq.com.

0. INTRODUCTION

This paper describes an approach for verifying programs in the presence of data abstraction, object types, and information hiding. The genesis of this work was the Extended Static Checking project (ESC) [Detlefs et al. 1998], which applies program verification technology to systems programs written in Modula-3. The aim of ESC is not to prove full functional correctness, but to prove the absence of common errors, such as array index errors, **nil** dereference errors, race conditions, deadlocks, etc.

One of the biggest problems we encountered in the ESC project is that the verification methodology we know from the literature does not seem to apply to the systems programs in the Modula-3 libraries. The problem is not that the programs use low-level tricks or unsafe code; the problem is that the programs use patterns of modularization and data abstraction that are richer than those treated in the verification literature. This is not an artifact of Modula-3, but would apply to any modern object-oriented language.

The data abstraction technology we know from the literature extends and refines the seminal paper on data abstraction by C.A.R. Hoare [1972]. In particular,

Hoare and all subsequent treatments that we know impose the requirement that all of the concrete variables used to represent an abstraction must be declared in the same module. This requirement is too strict: if it were applied to the Modula-3 libraries, many small modules would have to be combined, with a loss of desirable information-hiding. For example, this requirement would force the various subtypes of our example in Section 4, *readers* (buffered input streams), to be declared in one common module. Writing specifications is supposed to improve the structure of a program, so it is ironic that standard treatments of data abstraction are incompatible with good modularization. Therefore, in this paper we weaken Hoare's requirement and allow the concrete variables used to represent an abstraction to be divided among several modules.

A key technical challenge is to check modules where an abstract variable is visible, some of the concrete variables used to represent it are visible, but the representation function that connects them is not visible. To meet this challenge, we introduce a new specification construct called the *abstraction dependency*. This construct specifies that an abstraction connection exists between the variables, but does not specify the actual representation function, which can be confined to a more private scope. There are different types of dependencies, and these types produce a useful taxonomy of the patterns of abstraction in modular software.

Abstraction dependencies give the programmer considerable freedom in arranging the declarations of abstract variables, concrete variables, abstraction representation functions, and dependencies among the modules of a program. Too much freedom: without further restrictions, we would lose the property of modular soundness, that is, the property that the separate verifications of the individual modules of the program suffice to ensure the correctness of the composite program. We therefore impose several requirements, called *modularity requirements*, and argue that modular verification is sound for programs that meet the modularity requirements.

An essential aspect of information hiding in software design was enunciated in a classic paper of Parnas [1972], which we paraphrase as "every module hides a secret". To clarify the role of our dependencies, we distinguish two secrets involved in data abstraction: (a) the representation function and (b) the identity of the variables that are arguments to the representation function. We have found designs in which secret (b) should be less closely guarded (visible in more modules) than secret (a). The abstraction dependency makes it possible to achieve this.

1. ON THE NEED FOR DATA ABSTRACTION

Before we get into the details of our generalization, we set the stage by reviewing the role of data abstraction in modular verification.

To check that a large program does what it is supposed to do, we must study it piece by piece. Nobody's short-term memory is big enough to hold all the details of a large program. If the checking effort (formal or informal) is to be manageable, we cannot afford to re-examine the body of a procedure for every one of its calls. This is the reason for writing specifications, formal or informal. Given specifications, we check that each procedure meets its specification, assuming that the procedures it calls meet theirs.

This checking process is called modular verification, and for simple programming languages it has been understood since C.A.R. Hoare's work on axiomatic semantics

in the 1960s. (As long as the bulk of the verification is done modularly, we do not exclude simple whole-program checks, such as the check that each procedure is implemented somewhere in the program.) The central goal of this paper is to understand modular verification in the presence of two modern programming features: data abstraction and information hiding.

A procedure specification includes a precondition and a postcondition. The precondition is the part of the contract to be fulfilled by the caller of the procedure, and the postcondition is the part of the contract to be fulfilled by the procedure implementation. But precondition and postcondition are not enough: the specification also includes a "modifies list" that limits which variables the procedure is allowed to modify. Without the modifies list, the contract would allow a procedure to have arbitrary side effects on any variable not constrained by the postcondition, which would make the contract useless to the caller.

It is possible to view the modifies list as syntactic sugar for extra conjuncts in the postcondition, asserting that every variable not mentioned in the modifies list is unchanged. That is, in a program with three variables x, y, and z, the specification

$$\textbf{requires } P \textbf{ modifies } x \textbf{ ensures } Q$$

could be "desugared" into

$$\textbf{requires } P \textbf{ ensures } Q \ \wedge \ y = y' \ \wedge \ z = z'$$

in which primed variables denote post-values and unprimed variables denote pre-values. We cannot, however, use this desugaring to pretend that each procedure specification consists of a precondition and postcondition only. The reason is that, in modular verification, we never know, when verifying a procedure, what the set of all variables in the final program will be. Perhaps x, y, and z are the only variables visible where the procedure is declared, but more variables may be visible where the procedure is called. Therefore, in this paper we take the view that the modifies list is an integral part of the specification. Although we will rewrite modifies lists, the rewriting is different for different scopes.

Unfortunately, and perhaps surprisingly to those who have used verification more in principle than in practice, the methodology described so far is still inadequate. In many cases, it would be preposterous to try to list every piece of state that might be modified by a call to a procedure. For example, what would be the list for the `putchar` procedure from the C standard I/O library? What `putchar` does is simply write a character to output, but anybody who has implemented an I/O system will be aware that the list of what can be modified during the execution of a call to `putchar` is very long. It includes, for example, the I/O buffers, the internal state of the device drivers for the disk and network, the device registers in these drivers, and the disk and network themselves. The minor problem is that this list is long; the major problem is that the variables in the list are not visible at the point of declaration of `putchar`, and to make them visible would be to give up on information hiding, which would be to resign the game before it starts.

The solution to this difficulty —at least the only solution that we can imagine— is *data abstraction*: the use in specifications of *abstract* variables whose values are not directly manipulated by the compiled program but are instead represented as functions of other variables, abstract or concrete. Abstractly, `putchar` modifies a

single abstract variable, of a simple type (say, sequence of byte). All the internal state, from buffers to devices, must be treated as concrete state that is part of the representation of the abstract state.

Some people see data abstraction as an algorithm design methodology only, as a methodology for deriving an efficient algorithm from a simple algorithm by changing the representation of the state. We have no quarrel with their use of data abstraction, but our point is that data abstraction is also an essential ingredient in any scheme for modular verification of large systems, since it seems to be the only hope for writing a useful modifies list for a procedure whose implementation changes the system state at many levels of abstraction.

Having identified the general idea of the solution to the `putchar` problem as data abstraction, we would add that the patterns of data abstraction that arise in verifying `putchar` are beyond the current state of the art of specification: we believe that no semantics or methodology presented in the literature is equal to the task. We feel that by defining the notion of an abstraction dependency and providing a sound methodology for static dependencies, this paper takes two important steps toward the goal of reasoning about such programs. But our lack of a soundness theorem for dynamic dependencies, and our immature treatments of cyclic dependencies, array dependencies, and modular invariants, make it clear that much methodology remains to be invented before the goal is within reach.

2. VALIDITY AS AN ABSTRACT VARIABLE

The generalized data abstraction described in this paper is relevant regardless of whether verification is being used for full functional correctness or for more limited aims, such as the ESC aim of verifying the absence of certain classes of errors only. The examples in this paper will be ESC verifications. These verifications tend to have a typical form, which is described in this section.

In a typical ESC verification, we associate two abstract variables with each type, *valid* and *state*. The first of these records whether objects of the type satisfy the internal representation invariant required by the implementation, and the second represents the abstract value of variables of the type.

If we were verifying full functional correctness, we would have to write many specifications about the *state* variable. But in doing extended static checking, we rarely say anything about the state. We aren't proving that the program meets its full functional specification, only that it doesn't crash. The main purpose of the *state* variable is to account for the side effects in the implementations of the methods, which otherwise would lead to spurious errors reported by the verifier. Indeed, in many ESC verifications, we don't even bother to provide the concrete representation for *state*.

In contrast to *state*, the checking performed by ESC depends critically on *valid*. Most operations on an object o will have $valid[o]$ as a precondition. The checker uses the concrete representation for *valid* to translate $valid[o]$ into a concrete precondition, which it then uses in proving that the implementation of the operation does not cause an error.

In Hoare's original paper on data abstraction, the notion of a validity invariant was built into the methodology. Initialization was required to establish validity and all other operations were required to preserve it. In contrast, we consider *valid* to

be an abstract variable like any other; the programmer explicitly provides *valid* as a precondition (and/or postcondition), and the implementation infers the details of validity in terms of the concrete state via the usual process of data abstraction. Our approach has several advantages over Hoare's, of which we mention one: we allow operations like closing a file, which destroy validity. Such operations are frequently essential in order to deallocate resources.

3. DEFINITION OF NOTATION

This section introduces the notation and terminology of the formal system that we use for modeling programs in the rest of the paper.

Modularity. A *program* is a collection of *declarations*. Declarations introduce names for entities (such as types, abstract and concrete variables, and methods) and/or specify properties of named entities (such as subtype relationships, representations of abstract variables, method specifications, and method implementations). The declarations of a program are partitioned into *units* (sometimes called interfaces and modules). The declarations in effect in a unit are its own declarations and the declarations in effect in units that it *imports*. If an entity E is declared in a unit M, it is known as $M.E$ in importers of M and known simply as E within M. For example:

unit M	**unit** N **import** M
type T	... uses of $M.T$ ...
... uses of T ...	

In this paper, we sometimes write E instead of $M.E$ when M is clear from the context.

One of the purposes —perhaps the main purpose— of module systems is to reduce the portion of a program that must be read and potentially fixed to accommodate a change, addition, or deletion of a declaration. With our very general system of units, the portion of the program sensitive to a declaration in a unit M is exactly the set of units that import M directly or indirectly.

A set of units D is called a *scope* if it is closed under imports, that is, if whenever a unit M in D imports a unit N, then N is also in D. A declaration is *visible* in a scope if it appears in one of the units in the scope.

We use units and imports in this paper since they are simple and extremely general. Restrictive patterns are common in practice. For example, Modula-3 requires that every unit be an interface unit, which can declare procedures and methods but not declare implementations, or an implementation unit, which can declare implementations but which cannot be imported (and therefore has the property that no other portion of the program can be sensitive to it). As another example, CLU imposes a correspondence between units and type declarations. We have not imposed such restrictions in this paper, because they seem orthogonal to the modularity issues that we are discussing.

Our units provide all of the functionality of accessibility modifiers like `private` and `public` in languages like C++ and Java. By way of illustration, here is a tiny Java class together with a possible translation into our notation (the translation

uses some notation that will be defined below):

```
class T {
  private int x;
  public void m(int y) {
      C
    }
}
```

```
unit Public
  type T
  proc m(this: T, y: int)
unit Private import Public
  var x: T → int
  impl m(this: T, y: int) is
      C
    end
```

An advantage of the units version is that the *Public* unit has no need to import any units that are required by m's implementation only.

Types. In this paper, we will use primitive types like **int** and **bool**, as well as object types and array types. Our objects are like those of Simula, Modula-3, and Java (excluding interface types): they are implicitly references, and each object type has a uniquely determined direct supertype. More precisely, an *object* is either **nil** or a reference to a set of data fields and methods; a method is a procedure that will accept the object as its first parameter. Equality of objects is reference equality. An *object type* determines the names and types of a prefix of the fields and the names and signatures of a prefix of the methods of its objects.

An object type T is declared

$$\textbf{type } T <: S$$

where S is an object type declared elsewhere. This introduces the name T for a new object type whose direct supertype is S, meaning that T contains all the fields and methods of S and possibly includes other fields and methods declared elsewhere. The "$<: S$" is optional; if omitted, S defaults to an anonymous object type serving as the root of the subtype hierarchy.

Every object has a *dynamic type* determined when it is allocated. Every expression has a *static type* determined at compile time. If v is the dynamic value of an expression E, v has dynamic type D, and E has static type S, then conventional static type-checking rules assure that D is a subtype of S.

We consider a data field, abstract or concrete, to be a map from objects to values. Thus, where others write

$$\textbf{class } T \{ \ \ldots \ \textbf{int } f; \ \ldots \ \}$$

we write

$$\begin{array}{l}\textbf{type } T\\ \textbf{var } f\colon T \to \textbf{int}\end{array}$$

(Figure 0 shows a more realistic example.) Also, we write $f[t]$ where others write $t.f$ to denote the value of the f field of object t. We refer to T and **int** as the *index type* and *range type* of f, respectively. The **class** notation forces f to be co-declared with T, whereas our notation allows them to be declared independently. This generality is not problematical; in fact, it simplifies the semantics. We call the variables (like f) that represent fields "maps". They might also be called "functions", but we prefer a name that emphasizes that they are values in a first-order theory.

If T is a type, we write

$$\mathbf{array}[T]$$

to denote the type of (references to) arrays with element type T. If a is of type $\mathbf{array}[T]$ and is non-**nil**, then $\mathbf{number}(a)$ denotes the number of elements in a, and $a[i]$ denotes element i of a for $0 \leq i < \mathbf{number}(a)$. To properly model the fact that arrays are references, we introduce the predeclared map variable *elems*: the expression $elems[a]$ denotes the sequence of elements referred to by an array a. For example, $a = b$ means that a and b reference the same sequence, while $elems[a] = elems[b]$ means that the sequences referenced have the same elements. In fact, $a[i]$ is shorthand for $elems[a][i]$.

If T is an object type, $\mathbf{new}(T)$ allocates and returns a new object of dynamic type T. For any type T, $\mathbf{new}(T, n)$ allocates and returns a new array of dynamic type $\mathbf{array}[T]$ and of length n.

A method m for type T is declared and specified as follows:

$$\begin{array}{l}\mathbf{proc}\ m(t\colon T, \ldots\ args\ \ldots)\colon R\\ \quad \mathbf{requires}\ P\\ \quad \mathbf{modifies}\ w\\ \quad \mathbf{ensures}\ Q\end{array}$$

where in the *signature* "$(t\colon T, \ldots\ args\ \ldots)\colon R$", T is an object type, t is the self parameter, *args* lists the names and types of any additional parameters, and R is the result type. In this paper, all parameters are in-parameters. In addition to declaring the name and signature of the method, the declaration associates with it the precondition P, postcondition Q, and modifies list w. A program can contain at most one declaration for a given method for a given type; for example, we don't allow strengthening a method specification in a subtype (this is a simplification that does not actually limit expressiveness, see p. 348 of [Leino 1998b]). In the postcondition, **result** denotes the result value, primed variables denote values in the post-state, and unprimed variables denote values in the pre-state. If the precondition or postcondition is omitted, it defaults to *true*; if the modifies list is omitted, it defaults to the empty list.

A method m for type T can be implemented differently for each subtype of T. A method implementation of m for some subtype U of T is declared by

$$\mathbf{impl}\ m(u\colon U, \ldots\ args\ \ldots)\colon R\ \mathbf{is}\ S\ \mathbf{end}$$

where S is an executable statement, and the implementation signature

$$(u\colon U, \ldots\ args\ \ldots)\colon R$$

coincides with the declared signature except (possibly) for the type of the first parameter. Statement S must satisfy (that is, the verifier checks that it satisfies) the specification associated with the m method for T. The ideas in this paper don't depend on the particular executable statements allowed. The examples in this paper use Algol-like executable statements, whose meaning we hope will be clear to the reader.

A method is called by

$$t.m(\ldots\ args\ \ldots)$$

```
type Rat
spec var valid: Rat → bool
type Ratio <: Rat
var num, den: Ratio → int
rep valid[r: Ratio] ≡ den[r] > 0
type CFrac <: Rat
var parquo: CFrac → array[int]
rep valid[cf: CFrac] ≡
    parquo[cf] ≠ nil ∧
    ⟨ ∀ i :: 1 ≤ i < number(parquo[cf]) ⇒ parquo[cf][i] > 0 ⟩
```

Fig. 0. An example program, illustrating that representation of an abstract variable can be subtype-specific.

where t is an object (the actual self parameter), m is a method name, and *args* is a list of any additional actual parameters. It would be more logical to write $m(t, \ldots)$ instead of $t.m(\ldots)$, but whereas we chose to be logical ($f[t]$) rather than conventional ($t.f$) for field accesses, we choose to be conventional rather than logical for method calls. The static type of t is used in determining the declaration and specification of m. The declaration is used to type-check the actual parameters and determine the static type of the result, the specification is used to reason about the semantics of the call. The dynamic type of t is used at run-time to determine which implementation of m to invoke. Since all method implementations are proved to meet their specifications, and since the dynamic type of t is a subtype of the static type of t, it is sound to reason about the semantics of the dynamic dispatch in this way.

Abstraction. A data field can be declared to be *abstract* by preceding its declaration with **spec**. For example:

$$\textbf{spec var } \mathit{valid}\colon T \to \textbf{bool}$$

An abstract field occupies no memory at run-time; it is a fictitious field whose value (or *representation*) is defined as a function of other fields. An abstract variable cannot appear in ordinary program text; it appears only in specifications. The representation is declared by a syntax like

$$\textbf{rep } \mathit{valid}[t\colon T] \equiv f[t] \neq 0$$

which means that for any non-**nil** object t of type T, the abstract value of $\mathit{valid}[t]$ is *true* if and only if $f[t] \neq 0$.

The representation of an abstract variable can be different for different subtypes. As an example, consider the object type *Rat* representing rational numbers, and two of its subtypes, *Ratio*, which represents each rational as a ratio, and *CFrac*, which represents each rational as a continued fraction (which is a representation of a rational as a sequence of integers), see Figure 0. These declarations specify that the concrete representation of $\mathit{valid}[q]$ varies depending on the dynamic type of q: for rationals represented as ratios, validity means that the denominator is positive, whereas for continued fractions, validity means that each partial quotient is positive, except possibly the first.

A **rep** declaration given at a type T applies to all non-**nil** objects of type T, including those whose dynamic type is a subtype of T. One might think that it would be possible to override a **rep** declaration at T with another **rep** declaration at some subtype of T, but this is not allowed, since it would be unsound. Checking compliance with this rule is a whole-program check, but it is simple.

The variables appearing in the right-hand side of the **rep** declaration for an abstract variable are called *dependencies* of the abstract variable. The dependencies can themselves be either concrete or abstract. To avoid any possible confusion, we state that our abstraction dependencies are not related to use-def dependencies [Aho et al. 1986].

A major novelty of our approach is to require that dependencies be declared explicitly. For example, standing alone, the representation

$$\textbf{rep}\ \mathit{valid}[t\colon T] \equiv f[t] \neq 0$$

is forbidden on the grounds that it contains an undeclared dependency. Allowing the representation requires an explicit dependency declaration of the form

$$\textbf{depends}\ \mathit{valid}[t\colon T]\ \textbf{on}\ f[t]$$

In this paper, we sometimes omit the "$\colon T$" when T is obvious or unimportant. The **depends** declaration can be subtype-specific, just like the **rep** declaration. For example, the representations in Figure 0 might be accompanied by

$$\textbf{depends}\ \mathit{valid}[r\colon \mathit{Ratio}]\ \textbf{on}\ \mathit{den}[r]$$
$$\textbf{depends}\ \mathit{valid}[\mathit{cf}\colon \mathit{CFrac}]\ \textbf{on}\ \mathit{parquo}[\mathit{cf}], \mathit{elems}[\mathit{parquo}[\mathit{cf}]]$$

The validity of the continued fraction cf depends both on the array $parquo[cf]$ and on the contents of the array. These are different dependencies and both must be declared, as shown above. The validity of the ratio r depends only on $den[r]$.

This paper is principally concerned with two forms of dependencies, static and dynamic. A *static* dependency has the form

$$\textbf{depends}\ a[t\colon T]\ \textbf{on}\ c[t] \tag{0}$$

A *dynamic* dependency has the form

$$\textbf{depends}\ a[t\colon T]\ \textbf{on}\ c[b[t]] \tag{1}$$

For the static dependency (0), two variables $a[t]$ and $c[x]$ are connected if $x = t$, which can be determined statically; for the dynamic dependency (1), $a[t]$ and $c[x]$ are connected if $x = b[t]$, a condition that involves the dynamic value of $b[t]$. In each case, a is an abstract variable and c is either an abstract or a concrete variable. In the case of the dynamic dependency, b is concrete. A dependency on the contents of an array counts as a dynamic dependency, with *elems* playing the role of c. Other forms of dependencies will be discussed in Section 9.1, but static and dynamic dependencies are more common and fundamental.

A major goal of this paper is to design a discipline for the placement of dependency declarations in a multi-module program. The paper is long, but the main conclusion is short: the static dependency (0) must be visible wherever c is, and the dynamic dependency (1) must be visible wherever b is.

Dependencies affect the verification process in several ways. One way is *modifies list desugaring*. For example, in a scope where

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

is visible, the modifies list

modifies $a[t]$

is desugared into

modifies $a[t], c[t]$

This reflects the common-sense view that the license to modify an abstract variable implies the license to modify its representation. The precise details of modifies list desugaring will be described later in the paper.

4. EXAMPLE: READERS

From our experience with ESC, we have found that dependencies are not just a detail but a key ingredient of the specification language that we used constantly. However, since dependencies are a tool for programming in the large, no small example does them full justice. This section presents the smallest example we know that motivates the essential points: a simplified version of *readers*, which are the object-oriented buffered input streams used in the standard I/O library of Modula-3. A key point that the example will illustrate is that modern information hiding together with subtyping creates situations where both an abstract variable and one or more of its dependencies are visible, but the associated representation is not visible. In these situations, sound modular verification would be impossible, but dependencies save the day.

Readers (and their output counterparts, *writers*) were invented by Stoy and Strachey [1972] for the OS6 operating system. Although Stoy and Strachey never used the word "object" or "class" in describing them, they are in fact one of the most compelling examples of the engineering utility of object-oriented programming. Each reader is an object with a buffer and a method for refilling the buffer. Different subtypes of readers override the refill method with code appropriate to that type of reader; for example, a disk reader fills the buffer from the disk, a network reader from the network.

As part of the ESC project, we have mechanically verified the absence of errors from most of the Modula-3 standard I/O library, including all the standard reader subtypes. In this paper we want to focus on generalized data abstraction, and many of the complexities of the actual I/O system would distract us from this focus, so we will simplify the reader interface rather drastically. (The actual code and specifications that we have used as input to the Extended Static Checker can be found on the web [Extended Static Checking for Modula-3].)

Our simplified interface *Rd* declares the type T representing a reader, and specifies the two methods *getChar* and *close*, see Figure 1. Since our examples show ESC verifications only, we specify the range type of *state* as **any**, and we ignore the effects on *state* in the **ensures** clauses. We use the convention that $rd.getChar()$ returns -1 when rd is exhausted, and otherwise returns the next byte of input. The specification of *close* reflects the design decision that a reader can be closed

```
unit Rd
  type T
  spec var valid: T → bool
  spec var state: T → any
  proc getChar(rd: T): int
    requires valid[rd]
    modifies state[rd]
    ensures −1 ≤ result < 256
  proc close(rd: T)
    requires valid[rd]
    modifies valid[rd], state[rd]
```

Fig. 1. The interface *Rd*, which declares type *T* representing readers.

```
unit RdRep import Rd
  var lo, cur, hi: Rd.T → int
  var buff: Rd.T → array[byte]
  spec var svalid: Rd.T → bool
  rep valid[rd: Rd.T] ≡
    0 ≤ lo[rd] ≤ cur[rd] ≤ hi[rd] ∧
    buff[rd] ≠ nil  ∧  hi[rd] − lo[rd] ≤ number(buff[rd]) ∧
    svalid[rd]
  proc refill(rd: Rd.T)
    requires valid[rd]
    modifies state[rd]
    ensures cur[rd] = cur′[rd]
  depends valid[rd: Rd.T] on lo[rd], cur[rd], hi[rd], buff[rd], svalid[rd]
  depends state[rd: Rd.T] on lo[rd], cur[rd], hi[rd], buff[rd],
                             elems[buff[rd]]
  depends svalid[rd: Rd.T] on lo[rd], hi[rd], buff[rd]
```

Fig. 2. The interface *RdRep*, which defines the buffer structure common to all objects of type *Rd.T*.

only once (a second call to *close* requires validity, which may have been destroyed by the first call).

We call attention to the absence of *valid*[*rd*] from the modifies list of *getChar*. In our system, this specifies that calls to *getChar* preserve validity. This specification is enforced even if *state* and *valid* are represented in terms of the same data fields.

Next we describe the unit that defines the generic buffer structure (by generic, we mean common to all readers, as opposed to subtype-specific), see Figure 2. The integer *cur*[*rd*] is the index in the abstract stream *rd* of the next byte to be returned by *getChar*. The integers *lo*[*rd*] and *hi*[*rd*] delimit the range of bytes in the abstract stream that are contained in the buffer *buff*[*rd*] (see Figure 3).

Interface *RdRep* declares and specifies the *refill* method, but leaves its implementation to various subtypes. The convention used by *refill* is that the call *rd.refill*() must make at least one new byte available (that is, it must establish *cur*[*rd*] < *hi*[*rd*]), unless *rd* is exhausted, in which case it must establish the condition *cur*[*rd*] = *hi*[*rd*].

The postconditions of *getChar* and *refill* don't reflect the conventions for signal-

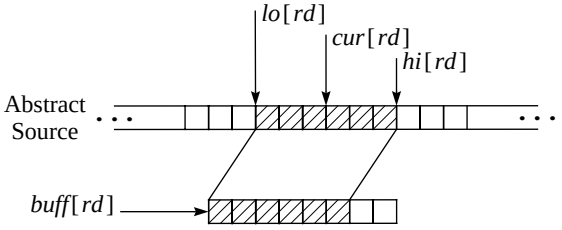


Fig. 3. Buffer representation of readers.

ing that the reader is exhausted (nor does the variable *state* model the condition that the reader is exhausted), because our example is an ESC verification, not a verification of full functional correctness.

The **rep** declaration reveals the representation of the abstract variable *valid* in terms of the concrete variables *lo*, *cur*, *hi*, and *buff*. In addition, because subtypes may have their own validity invariants, the interface declares the abstract variable *svalid*, and adds the conjunct *svalid*[*rd*] to the representation of *valid*[*rd*]. The intended meaning of *svalid*[*rd*] is that *rd* satisfies the validity invariant of its dynamic type. Each subtype of $Rd.T$ will include a **rep** declaration specifying the representation of *svalid* for readers of that subtype. For example, a reader for a disk file would include a file handle as one of its fields, and its *svalid* would include the validity of the file handle.

The **depends** declaration for *valid* is explained by our requirement that dependencies be explicit—without it, the methodology would forbid the displayed **rep** declaration on the grounds that *valid* contains undeclared dependencies. The **depends** declarations for *state* and *svalid* are more subtle and will be explained later.

The generic implementation includes an implementation of *getChar*, shown in Figure 4. The actual Modula-3 readers package includes many routines in the generic implementation that are identical to *getChar* from the point of view of modularity and information hiding: routines to read a line, to read a decimal numeral, integral or floating point, and so on. All of these generic routines are available to any subclass implementor "for free". We find this a very attractive feature of the readers design, although some people contend that all of these operations should be implemented by every subtype rather than by generic code. In any case, we think that even those who would have designed readers differently would agree that the present design is credible enough that a programming methodology should be able to handle it.

To give the flavor of an ESC verification, consider checking that $cur[rd] - lo[rd]$ is a valid index into *buff*[*rd*] in the implementation of *getChar*. Since *valid*[*rd*] is a precondition of *getChar*, and is specified to be preserved by *rd.refill*(), we conclude that *valid*[*rd*] holds at the first semicolon. Thus, the validity of the index boils down to showing that

$$0 \leq cur[rd] - lo[rd] \ \wedge \ cur[rd] - lo[rd] < \mathbf{number}(buff[rd]) \qquad (2)$$

```
unit RdImpl import Rd, RdRep
  impl getChar(rd: Rd.T): int is
      if cur[rd] = hi[rd] then rd.refill() end ;
      if cur[rd] = hi[rd] then
        result := −1
      else
        result := buff[rd][cur[rd] − lo[rd]] ;
        cur[rd] := cur[rd] + 1
      end
    end
```

Fig. 4. The implementation unit *RdImpl*, which contains the implementation of the method *getChar*.

follows from

$$valid[rd] \ \wedge \ cur[rd] \neq hi[rd] \tag{3}$$

Since *RdImpl* imports *RdRep*, the representation of $valid[rd]$ is visible. Since this representation contains the conjunct $lo[rd] \leq cur[rd]$, the first conjunct of (2) follows immediately. The proof of the second conjunct is:

$$\begin{array}{ll} & cur[rd] - lo[rd] \\ < & \quad \{ \ \ cur[rd] \leq hi[rd] \ \wedge \ cur[rd] \neq hi[rd] \text{ (from (3))} \ \ \} \\ & hi[rd] - lo[rd] \\ \leq & \quad \{ \ \ valid[rd] \ \ \} \\ & \textbf{number}(buff[rd]) \end{array}$$

Returning to general comments about *rd.getChar*(), notice that the implementation modifies $cur[rd]$, but the **modifies** clause in the specification of the method *getChar* does not mention $cur[rd]$. Why does the methodology allow this? Because of modifies list desugaring, as mentioned in the previous section. Modifies list desugaring gives *getChar* the license to modify $cur[rd]$, because *getChar* is specified to modify $state[rd]$, which is declared in *RdRep* to depend on $cur[rd]$. This explains why $cur[rd]$ was declared a dependency of $state[rd]$.

Having written *RdRep* and *RdImpl*, we have specified a design for the input streams, but please note that our implementation has barely begun, since we have not yet implemented a single *refill* method. The design allows the rest of the implementation to be structured as a collection of subtypes of $Rd.T$, each of which implements its own *refill* (and *close*) method. The design also allows the private declarations of these subtypes to be hidden in separate units. To illustrate the modularity issues that arise when a subtype is defined, we now give the interface (Figure 5) and implementation (Figure 6) of a trivial type of reader, a *blank reader*, which delivers a sequence of blanks whose length is determined at initialization time. More precisely, the expression $\textbf{new}(BlankRd.T).init(n)$ allocates, initializes, and returns a reader that delivers a stream of exactly n blanks. The conjunction "**result** = *brd*" in the postcondition specifies that the *init* method returns the object that it initializes, a convention we have found useful. The implementation of the method stores the argument n in the field $num[brd]$ for later use by the method *refill*. The method also initializes the *lo*, *cur*, and *hi* fields in the obvious way,

```
unit BlankRd import Rd
  type T <: Rd.T
  proc init(brd: T, n: int): T
    requires 0 ≤ n
    modifies valid[brd], state[brd]
    ensures valid'[brd] ∧ result = brd
```

Fig. 5. Unit *BlankRd* declares a subtype *BlankRd.T* of *Rd.T*, whose readers deliver streams of blanks.

```
unit BlankRdImpl import Rd, RdRep, BlankRd
  var num: BlankRd.T → int
  rep svalid[brd: BlankRd.T] ≡ hi[brd] ≤ num[brd]
  impl init(brd: BlankRd.T, n: int): BlankRd.T is
      num[brd] := n ;
      buff[brd] := new(byte, min(8192, n)) ;
      lo[brd] := 0 ; cur[brd] := 0 ;
      hi[brd] := number(buff[brd]) ;
      for i := 0 to hi[brd] − 1 do
        buff[brd][i] := 32
      end ;
      result := brd
    end
  impl refill(brd: BlankRd.T) is
      lo[brd] := cur[brd] ;
      hi[brd] := min(lo[brd] + number(buff[brd]), num[brd])
    end
  depends state[brd: BlankRd.T] on num[brd]
  depends svalid[brd: BlankRd.T] on num[brd]
```

Fig. 6. The blank reader implementation unit *BlankRdImpl*.

allocates a buffer of size up to 8192 (that is, up to 8 kilobytes), and fills the buffer with blanks (code 32). The implementation unit for blank readers declares *num* to be a further dependency of *state* and *svalid*

As we shall see later, it is critical to the verification of the module that each blank reader *brd* satisfy the invariant $hi[brd] \leq num[brd]$. Therefore, the implementation unit *BlankRdImpl* provides a subtype-specific representation for *svalid*, effectively strengthening the general reader validity invariant as needed for the particular subtype *BlankRd.T*.

Recall that the *refill* method is specified to preserve validity. Each *refill* implementation must therefore be proved to maintain validity, so we must prove that *brd.refill*() does not change $valid[brd]$. Sometimes proof obligations of this form can be discharged simply by observing that no operation in the method body has any effect on the variable that must be preserved. But in the present case, this simple approach doesn't suffice, since the *refill* implementation modifies the representation of *valid*. Instead we must use the **rep** declaration and prove the conjuncts of $valid[brd]$ one by one. Among them is that at exit,

$$cur'[brd] \leq hi'[brd]$$

Since the body of *refill* does not change $cur[brd]$ and makes $hi'[brd]$ equal to

$$\mathbf{min}(cur[brd] + \mathbf{number}(buff[brd]), num[brd])$$

proving this postcondition boils down to showing that each argument to **min** is at least $cur[brd]$. For the first argument, this follows from the fact that the **number** of any array is non-negative. The proof for the second argument is:

$valid[brd]$
$\Rightarrow$ { **rep** for *valid* }
$cur[brd] \leq hi[brd] \ \wedge \ svalid[brd]$
$=$ { brd is of type $BlankRd.T$, **rep** for *svalid* for this type }
$cur[brd] \leq hi[brd] \ \wedge \ hi[brd] \leq num[brd]$
$\Rightarrow$ { transitivity }
$cur[brd] \leq num[brd]$

We present this calculation in detail to illustrate that the verification of the *refill* method of even the trivial $BlankRd.T$ requires the subtype-specific validity conjunct. (The need for the *svalid* conjunct is more conspicuous in more interesting reader subtypes.)

Read-only by specification. The calculation that $cur[brd] \leq num[brd]$ follows from $valid[brd]$ would be in vain if the generic code could modify $hi[brd]$. If, for example, the generic implementation of $rd.getChar()$ would sometimes increment $hi[rd]$, then it could destroy $svalid[rd]$, which could cause all kinds of errors. To prevent this, the Modula-3 interface from which we translated *RdRep* contains the following English comment:

> The generic code modifies $cur[rd]$, but not $lo[rd]$, $hi[rd]$, or $buff[rd]$. (4)

This guarantee is essential to subtypes, since between calls to their *refill* methods, they may need to know that *lo*, *hi*, and *buff* have not been changed by the generic code.

How do we translate the sentence (4) into a formal specification? Modula-3 does not have any kind of **readonly** qualifier for field declarations. Java and C++ have qualifiers like `private` and `protected` that limit access to some field to the type where it is declared (or to that type and its subtypes), but neither of them (not even C++) has a qualifier like

```
readonlytothistypewritabletosubtypes
```

which is the sort of qualifier that would actually be needed in this case. Clearly we cannot expect a programming language to have a qualifier that enforces this highly particular access policy, but modular verification won't be sound unless the policy is formally stated and enforced.

We wrestled with the problem for some time before realizing happily that abstraction dependencies provide a neat solution. In fact, the third dependency declaration in *RdRep*, which states that $svalid[rd]$ depends on $lo[rd]$, $hi[rd]$, and $buff[rd]$, is the desired formalization of (4). For if the generic code were to modify any of these fields, the presence of the dependency would imply that *svalid*, and therefore *valid*, might be changed. In other words, in a scope where $lo[rd]$, $hi[rd]$, and $buff[rd]$ are known to be part of the representation of $svalid[rd]$, but the explicit representation

is unknown, the only hope for maintaining *svalid*[*rd*] invariant is to avoid modifying *lo*[*rd*], *hi*[*rd*], and *buff*[*rd*]. We call this technique "read-only by specification".

Summary. To repeat our main conclusion from this example, we see that modular verification with subtyping creates situations where both an abstract variable and one or more of its dependencies are visible, but where the associated representation is not visible. We have seen two instances of this:

— The dependencies of *state* are specified in *RdRep*, but no representation for *state* is specified. The dependencies must be visible so that the implementations of operations that modify the state (for example, *getChar*) will have the license to modify the concrete variables that represent the state. The representation declaration cannot be visible, for two reasons. First, because we are doing extended static checking only, we never give a representation for the state. Second, even if we were doing full-scale verification, the representation would be subtype-specific, but the dependencies must be visible in the generic scope.

— The dependencies of *svalid* are specified in *RdRep*, but no representation for *svalid* is given there. The dependencies are necessary to prevent generic operations from modifying the variables that are reserved for the use of subtypes. But it is clearly impossible to present a representation declaration for *svalid* in the *RdRep* scope, since the whole point of *svalid* is to allow subtypes to include their own invariants as part of validity: these invariants can't be known in the generic scope.

Explicit dependencies may seem verbose, and it would be nice to be able to infer them automatically. But this will not always be possible. For example, of the three **depends** declarations in *RdRep*, we can imagine inferring the first (from the **rep** declaration for *valid*), but the second and third could not be automatically inferred since they are used to specify non-trivial design decisions (namely, which fields can be modified by generic code, and which can be modified by subtypes only).

This concludes our example of the role of dependencies in modular verification. In the remainder of the paper, we investigate different kinds of dependencies and the way they affect the verification process.

5. STATIC DEPENDENCIES

In this section, we describe more fully how dependencies affect the verification process. Our guiding principles are:

— *Abstraction function principle.* An abstract variable is a function of the concrete variables on which it depends.

— *Abstraction modification principle.* The license to modify an abstract variable implies the license to modify its concrete representation, but the license to modify a concrete variable does not imply the license to modify an abstract variable that depends on it.

Our technique is to rewrite preconditions, postconditions, modifies lists, and **rep** declarations into equivalent forms that contain concrete variables only. In this section, we will describe the rewriting steps and explain how they follow from the principles.

We confine ourselves to static dependencies for simplicity; in Section 7, we will extend this material to dynamic dependencies.

We feel it only fair to warn the reader that this section on dependencies and the next section on soundness of modular verification are rather technical. They record the design decisions that we took in order to build a checker that handles realistic programs. We thought it appropriate to record this design in enough detail that others could replicate our results, should they desire to. An alternative approach, that of Müller and Poetzsch-Heffter [Müller 2001], avoids many of the technicalities of these two sections, but the essential methodological requirements that our approach uncovers emerge in their approach as well, and the degree to which their approach admits automation is an open question.

5.0 Functionalization

Guided by the abstraction function principle, and following in the footsteps of Hoare, we introduce a new function symbol for each abstract variable. In this paper, we write $\mathcal{F}.a$ to denote the function symbol introduced for the abstract variable a. The idea is that $\mathcal{F}.a$ gives a's value as a function of the concrete state. Occurrences of a in preconditions and postconditions are replaced by function applications of the form $\mathcal{F}.a(\ldots)$. For example, $a[t] > 6$ becomes $\mathcal{F}.a(\ldots)[t] > 6$. The arguments to $\mathcal{F}.a$ are the variables on which a depends. The process of substituting $\mathcal{F}.a$ for a is called *functionalization*.

The **rep** declaration for a is rewritten into an appropriate axiom about $\mathcal{F}.a$ (a *rep axiom*). If a is visible but its representation is not, then $\mathcal{F}.a$ occurs in the rewritten program but its rep axiom does not. In this case, the methodology treats $\mathcal{F}.a$ as an uninterpreted function.

Functionalization and pointwise axioms. There are more details to be presented about functionalization. We will introduce them with an example. Consider

$$\begin{array}{l} \textbf{spec var } a\colon T \to X \\ \textbf{var } c\colon T \to Y \\ \textbf{var } d\colon T \to Z \\ \textbf{depends } a[t\colon T] \textbf{ on } c[t], d[t] \end{array} \tag{5}$$

Then occurrences of a are replaced by the expression $\mathcal{F}.a(c, d)$. Had c for example also been abstract, functionalization would continue, producing the expression $\mathcal{F}.a(\mathcal{F}.c(\ldots), d)$.

Notice that c, d, and $\mathcal{F}.a(c, d)$ are all maps (that is, fields). In typical functionalized expressions, we can expect to encounter expressions like $\mathcal{F}.a(c, d)[t]$. Allowing $\mathcal{F}.a$ to take maps as arguments is technically convenient, but without further restrictions, it would allow $\mathcal{F}.a(c, d)[t]$ to depend on the entire maps c and d, which we do not want: our view is that the dependency declaration (5) implies that $a[t]$ is unchanged by a modification to $c[s]$ or $d[s]$ for $s \neq t$. We enforce this point of view by imposing a *pointwise axiom* on each abstraction function. In the case of a, c, and d above, this axiom is:

$$\begin{array}{l} \langle\, \forall\, t\colon T,\ c0,\ c1,\ d0,\ d1 :: \\ \qquad c0[t] = c1[t] \ \wedge\ d0[t] = d1[t] \\ \qquad \Rightarrow\ \mathcal{F}.a(c0, d0)[t] = \mathcal{F}.a(c1, d1)[t] \,\rangle \end{array} \tag{6}$$

We would like to emphasize that in (6), variables $c0$, $c1$, $d0$, and $d1$ are dummies, not program variables. Even if program variables c and d were abstract, there would be no need to functionalize the dummies in (6).

For each abstract variable, there will be a pointwise axiom for each subtype of its index type, since different subtypes may have different dependencies. For example, consider

type T
spec var a: $T \to X$
type $U <: T$
var c: $U \to Y$
depends $a[u: U]$ **on** $c[u]$
type $V <: T$
var d: $V \to Z$
depends $a[v: V]$ **on** $d[v]$

There will be three pointwise axioms, one for each of the types T, U, and V. The axiom for T is (6), the same as the axiom where c and d had index type T. The axiom for U is

$$\begin{array}{l}\langle\, \forall\, u\colon U,\ c0,\ c1,\ d0,\ d1 ::\\ \quad c0[u] = c1[u]\\ \quad \Rightarrow\ \mathcal{F}.a(c0, d0)[u] = \mathcal{F}.a(c1, d1)[u]\ \rangle\end{array}$$

The axiom for V is similar.

When rewriting a postcondition, a post-value a' leads to post-values in the arguments to $\mathcal{F}.a$. For example, the postcondition of *init* for blank readers includes the conjunct

$$valid'[brd]$$

This is rewritten into

$$\mathcal{F}.valid(lo', cur', hi', buff', \mathcal{F}.svalid(lo', hi', buff', num'))[brd]$$

The number of arguments of $\mathcal{F}.a$ depends on the number of dependencies of a that are visible in the scope where the rewriting takes place. For example, in the unit *RdRep*, $\mathcal{F}.svalid$ has three arguments, whereas in *BlankRdImpl*, $\mathcal{F}.svalid$ has four arguments because of the extra dependency of $svalid[brd]$ on $num[brd]$. Within the verification of any one unit, all occurrences of $\mathcal{F}.a$ have the same number of arguments.

Rep axioms. We now explain how a **rep** declaration is rewritten into a rep axiom. A **rep** declaration has the form

$$\textbf{rep}\ a[t: T] \equiv R$$

where the only free variables allowed in R are fields that are dependencies of a, and each occurrence of such a field must be indexed by the dummy t. For definiteness, suppose that these dependencies are

var c: $T \to X$
var d: $T \to Y$
depends $a[t: T]$ **on** $c[t], d[t]$

This **rep** declaration is rewritten into the rep axiom

$$\langle \forall\, t\colon T,\ cV,\ dV\ ::\ t \neq \mathbf{nil}\ \Rightarrow\ \mathcal{F}.a(cV, dV)[t] = R(c, d := cV, dV)\ \rangle$$

in which we use the assignment operator to denote substitution. In this axiom, we have appended V's in the names of the dummies to emphasize that they are universally quantified dummies, not the program variables c and d.

The same treatment works with minor alterations to accommodate subtype-specific **rep** declarations and dependencies. For example, the **rep** declarations in

$$\begin{array}{l}\mathbf{type}\ T\\ \mathbf{spec\ var}\ a\colon T \to W\\ \mathbf{var}\ c\colon T \to X\\ \mathbf{depends}\ a[t\colon T]\ \mathbf{on}\ c[t]\\ \\ \mathbf{type}\ T0 <\colon T\\ \mathbf{var}\ d\colon T0 \to Y\\ \mathbf{depends}\ a[t\colon T0]\ \mathbf{on}\ d[t]\\ \mathbf{rep}\ a[t\colon T0]\ \equiv\ R0\\ \\ \mathbf{type}\ T1 <\colon T\\ \mathbf{var}\ e\colon T1 \to Z\\ \mathbf{depends}\ a[t\colon T1]\ \mathbf{on}\ e[t]\\ \mathbf{rep}\ a[t\colon T1]\ \equiv\ R1\end{array}$$

produce the rep axioms

$$\begin{array}{l}\langle \forall\, t\colon T0,\ cV,\ dV,\ eV\ ::\ t \neq \mathbf{nil}\ \Rightarrow\ \mathcal{F}.a(cV, dV, eV)[t] = R0(c, d := cV, dV)\ \rangle\\ \langle \forall\, t\colon T1,\ cV,\ dV,\ eV\ ::\ t \neq \mathbf{nil}\ \Rightarrow\ \mathcal{F}.a(cV, dV, eV)[t] = R1(c, e := cV, eV)\ \rangle\end{array}$$

Note that different rep axioms are produced for the different **rep** declarations. Note also that all dependencies of a for any subtype become arguments to $\mathcal{F}.a$, and each axiom ignores those arguments that are irrelevant to its subtype.

Examples. In the unit *RdRep* described previously, the precondition of *refill* is written

$$\mathit{valid}[\mathit{rd}]$$

Using the static dependencies of $\mathit{valid}[\mathit{rd}]$, this precondition is rewritten into

$$\mathcal{F}.\mathit{valid}(\mathit{lo}, \mathit{cur}, \mathit{hi}, \mathit{buff}, \mathit{svalid})[\mathit{rd}]$$

Since *svalid* is itself abstract, the rewriting continues:

$$\mathcal{F}.\mathit{valid}(\mathit{lo}, \mathit{cur}, \mathit{hi}, \mathit{buff}, \mathcal{F}.\mathit{svalid}(\mathit{lo}, \mathit{hi}, \mathit{buff}))[\mathit{rd}] \tag{7}$$

which is the final functionalized form of $\mathit{valid}[\mathit{rd}]$ in the scope *RdRep*.

As an example of a rep axiom, the **rep** for *valid* in *RdRep* is rewritten into

$$\begin{array}{l}\langle\, \forall\, rd\colon Rd.T,\ loV,\ curV,\ hiV,\ buffV,\ svalidV\ ::\\ \quad rd \neq \mathbf{nil}\ \Rightarrow\\ \quad (\mathcal{F}.valid(loV, curV, hiV, buffV, svalidV)[rd]\ \equiv\\ \qquad 0 \leq loV[rd] \leq curV[rd] \leq hiV[rd]\ \wedge\\ \qquad buffV[rd] \neq \mathbf{nil}\ \wedge\\ \qquad hiV[rd] - loV[rd] \leq \mathbf{number}(buffV[rd])\ \wedge\\ \qquad svalidV[rd])\ \rangle\end{array} \tag{8}$$

To see how these formulas work together, consider the verification of the method *refill*. The rewritten precondition (7) together with the rep axiom (8) allow the verifier to conclude that $buff[rd] \neq \mathbf{nil}$, by instantiating *buffV* to *buff*, *loV* to *lo*, and so on.

Because *RdRep* contains no **rep** declaration for *svalid*, $\mathcal{F}.svalid$ remains an uninterpreted function in this scope. A subtype-specific rep axiom is produced in the scope of an implementation of a reader subtype like *BlankRd*.

Our final example illustrates reasoning about the abstraction function as an uninterpreted function symbol. Consider the following generic procedure, which replaces a reader's buffer, copying the contents of the old buffer into the new:

```
proc copyBuffer(rd: Rd.T)
  requires valid[rd]
  modifies state[rd]
impl copyBuffer(rd: Rd.T) is
    var nb := new(byte, number(buff[rd])) in
      for i := 0 to number(buff[rd]) - 1 do
        nb[i] := buff[rd][i]
      end ;
      buff[rd] := nb
    end
  end
```

The proof that *copyBuffer* maintains *valid*$[rd]$ boils down to proving

$$\begin{array}{l} lo[rd] = lo'[rd]\ \wedge\ hi[rd] = hi'[rd]\ \wedge\ elems[buff[rd]] = elems'[buff'[rd]]\\ \Rightarrow\\ svalid[rd] = svalid'[rd]\end{array}$$

which functionalizes into

$$\begin{array}{l} lo[rd] = lo'[rd]\ \wedge\ hi[rd] = hi'[rd]\ \wedge\ elems[buff[rd]] = elems'[buff'[rd]]\\ \Rightarrow\\ \mathcal{F}.svalid(lo, hi, buff)[rd] = \mathcal{F}.svalid(lo', hi', buff')[rd]\end{array}$$

But this cannot be proved, since distinct arrays may have the same elements. Thus, the methodology would forbid *copyBuffer*, on the grounds that it possibly destroys the validity of *rd*. This prohibition is required by the reader design, since generic code is not allowed to modify *buff*. For example, the design of readers allows a subtype to cache the buffer pointer, but such a cache would be invalidated un-

expectedly by *copyBuffer*. Thus, reasoning about the abstraction function as an uninterpreted function symbol enforces the read-only by specification idiom.

An alternative design for readers would have replaced the dependency

$$\textbf{depends } \mathit{svalid}[\mathit{rd}] \textbf{ on } \mathit{buff}[\mathit{rd}]$$

by

$$\textbf{depends } \mathit{svalid}[\mathit{rd}] \textbf{ on } \mathit{elems}[\mathit{buff}[\mathit{rd}]]$$

In this design, *copyBuffer* would be legal, and it would be illegal for subtypes to assume that the buffer pointer remains unchanged by generic code.

So much for rewriting preconditions and postconditions. Now we consider rewriting modifies lists.

5.1 Modifies list desugaring

Guided by the abstraction modification principle (page 17), we introduce a closure operation on modifies lists. The closure operation expands the modifies list as required by the first half of the principle without expanding it so much as to violate the second half of the principle. The rewritten specification allows a method to modify a field $f[s]$ (abstract or concrete) if and only if the closure of the method's modifies list includes $f[s]$. Thus the rewriting is parameterized by the definition of closure. In this section, we first define the rewriting from a closed modifies list, and then define the closure operation appropriate for static dependencies. In Section 7.1, we will define the closure operation for dynamic dependencies.

Modification constraints. Closed modifies lists are rewritten into *modification constraints*. Consider a specification

$$\textbf{modifies } M \textbf{ ensures } P \tag{9}$$

occurring in a scope D. We rewrite this specification into

$$\textbf{modifies } N \textbf{ ensures } P \ \wedge \ Q$$

where N is the list of all concrete maps f for which a term of the form $f[E]$ occurs in the closure of M, and Q is a conjunction with one conjunct for each map variable visible in the scope. The conjunct for a particular map f asserts that $f[s]$ changes only where it is allowed to change. That is, if $\{\, f[E_1], \ldots, f[E_n] \,\}$ is the set of terms in the closure of M of the form $f[\ldots]$ (that is, the set of terms whose outer map variable is f), then the conjunct for f is

$$\langle\, \forall\, s \,::\, f[s] = f'[s] \ \vee \ s = E_1 \ \vee \ \ldots \ \vee \ s = E_n \,\rangle$$

We call this conjunct the *modification constraint* for f, and we call $\{E_1, \ldots, E_n\}$ the set of *modification points* of f. The modification constraint for a map variable limits the points at which the variable may be modified, that is, it protects the variable from change at other points. In particular, if the dependencies of an abstract variable a are changing at points where a itself is not allowed to be changed, a's modification constraint limits the modification of a's representation to preserve the values at points where a is not allowed to change. We say that a is protected from changes to its representation.

(A practical note: When verifying an implementation, ESC does not bother to produce a modification constraint for a map if a syntactic scan of the implementation determines that the map is never changed by the implementation.)

This strengthening of the postcondition occurs before the postcondition is functionalized.

Closure definition. A set of terms M is *statically closed* in a scope D if

$$a[E] \in M \ \wedge \ \text{“}\textbf{depends } a[t] \textbf{ on } c[t]\text{”} \in D \ \Rightarrow \ c[E] \in M$$

(We have intentionally ignored the type of E and the index types of a and c in this definition. Thus, a closed set of terms may include $f[E]$ even if E is not of the index type of f. Actually, ESC does use type information to produce a smaller closure, but in retrospect, we don't think it makes much difference.)

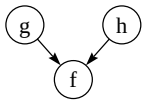
The *static closure* of a modifies list is its smallest statically closed superset.

For example, in the scope of the unit *BlankRdImpl*, the static closure of $valid[brd]$ is

$$lo[brd], cur[brd], hi[brd], buff[brd], svalid[brd], num[brd]$$

Example. As an artificial example, suppose that f is a concrete field and consider the dependencies

depends $g[t]$ **on** $f[t]$
depends $h[t]$ **on** $f[t]$



and the modifies list

$$\textbf{modifies } g[u], f[v]$$

The static closure of this modifies list is

$$\textbf{modifies } g[u], f[u], f[v]$$

This produces the rewritten specification

$$\begin{array}{l}\textbf{modifies } f \\ \textbf{ensures } \langle\, \forall\, s \ ::\ g[s] = g'[s] \ \vee \ s = u \,\rangle \wedge \\ \qquad\qquad \langle\, \forall\, s \ ::\ f[s] = f'[s] \ \vee \ s = u \ \vee \ s = v \,\rangle \wedge \\ \qquad\qquad \langle\, \forall\, s \ ::\ h[s] = h'[s] \,\rangle\end{array}$$

Notice that since there are no modification points for h, the conjunct for h in the rewritten postcondition does not allow it to be changed anywhere. Thus, the second half of the abstraction modification principle is satisfied: the license to modify $g[u]$ does not imply the license to modify $h[u]$, even though $g[u]$ and $h[u]$ have the concrete dependency $f[u]$ in common. Also, the license to modify $f[v]$ does not imply the license to modify $g[v]$ or $h[v]$.

Finally, functionalization produces

$$\begin{array}{l}\textbf{modifies } f \\ \textbf{ensures } \langle\, \forall\, s \ ::\ \mathcal{F}.g(f)[s] = \mathcal{F}.g(f')[s] \ \vee \ s = u \,\rangle \wedge \\ \qquad\qquad \langle\, \forall\, s \ ::\ f[s] = f'[s] \ \vee \ s = u \ \vee \ s = v \,\rangle \wedge \\ \qquad\qquad \langle\, \forall\, s \ ::\ \mathcal{F}.h(f)[s] = \mathcal{F}.h(f')[s] \,\rangle\end{array}$$

That is, the specification allows changes to f at indices u and v, provided the changes preserve the value of $\mathcal{F}.h(f)[s]$ for all s, and $\mathcal{F}.g(f)[s]$ for all s except u.

6. SOUNDNESS OF MODULAR VERIFICATION

We remind the reader that we are interested in modular soundness, that is, the property that the separate verifications of the individual modules of the program suffice to ensure the correctness of the composite program.

The standard approach for reasoning about procedure calls breaks down for modular programs. The standard approach reasons about a procedure call by assuming that it meets its specification, and discharges this assumption by verifying the implementation of the procedure. The approach breaks down if the specification is interpreted differently in the two contexts. But as we have seen, the meaning of a modifies list depends on the scope in which it is used. In particular, it may be desugared differently when reasoning about a call to a procedure than when reasoning about the implementation of the procedure.

To be more precise about modular soundness, we will define *scope monotonicity*, which means that anything verifiable in a scope is also verifiable in any larger scope. Then, we will argue that modular soundness is equivalent to scope monotonicity. The notion of scope monotonicity seems to be new.

For a scope D and a procedure implementation P in D, the judgment

$$D \vdash P$$

means that P meets its specification in D. More precisely, let

$$\textbf{requires } \mathit{Pre} \textbf{ modifies } M \textbf{ ensures } \mathit{Post} \tag{10}$$

be the result of desugaring the specification of P in scope D, as described in Section 5. Let A be the body of P, and let R be the conjunction of pointwise axioms and rep axioms in D, as described in Section 5.0. The requirement is that R implies that A meets the specification (10). In checking this, the verification condition generator reasons about method calls within A by using their specifications as desugared in D.

We say that $\vdash$ is *monotonic with respect to scope* if, for any procedure implementation P and scopes D and E,

$$\text{if } D \subseteq E, \text{ then } D \vdash P \text{ implies } E \vdash P$$

If we can prove that $\vdash$ is monotonic with respect to scope, then it is reasonable to say that our modular verification system is sound. For, if P has been verified in a scope D, that is, if we have proved $D \vdash P$, it follows by monotonicity that $E \vdash P$, where E is the entire program. Thus, anything that verifies in a limited scope would also verify had there been no information hiding and all information had been global.

It is too much to hope that $\vdash$ be monotonic in any program whatsoever. We will impose some requirements, called *modularity requirements*, such that $\vdash$ is monotonic in any program that meets the requirements. We will also argue that these requirements are reasonable from a methodological point of view, that is, that they don't rule out useful designs.

Our notion of modular soundness is different from the soundness of an axiomatic semantics with respect to an operational semantics. The consistency of axiomatic and operational semantics is certainly important, but it concerns the conventional control structures of programming in the small, like iteration and conditionals. These are mostly irrelevant to the issues of information hiding in programming in the large, which are the issues of concern in this paper. In this paper, we simply assume that the standard operational semantics is consistent with the axiomatic semantics of a single-module program. Therefore, any discrepancy between the axiomatic and operational semantics is due to unsound modular verification.

6.0 Visibility requirement

Our first modularity requirement is the *visibility requirement*. A program satisfies the visibility requirement if each of its static dependencies

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

is visible in every scope in which both a and c are visible.

It is easy to see that this requirement is necessary to have any hope of achieving scope monotonicity. Suppose there were a scope where a and c are visible but the dependency is not. In such a scope, it is provable that a change to c has no effect on a. But this would not be provable in a larger scope where the dependency is visible.

The requirement is necessary for informal as well as formal checking. If a program violated the requirement, it would be impossible to reason about a and c in the scope where they are visible but the dependency is not. In such a scope, an assignment to c could change a unexpectedly, and a call to a procedure that modifies a could change c unexpectedly. Nothing in the program text warns of either side effect. Almost all failures of scope monotonicity can be traced to unexpected side effects of this sort.

For example, consider what would happen if the dependency

$$\textbf{depends } \mathit{svalid}[\mathit{rd}] \textbf{ on } \mathit{hi}[\mathit{rd}]$$

were placed not in unit *RdRep* but in unit *BlankRdImpl*. A modular checking methodology would then allow the generic implementation, where the dependency of *svalid* on *hi* is not visible, to increase the value of $\mathit{hi}[\mathit{rd}]$ beyond $\mathit{num}[\mathit{rd}]$, which for blank readers destroys validity.

6.1 Top-down requirement

The second modularity requirement is the *top-down requirement*. A program satisfies the top-down requirement if, for each of its static dependencies

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

variable a is visible in every scope in which c is visible.

Here's an example of a rather pathological program unit that violates the top-down requirement.

```
unit U import Rd, RdRep
  type T <: Rd.T
  spec var isEven: T → bool
  depends isEven[t: T] on cur[t]
  rep isEven[t: T] ≡ cur[t] mod 2 = 0
  proc P(t: T)
    requires Rd.valid[t], isEven[t]
    modifies Rd.state[t]
  impl P(t: T) is
      t.getChar() ; assert cur[t] mod 2 = 0
    end
```

This pathological unit would verify, since

—the precondition of P requires $isEven[t]$,

—$isEven[t]$ does not appear in the modifies list of *getChar*, and consequently, $isEven[t]$ is formally provable at the exit of the call to $t.getChar()$, and

—the representation of $isEven[t]$ implies $cur[t] \bmod 2 = 0$.

But of course the assert would fail at run-time, since *getChar* will change the parity of $cur[t]$.

At first, this problem may not seem like a failure of scope monotonicity, but it is. The *getChar* method verified in the scope *RdImpl*, where it was presented earlier in the paper. But if the scope *RdImpl* were expanded by importing the unit U, then *getChar* would no longer verify, because it does not preserve the value of $isEven[t]$.

To put it another way, the problem is that *isEven* is not visible in the scope where *getChar* is implemented, and therefore the desugaring of *getChar*'s specification does not strengthen the postcondition to protect *isEven* from change. The top-down requirement ensures that *isEven* is visible wherever *cur* is, and thus any procedure that modifies *cur* and claims not to modify *isEven* will be checked appropriately.

To be completely clear, we are not suggesting that *isEven* should have been declared where *cur* was introduced. *isEven* should never have been introduced at all, the program is useless and mistaken. Our point is to identify a particular modularity requirement that is violated by this useless pile of code: the top-down requirement.

Here is an explanation of the name of this requirement. The reader package was designed in a top-down fashion, and *cur* was introduced as part of the concrete representation of *Rd.state* and *Rd.valid*. To come along later and define a new unit (U) that attempts to use *cur* for part of the representation of something else (*isEven*) would be a violation of top-down design. We believe that imposing the top-down requirement for static dependencies does not rule out any useful designs. As we shall see in Section 7, the situation is more interesting for dynamic dependencies.

6.2 Static placement rule

There is a simple discipline that guarantees that both the visibility and top-down requirements are satisfied, called the *static placement rule*: simply place each static

dependency

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

in the unit that declares c. We leave it to the reader to show that the visibility and top-down requirements follow from this rule. Furthermore, the converse is almost true: for programs without cyclic imports, if both requirements are satisfied, then the static placement rule is satisfied as well. Thus, if we'd like, we can replace both requirements by the rule. We have stated the requirements separately, because they seem to be separable concerns, and are used in different parts of the soundness proof.

6.3 Residues

The visibility and top-down requirements are two giant steps toward modular soundness. But they don't quite reach the goal. If they did, then the following implication would be true, for any procedure implementation P and scopes D and E containing static dependencies only:

$D \vdash P$ and $D \subseteq E$ and E satisfies the two modularity requirements
$\Rightarrow$
$E \vdash P$

Unfortunately, given what we have said so far, this is false. There is one more technicality that must be introduced to fix the problem, called *residues*. Here is an artificial program that demonstrates the problem:

unit A
  type T
  spec var a: $T \rightarrow$ **any**
  var c: $T \rightarrow$ **int**
  depends $a[t$: $T]$ **on** $c[t]$
  proc $outer(t$: $T)$
  proc $inner(t$: $T)$ **modifies** $a[t]$ **ensures** $c[t] = c'[t]$
  impl $outer(t$: $T)$ **is** $t.inner()$ **end**

The absence of a modifies list for *outer* means that a call to *outer* has no side effects. We will now argue that without residues, unit A verifies. We then argue that it should not verify. Finally, we will define residues and explain how they fix the problem.

As described in Section 5, the modifies list $a[t]$ of the call $t.inner()$ has the static closure $a[t]$, $c[t]$, so the rewritten specification of $t.inner()$ (before functionalization) is

modifies c
ensures $c[t] = c'[t] \wedge$
  $\langle \forall\, s \;::\; c[s] = c'[s] \;\vee\; s = t \,\rangle \wedge$
  $\langle \forall\, s \;::\; a[s] = a'[s] \;\vee\; s = t \,\rangle$

After functionalization, the specification is

modifies c
ensures $c[t] = c'[t] \wedge$
  $\langle \forall\, s \;::\; c[s] = c'[s] \;\vee\; s = t \,\rangle \wedge$
  $\langle \forall\, s \;::\; \mathcal{F}.a(c)[s] = \mathcal{F}.a(c')[s] \;\vee\; s = t \,\rangle$

The first two lines imply that $c[s]$ does not change for any s. The third line then implies that $a[s]$, that is, $\mathcal{F}.a(c)[s]$, also does not change for any s. Therefore, the call $t.inner()$ has no side effects at all, and the body of *outer* will verify.

But we now argue that *outer*'s body should not verify. Consider the following unit B, providing an implementation of *inner*.

```
unit B import A
  var d: A.T → int
  depends a[t: A.T] on d[t]
  impl inner(t: A.T) is d[t] := 0 end
```

Unit B reveals another dependency (d) of a, which the implementation of *inner* in fact modifies. Unit B will verify in isolation, because *inner* modifies only variables in the static closure of its modifies list $a[t]$.

We are in trouble, because *outer*'s side effect on d will be unexpected in a scope that sees d together with *outer*'s specification:

```
unit C import A, B
  proc R(t: A.T) modifies a[t]
  impl R(t: A.T) is
      var dd := d[t] in t.outer() ; assert dd = d[t] end
    end
```

This implementation verifies, because *outer*'s modifies list does not include d, but clearly the assert will fail at run-time.

This is a failure of scope monotonicity, because although *outer*'s body verifies in the unit A, it would not do so in the larger scope of unit B, where d is visible and the call $t.inner()$ will be desugared to have a side effect on $d[t]$.

We blame the failure on the body of *outer*. Here's an informal explanation of why: Procedure *outer*, which is specified to be side-effect free, calls *inner*, which modifies a. Although a depends on c, it should not be inferred that a depends only on c. Therefore, the call to *inner* should be inconsistent with *outer*'s modifies list.

Individual residues. To change our rewriting so that *outer*'s body will not verify, we introduce residues. The *residue* of an abstract variable a, written $res.a$, can be viewed as a stand-in for those of a's dependencies that are not visible. Residues are introduced automatically by the methodology and cannot be mentioned explicitly in specifications or programs. The methodology treats every abstract variable declaration

```
spec var a: T → X
```

as a shorthand for the three declarations

```
spec var a: T → X
var res.a: T → any
depends a[t: T] on res.a[t]
```

The implicit dependency of a on $res.a$ introduces $res.a$ into the static closure of any modifies list that mentions a, just as for explicit dependencies.

Desugaring of modifies lists as described in Section 5 will now work out soundly for this example. The modifies list $a[t]$ of the call $t.inner()$ in the body of *outer*

```
unit D
  type T
  spec var a: T → int
  spec var b: T → int
  var c: T → int
  depends b[t: T] on c[t]
  proc outer(t: T) modifies a[t]
  proc inner(t: T) modifies a[t]
  impl outer(t: T) is
      var cc := c[t] in
        c[t] := 0 ; t.inner() ; c[t] := cc
      end
    end
```

Fig. 7. Example program that motivates shared residues.

has the static closure $a[t], res.a[t], c[t]$, so the rewritten specification of $t.inner()$ (before functionalization) is

$$\begin{array}{l}\textbf{modifies } c, res.a\\ \textbf{ensures } c[t] = c'[t] \wedge\\ \qquad \langle\, \forall\, s \;::\; c[s] = c'[s] \;\vee\; s = t \,\rangle \wedge\\ \qquad \langle\, \forall\, s \;::\; res.a[s] = res.a'[s] \;\vee\; s = t \,\rangle \wedge\\ \qquad \langle\, \forall\, s \;::\; a[s] = a'[s] \;\vee\; s = t \,\rangle\end{array}$$

This allows both $a[t]$ and $res.a[t]$ to change, and therefore the implementation of *outer* will not verify.

We would like to point out that residue variables are an implementation technicality. Users of an automatic program checker based on our methodology need not be aware of them: users never write them, and the only time users would ever read them is in a warning message that a procedure might modify some residue variable in violation of the procedure's modifies list. The user must learn that such a warning means that the corresponding abstraction was modified by some procedure call in a manner that could have changed details of the representation that are not visible in the verification scope.

Shared residues. We are now very close to modular soundness, so close that it took our colleague Jim Saxe to find a sufficiently pathological example to demonstrate that we are not yet there. The example is shown in Figure 7. In this scope, the implementation of *outer* verifies, because $a[t]$ and $res.a[t]$ are allowed to be changed, $c[t]$ is restored to its initial value, $res.b[t]$ is not changed by the body, and the invariance of $b[t]$ (i.e., of $\mathcal{F}.b(res.b, c)[t]$) follows from the invariance of $res.b$ and c. But in a larger scope in which it is revealed that a and b have a common dependency, *outer* will not verify:

```
unit E import D
  var d: D.T → int
  depends a[t: D.T] on d[t]
  depends b[t: D.T] on d[t]
```

In scope E, the required proof of invariance of $b[t]$ for *outer* does not go through. The modification constraint for b that is added to the postcondition of *outer* is

$$\langle\, \forall\, s \;::\; \mathcal{F}.b(res.b, c, d)[s] = \mathcal{F}.b(res.b, c', d')[s] \,\rangle \tag{11}$$

The modification constraints for b and d that we get to assume at the return from the call to *inner* are

$$\begin{array}{l} \langle\, \forall\, s \;::\; d[s] = d'[s] \;\vee\; s = t \,\rangle \;\wedge \\ \langle\, \forall\, s \;::\; \mathcal{F}.b(res.b, store(c, t, 0), d)[s] = \mathcal{F}.b(res.b, store(c, t, 0), d')[s] \,\rangle \end{array} \tag{12}$$

where the expression $store(c, t, 0)$ denotes a map like c but mapping t to 0.

But (11) does not follow from (12). Although *inner* is constrained to modify d only in ways that preserve $\mathcal{F}.b(res.b, c, d)$, this constraint is in force only for the value of c at the time of the call to *inner*, which in terms of the initial value of c is $store(c, t, 0)$. Therefore, scope monotonicity and modular soundness do not hold.

To restore modular soundness, we must arrange either that *outer* verifies in unit E or that it does not verify in unit D. We choose the latter, that is, we take the view that *outer* was misprogrammed: modifying part of the representation (c) of an abstraction (b) whose representation is hidden and then calling a method (*inner*) that may manipulate the abstraction is methodologically unjustifiable, even if the modification of c is restored after the call.

Consider that the example might continue as follows:

```
rep b[t: T] ≡ c[t] · d[t]
impl inner(t: T) is
    if c[t] = 0 then d[t] := d[t] + 1 end
  end
```

This possible continuation shows clearly that the failure of *outer* to verify in unit E is appropriate, and therefore its verification in unit D is inappropriate.

The essential difficulty revealed by Saxe's example is that two abstract variables that have no common dependency in a small scope may turn out to have a common dependency in a larger scope. To fix our proof system to be modularly sound, we will force all small-scope verifications to respect the possibility that larger scopes may reveal common dependencies. To do this, we introduce another residue variable, a *shared residue sres* to augment the individual residues introduced earlier.

In more detail, *sres* is a predeclared variable visible in all scopes. The methodology treats every abstract variable declaration

```
spec var a: T → X
```

as shorthand for

```
spec var a: T → X
depends a[t: T] on sres[t]
var res.a: T → any
depends a[t: T] on res.a[t]
```

The combination of individual and shared residues achieves modular soundness. For Saxe's example, the attempted verification of *outer* in unit D will now fail: the details of the failure are exactly the previously described details of the failure of

outer to verify in unit E (see formulas (11) and (12)) with *sres* playing the role of d.

It seems necessary to introduce both the shared residue and the individual residues. Here is an example that shows that the shared residue alone does not suffice for modular soundness. We begin with a small unit G:

unit G
  type T
  spec var a: $T \to X$
  spec var b: $T \to Y$
  proc $outer(t\colon T)$ **modifies** $a[t]$
  proc $inner(t\colon T)$ **modifies** $b[t]$ **ensures** $b[t] = b'[t]$
  impl $outer(t\colon T)$ **is** $t.inner()$ **end**

The following unit H shows that in a larger scope, the call to *inner* may have side effects that are not allowed by *outer*'s specification:

unit H **import** G
  var c: $T \to Z$
  depends $b[t\colon T]$ **on** $c[t]$

But with the shared residue variable only, *inner*'s modification of the shared residue is consistent, in unit G, with *outer*'s modification constraint. To achieve the verification failure that we need, we must distinguish $res.a$ from $res.b$.

The residue constraint. In summary, in addition to the modularity requirements, we impose the following *residue constraint*:

> Each abstract variable a implicitly depends on $res.a$ and on *sres*, and neither $res.a$ nor *sres* appears in any user-supplied declaration.

It may now seem that we need additional shared residue variables for every pair (or subset) of abstract variables. But this is not the case, as our soundness theorem shows.

6.4 Modular soundness for static dependencies

The two modularity requirements and the residue constraint are the whole story. We can now state the modular soundness theorem for static dependencies:

THEOREM 6.0. *Modular soundness for static dependencies For any scopes D and E containing static dependencies only, and any procedure implementation P in D,*

$D \vdash P$ *and* $D \subseteq E$ *and*
D *and* E *satisfy the two modularity requirements and the residue constraint*
$\Rightarrow$
$E \vdash P$

We have proved this theorem, but since the proof is more than 80 pages long and has been printed elsewhere [Leino and Nelson 2000], we have not asked TOPLAS to include it here. Instead, we briefly outline the proof.

The soundness theorem states that any implementation that verifies in a scope D will also verify in a scope E that contains D. The proof is by induction on the

number of declarations in $E \setminus D$: that is, we can imagine proceeding from D to E by adding declarations gradually, proving for each addition that the verifiability of the method implementation is not lost. For some kinds of added declarations (type declarations, rep declarations, method declarations, and method implementation declarations), the proof is quite easy. The hard case is where the added declarations consist of a new field together with some number of dependencies on the field.

In this hard case, let ε be the new field. The are two differences between the small-scope and large-scope verification conditions.

The first difference is that, for any abstract variable a that depends on ε in the large scope, occurrences of a are functionalized differently in the two scopes. Wherever we have in the small scope a subexpression like $\mathcal{F}.a(sres, \ldots)$, we have instead in the large scope a subexpression like $\mathcal{F}.a(sres, \ldots, \varepsilon)$. The soundness proof handles this difference by instantiating the universally quantified *sres* with something like the pair $(sres, \varepsilon)$.

The second and more difficult difference is that modifies lists are desugared differently in the two scopes: the modification constraints in the large scope mention ε, but the corresponding modification constraints in the small scope do not. This adds a proof obligation to the large scope that was not present in the small scope (namely that ε is changed only at its modification points). It also changes the desugaring of method calls within the implementation being verified, adding assumptions about how these calls modify ε. The proof handles this difference by using individual residues: it can be shown that the set of modification points of ε equals the union of the modification points of the individual residue variables $res.a$ for all a that depend on ε. This fact, and the fact that the small-scope verification condition includes that the method meets its modification constraints for each such $res.a$, together imply that the new modification condition (for ε) in the large-scope verification condition is valid.

A rather different approach has been taken in the PhD thesis of Peter Müller [2001], done under the supervision of

Arnd Poetzsch-Heffter. In the Müller/Poetzsch-Heffter approach, each verification condition is an implication whose consequent is scope independent and whose antecedent is a conjunction whose strength is a monotonic function of the scope in which the verification condition is constructed. This makes the soundness theorem trivial. Differences between their system and ours that contribute to their short soundness proof include: they encode the dependency relation in the logic, they introduce a single variable for the whole heap rather than variables for the data fields visible in a particular scope, and they do not functionalize abstract variables. As a consequence of their design, they do not need residue variables. The Müller/Poetzsch-Heffter system has been used to verify some small programs. These verifications were interactive rather than automatic. It seems that the complexity of our soundness proof is the cost of design decisions that we took in order to build a checker that is practical to use on realistic systems programs.

This marks the end of our presentation of static dependencies. In the next section, we describe dynamic dependencies.

7. DYNAMIC DEPENDENCIES

Most of the dependencies that arise in top-down program design are static. By a top-down design, we mean a design in which each successive layer of implementation provides the representation of the abstraction specified in layers above. However, not all useful designs are top-down. A bottom-up design is often better, in which an object type is defined and later used to build higher-level objects, which may not even have been envisioned at the time the first type was defined. Most of the dependencies that arise in bottom-up design are dynamic.

Recall that a dynamic dependency has the form

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

This means that the abstract state $a[t]$ is represented in terms of the concrete state $c[b[t]]$, which is a field not of the object t but of the separate object $b[t]$. The field b is called a *pivot field*. Pivot fields introduce a level of indirection that makes dynamic dependencies more complicated than static dependencies. Static dependencies allow the representation of an abstraction to be divided among several modules; dynamic dependencies allow it also to be divided among several dynamically allocated objects.

For example, sequences are useful abstractions. To define sequences and then use them in different ways is a bottom-up approach, which leads to the use of dynamic dependencies. To see this, consider a set type $Set.T$ implemented in terms of a sequence type $Seq.T$. Somewhere in the set implementation, there will be a field, say q, declared as

$$\textbf{var } q\colon Set.T \to Seq.T$$

The representation of the validity and state of a set s will inevitably involve properties of the sequence $q[s]$. Almost always, for example, set validity requires validity of the underlying sequence, in which case we have

$$\textbf{rep } Set.valid[s\colon Set.T] \equiv \ \ldots \ \wedge \ Seq.valid[q[s]]$$

This **rep** declaration requires the dependency

$$\textbf{depends } Set.valid[s\colon Set.T] \textbf{ on } Seq.valid[q[s]]$$

which is dynamic, with pivot field q.

In proceeding from static to dynamic dependencies (and, in Section 9, to other dependencies and invariants), the character of our results changes. The design of our methodology will more and more be guided by programming judgment of what is appropriate, rather than a mathematical observation of what is correct. We will end up with a system that we hope is consistent and sound, but with no soundness theorem. For those readers who may be alarmed by this, we make two observations.

The first observation is that mathematical hygiene is not the same thing as methodological value. The Algol 60 report may not be up to the metamathematical standards of the *Journal of Symbolic Logic*, but it was certainly a valuable contribution to programming methodology.

The second observation is that although our treatment of dynamic dependencies is unfinished mathematically, and we have no soundness theorem, it nevertheless compares favorably even on this dimension with much of the published work in

related areas. Data abstraction functions that depend on concrete state that is distributed between various dynamically linked objects have been used implicitly by programmers, as we found when we verified the Modula-3 libraries with our checker, but in the literature, the problem of reasoning about such programs has not been identified and confronted directly. Related problems, like rep exposure and aliasing, have been the subject of much discussion and many strictures, but the discussions have often been so imprecise as to sweep critical issues under the rug.

The material in the remainder of the paper is a record of what we learned by implementing a program verifier and applying it to thousands of lines of modern systems programs. The record is at a sufficient level of precision to allow others to duplicate our results by reimplementing our verifier.

In this section, we will explain how dynamic dependencies affect functionalization and modifies list desugaring, and then explain what we believe about their modularity requirements.

7.0 Functionalization

Functionalization in the presence of dynamic dependencies is analogous to functionalization in the presence of static dependencies only. Both of the fields in the right-hand side of the dynamic dependency become arguments to the abstraction function.

For example, in the presence of the dependencies

$$\begin{array}{l}\textbf{depends } a[t] \textbf{ on } e[t]\\ \textbf{depends } a[t] \textbf{ on } c[b[t]]\end{array}$$

the functionalized form of $a[x]$ is

$$\mathcal{F}.a(e, c, b)[x]$$

or, more precisely, taking residues into account,

$$\mathcal{F}.a(sres, res.a, e, c, b)[x]$$

The pointwise axiom for $\mathcal{F}.a$ is

$$\begin{array}{l}\langle\, \forall\, t, sres0, sres1, res.a0, res.a1, e0, e1, c0, c1, b0, b1 ::\\ \quad sres0[t] = sres1[t] \ \wedge\ res.a0[t] = res.a1[t] \ \wedge\ e0[t] = e1[t] \wedge\\ \quad c0[b0[t]] = c1[b1[t]]\\ \Rightarrow\\ \quad \mathcal{F}.a(sres0, res.a0, e0, c0, b0)[t] =\\ \quad \mathcal{F}.a(sres1, res.a1, e1, c1, b1)[t]\ \rangle\end{array}$$

We don't introduce anything like residue variables for dynamic dependencies.

7.1 Modifies list desugaring

Unlike functionalization, which is pretty much the same for static and dynamic dependencies, modifies list desugaring is surprisingly different in the two cases. It has taken us several tries to converge on a desugaring that suits all the examples that we know.

In this subsection, we assume that no abstract variable depends, directly or indirectly, on itself. This restriction will be lifted in Section 9.0.

To explain the issues, we start by exploring the obvious extension of the approach for static dependencies, and show how this goes wrong. Then we give what we think is the right desugaring, followed by two more supporting examples. Finally, we impose a restriction that seems to be necessary to make the desugaring sound.

Recall the main points of Section 5.1:

—the definition of closure,
—the rule that **modifies** M allows the modification of anything in the closure of M, and
—the modification constraints that enforce the rule.

We will reuse the second and third points. That is, to accommodate dynamic dependencies, we redefine closure and leave everything else the same.

The need to close upwards. Recall that a set of terms M is statically closed in a scope D if it satisfies the property

$$a[E] \in M \ \wedge \ \text{“}\textbf{depends}\ a[t]\ \textbf{on}\ c[t]\text{”} \in D \ \Rightarrow \ c[E] \in M \tag{13}$$

The obvious extension to include dynamic dependencies is to require in addition:

$$a[E] \in M \ \wedge \ \text{“}\textbf{depends}\ a[t]\ \textbf{on}\ c[b[t]]\text{”} \in D \ \Rightarrow \ c[b[E]] \in M \tag{14}$$

To give this new closure a name, we define a set of terms M to be *downward closed* in a scope D if it satisfies (13) and (14). Will we get a good desugaring if we replace “static closure” with “downward closure” in Section 5.1? Unfortunately not.

To explain why the replacement doesn't work, we give a straightforward example of integer sets implemented in terms of extensible integer sequences. Figure 8 shows the two interfaces, together with ESC-style specifications. (In these interfaces, we have varied our convention and elected not to return anything from the *init* methods.) A simple implementation of all *Set* objects, in which all elements are kept in a sequence with duplicates allowed, begins as shown in Figure 9.

The whole point of this example is: what will be the effective modifies list used in reasoning about the call to *Seq.init* in the body of *Set.init*? Since $Seq.init(sq)$ modifies $valid[sq]$, $state[sq]$, and $length[sq]$, the modifies list (before closure) of the call $q[st].init()$ is

$$\textbf{modifies}\ Seq.valid[q[st]], Seq.state[q[st]], Seq.length[q[st]]$$

This is also the modifies list after closure, since it is already downward closed (not counting residues, which we will ignore since they are irrelevant to this example). Transforming the closed modifies list into modification constraints, the postcondition of the rewritten specification is

$$\begin{aligned}
\textbf{ensures}\ & \langle \forall\, sqv \ ::\ Seq.valid[sqv] = Seq.valid'[sqv] \ \vee \ sqv = q[st] \ \rangle \ \wedge \\
& \langle \forall\, sqv \ ::\ Seq.state[sqv] = Seq.state'[sqv] \ \vee \ sqv = q[st] \ \rangle \ \wedge \\
& \langle \forall\, sqv \ ::\ Seq.length[sqv] = Seq.length'[sqv] \ \vee \ sqv = q[st] \ \rangle \ \wedge \\
& \langle \forall\, stv \ ::\ Set.valid[stv] = Set.valid'[stv] \ \rangle \ \wedge \\
& \langle \forall\, stv \ ::\ Set.state[stv] = Set.state'[stv] \ \rangle \ \wedge \\
& \langle \forall\, stv \ ::\ q[stv] = q'[stv] \ \rangle
\end{aligned}$$

```
unit Set
  type T
  spec var valid: T → bool
  spec var state: T → any
  proc init(st: T)
    modifies valid[st], state[st]
    ensures valid'[st]
  proc insert(st: T, x: int)
    requires valid[st]
    modifies state[st]
  proc delete(st: T, x: int)
    requires valid[st]
    modifies state[st]
  proc member(st: T, x: int): bool
    requires valid[st]

unit Seq
  type T
  spec var valid: T → bool
  spec var length: T → int
  spec var state: T → any
  proc init(sq: T)
    modifies valid[sq], state[sq], length[sq]
    ensures valid'[sq] ∧ length'[sq] = 0
  proc addhi(sq: T, x: int)
    requires valid[sq]
    modifies state[sq], length[sq]
    ensures length'[sq] = length[sq] + 1
  proc get(sq: T, i: int): int
    requires valid[sq] ∧ 0 ≤ i < length[sq]
```

Fig. 8. The interfaces *Set* for sets and *Seq* for sequences.

```
unit SetImpl import Set, Seq
  var q: Set.T → Seq.T
  rep Set.valid[st: Set.T] ≡ q[st] ≠ nil ∧ Seq.valid[q[st]]
  depends Set.valid[st: Set.T] on q[st], Seq.valid[q[st]]
  depends Set.state[st: Set.T] on Seq.state[q[st]], Seq.length[q[st]]
  impl init(st: Set.T) is
      q[st] := new(Seq.T) ; q[st].init()
    end
  impl insert(st: Set.T, x: int) is q[st].addhi(x) end
   ⋮
```

Fig. 9. The implementation unit *SetImpl*.

The fourth conjunct "protects" the higher-level abstraction *Set.valid* from a change to its representation. In the old world of static dependencies only, this was necessary, but in the new world with dynamic dependencies, it is preposterous. The whole purpose of the *Seq.init* call is to modify the validity of the enclosing set.

The example shows that using the downward closure produces too strong an **ensures** clause, that is, too small a closure.

Let us summarize what the example has shown about the difference between static dependencies and dynamic dependencies. In the presence of a static dependency of $a[t]$ on $c[t]$, the presence of the term $c[x]$ in the modifies list does not, and should not, imply the presence of $a[x]$ in the closure, since the license to modify a concrete variable does not imply the license to modify an abstract variable that depends on it. However, in the presence of a dynamic dependency of $a[t]$ on $c[b[t]]$, the example shows that the presence of the term $c[x]$ in the modifies list should imply the presence of $a[t]$ in the closure, for any t such that $b[t] = x$. That is, we must close upwards as well as downwards.

Dynamic closure. In the next few paragraphs, we define the closure that we use when desugaring modifies lists in the presence of dynamic dependencies, which we call the *dynamic closure*. We have already indicated that it is larger than the downward closure. In fact, it is the union of the downward closure with a portion of the upward closure (defined soon).

Another change from our previous treatment is that the closure will contain expressions of the form f^{-1}. We call these "map inverses", but they are not to be thought of as ordinary notation, for example, the notation does not imply that f is invertible: they are a syntactic fiction that will be eliminated when the closure is transformed into a modification constraint. The elimination is achieved by rewriting an equality of the form

$$s = f_1^{-1}[f_2^{-1}[\cdots[f_n^{-1}[E]]]] \tag{15}$$

into

$$f_n[\cdots[f_2[f_1[s]]]] = E$$

All of the map inverses will be eliminated by this rewriting, because the terms of the closure of a modifies list affect the rewritten specification only in modification constraints, in which map inverses will occur only in equalities of the form (15).

The *dynamic closure* in a scope D of a modifies list M is the union of the downward closure in D of M with the upward closure in D of the flexible subset of M.

The *flexible subset* of a set of terms M in a scope D consists of those terms $f[E]$ where D contains no dependency of the form **depends** $a[t]$ **on** $f[t]$.

A set of terms M is *upward closed* in a scope D if

$$c[E] \in M \ \wedge \ \text{“}\textbf{depends } a[t] \textbf{ on } c[t]\text{”} \in D \ \Rightarrow \ a[E] \in M$$
$$c[E] \in M \ \wedge \ \text{“}\textbf{depends } a[t] \textbf{ on } c[b[t]]\text{”} \in D \ \Rightarrow \ a[b^{-1}[E]] \in M$$

The *upward closure* of a set of terms is its smallest upward-closed superset.

Examples. Let us redo the *Set.init* example with the new rule. The desugaring begins, as before, with the modifies list from the specification, namely

$$\textbf{modifies } \mathit{Seq.valid}[q[st]], \mathit{Seq.state}[q[st]], \mathit{Seq.length}[q[st]]$$

The dynamic closure of this list includes the term

$$Set.valid[q^{-1}[q[st]]]$$

since the scope includes the dependency

$$\textbf{depends } Set.valid[st] \textbf{ on } Seq.valid[q[st]]$$

This extra term in the closure weakens the modification constraint for $Set.valid$. With the downward closure, the constraint was

$$\langle \forall\, stv \;::\; Set.valid[stv] = Set.valid'[stv] \,\rangle$$

However, with the dynamic closure, the constraint is

$$\langle \forall\, stv \;::\; Set.valid[stv] = Set.valid'[stv] \;\vee\; stv = q^{-1}[q[st]] \,\rangle$$

which when map inverses are eliminated becomes

$$\langle \forall\, stv \;::\; Set.valid[stv] = Set.valid'[stv] \;\vee\; q[stv] = q[st] \,\rangle$$

which in turn is functionalized to

$$\begin{array}{l}\langle \forall\, stv \;::\; \mathcal{F}.Set.valid(sres, res.Set.valid, q,\\ \qquad\qquad \mathcal{F}.Seq.valid(sres, res.Seq.valid))[stv]\\ \qquad = \\ \qquad \mathcal{F}.Set.valid(sres', res.Set.valid', q',\\ \qquad\qquad \mathcal{F}.Seq.valid(sres', res.Seq.valid'))[stv]\\ \quad \vee\; q[stv] = q[st] \,\rangle\end{array}$$

which eliminates the problem since the disjunct $q[stv] = q[st]$ allows the method to change the validity of st. (The disjunct also allows the method to change the validity of any other set whose q field coincides with the q field of st. This accurately reflects the semantics of the situation, and we take it as evidence that our rewriting is appropriate. It is a different issue whether the designer of $Set.T$ should allow such sharing of the q field—probably not, as explained in Section 9.3.)

Here is an example to show why the dynamic closure is the union of two closures, rather than, for example, the upward closure of the downward closure or some kind of bi-directional closure. Suppose that we were doing full functional verification instead of extended static checking only, and that sets were represented by sequences without duplicates. Then we would have the dependency

$$\textbf{depends } Set.valid[st] \textbf{ on } Seq.state[q[st]]$$

since the **rep** declaration for $Set.valid[st]$ would forbid duplicates in $q[st]$, which is an assertion about $Seq.state[q[st]]$. In addition, we still have the dependency

$$\textbf{depends } Set.state[st] \textbf{ on } Seq.state[q[st]]$$

If the dynamic closure were the upward closure of the downward closure, then the dynamic closure of the modifies list

$$\textbf{modifies } Set.state[st]$$

would include $Set.valid[st]$. Thus, in the scope of the implementation, any operation that changes the state of a set would be allowed also to modify its validity, which

would be preposterous. (It would also be unsound, since in the scope of a client of the *Set* interface, such a side effect would be unexpected.)

Finally, here is an example to show why the dynamic closure contains the full upward closure of the flexible terms of a modifies list, rather than a single level. Suppose R is a subtype of $Rd.T$ (see Section 4), that

$$\textbf{var } rq\colon R \to Seq.T$$

is a sequence-valued field of R readers, and that the subtype-specific validity of R readers depends on the validity of the associated sequence:

$$\textbf{depends } svalid[r\colon R] \textbf{ on } Seq.valid[rq[r]]$$

In this scenario, we would argue that a call that modifies $Seq.valid[rq[r]]$ should be allowed to modify both $svalid[r]$ and $Rd.valid[r]$.

Dependency segregation. Our desugaring of modifies lists requires a restriction, which we call the *dependency segregation restriction*: no field c occurs both in a static dependency of the form $a[t]$ on $c[t]$ and in a dynamic dependency of the form $z[s]$ on $c[b[s]]$. Because of the visibility and top-down requirements, this restriction can easily be enforced modularly.

To see that this restriction is necessary, consider the following example:

unit A
  ⋮
  depends $a[t]$ **on** $c[t]$
  proc $P(t)$ **modifies** $c[t]$
unit B **import** A
  ⋮
  depends $z[s]$ **on** $c[b[s]]$
  ... call $t.P()$ ...

Because $c[t]$ is not in the flexible subset of the modifies list of P, the caller in B expects the value of $z[b^{-1}[t]]$ to be unchanged. However, if the implementation of P is placed in unit A (or in any unit where the dynamic dependency is not visible), then no modification constraint will be added to the implementation to enforce the unchangedness of z.

The dependency segregation restriction does not seem to rule out any useful programs.

7.2 Modularity requirements for dynamic dependencies

The visibility and top-down requirements that we impose for static dependencies both have analogues for dynamic dependencies, but the analogues are significantly different from the originals. One of the differences is that because pivot fields can be updated dynamically, several of the requirements for dynamic dependencies can be checked only by reasoning about specifications, not by a simple check on the placement of declarations. Our methodology enforces the requirements of this sort by transforming an annotated input program into another annotated program that will verify exactly when the input program would verify and the input program obeys the requirements.

Before getting into these deep waters, we describe the one modularity requirement for dynamic dependencies that can be checked simply by looking at the placement of declarations.

Pivot visibility requirement. The *pivot visibility requirement* requires that a dynamic dependency

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

be visible anywhere b is.

This can be enforced simply by checking that the dependency is placed in the same unit as the declaration of b.

Here is the reason we impose the requirement. If there were a scope where a and b are visible but the dependency is not, then a modification to $b[t]$ could change $a[t]$ unexpectedly. The requirement is not burdensome, since the module that implements the abstraction a usually declares both the pivot field and the dependency.

It would probably be sound to require only that the dependency be visible where both a and b are, but we have not found any examples where the extra flexibility of this weaker requirement would be of any engineering use.

Absence of abstract aliasing. The visibility requirement for static dependencies prevents unexpected side effects between an abstract variable and its representation. For the dynamic dependency of $a[t]$ on $c[b[t]]$, the pivot visibility requirement prevents unexpected side effects between a and b, but we still need to protect against unexpected side effects between a and c. This is the function of *absence of abstract aliasing*.

For static dependencies, the problem is solved by requiring the dependency to be visible anywhere both a and c are, but for dynamic dependencies, this would be undesirably strict. For example, consider the sets and sequences described earlier. This strict version of the requirement would force the dependency (and therefore the pivot field as well) to be declared in the public interface *Set* instead of in the private implementation where they belong. (It is obviously unreasonable to place the pivot and dependency declarations in the public interface *Seq*, since sets may not have been envisioned when sequences were defined.)

To find the right modularity requirement, we focus on the situation that goes wrong, and use our judgment as programmers to assign blame. The situation that goes wrong is an unexpected side effect between $a[t]$ and $c[u]$ for some values t and u. For the side effect to happen, it must be that $b[t] = u$. For the side effect to be unexpected, it must occur in a scope where a and c are visible but the dependency is not. Because of the pivot visibility requirement, it must therefore be that b is not visible either.

More formally, we say that *abstract aliasing* occurs if execution reaches some point in the program text where, for some expressions E and F and pivot field b, all free variables of E and F are visible, b is not visible, and $E = b[F] \ \wedge \ E \neq \textbf{nil}$. Notice that the condition $E = b[F]$ makes sense even outside the scope of b, since b's value exists even at program points where b is not visible. We require that programs be designed so that abstract aliasing does not occur. This requirement, together with the pivot visibility requirement, is the analogue for dynamic dependencies of

the visibility requirement for static dependencies.

So much for the definition of abstract aliasing. A further question is to find a static discipline for avoiding the problem.

One simple discipline that prevents abstract aliasing would be to forbid communicating a pivot value $b[t]$ into or out of the scope declaring b. All forms of communication must be forbidden, including communication via procedure parameters, procedure results, and global and heap locations. We say that the value of a pivot field transferred into or out of the scope of the field's declaration is *leaked*.

Unfortunately, the simple discipline of forbidding all leaking is too strict, for several reasons. For example, initialization methods occasionally take parameters that are stored into pivot fields. Also, methods of container classes must return the elements of the container. Most compellingly, to operate on a pivot, an implementation of an abstraction must pass the pivot value to the pivot's own methods.

We have defined a more flexible discipline for avoiding leaking, which solves the three problems mentioned in the previous paragraph. But our solution is not totally satisfactory, and instead of describing it in this paper, we refer the reader to the companion paper *Wrestling with rep exposure* [Detlefs et al. 1998].

Disjoint ranges requirement. The *disjoint ranges requirement* states that pivot fields declared in distinct units have disjoint ranges. That is, if b and d are pivot fields whose declarations occur in different units, then, at any procedure boundary,

$$\langle\, \forall\, s, t \,::\; b[s] = d[t] \;\Rightarrow\; b[s] = \mathbf{nil} \,\rangle$$

where s and t range over non-**nil** objects. The requirement is enforced by rewriting pre- and postconditions.

To motivate the disjoint ranges requirement, we first recall the motivation for the top-down requirement for static dependencies. In the presence of the dependencies

$$\begin{array}{l} \textbf{depends } a[t] \textbf{ on } c[t] \\ \textbf{depends } v[t] \textbf{ on } c[t] \end{array}$$

the methodology protects related abstractions by adding the postcondition

$$v[t] = v'[t]$$

to any procedure specified with **modifies** $a[t]$. If the dependency of $v[t]$ on $c[t]$ were not visible, the translation would be unable to add this postcondition, making modular verification unsound. Soundness is achieved for static dependencies by imposing the top-down requirement.

The top-down requirement works for static dependencies, but it would be ridiculously strict to generalize it in the obvious way for dynamic dependencies. For example, it would be too strict to require that *Set.valid* be visible wherever *Seq.valid* is, since *Set* is a higher-level abstraction, which quite possibly was not envisioned when *Seq* was designed.

The disjoint ranges requirement is the analogue of the top-down requirement, but for dynamic instead of static dependencies. Consider the following variables and dependencies:

$$\begin{array}{l} \textbf{depends } a[t] \textbf{ on } c[b[t]] \\ \textbf{depends } v[t] \textbf{ on } c[d[t]] \end{array}$$

and a procedure P specified with **modifies** $a[t]$. Then P is allowed to modify $a[t]$ and $c[b[t]]$, but not $v[s]$ for any s, not even when $d[s] = b[t]$. If d is visible in the scope containing P's body, then modifies list desugaring adds an appropriate conjunct to the postcondition. But if d is not visible in that scope, the only way to guarantee that $v[s]$ is unchanged is to guarantee $d[s] \neq b[t]$, which is ensured by the disjoint ranges requirement.

Swinging pivots restriction. Consider the modifies list

$$\textbf{modifies } b[t]$$

It gives a procedure the license to modify b, but only at t. More precisely, the procedure is required to establish the postcondition

$$\langle\, \forall\, s \;::\; b_0[s] = b'[s] \;\vee\; s = t \,\rangle$$

where, in this discussion, we write b' for the final value of b and b_0 for the initial value of b.

Now consider the modifies list

$$\textbf{modifies } b[t], c[b[t]]$$

It, too, gives a procedure the license to modify b, only at t. In addition, it allows the modification of c at one point only. But is this point $b_0[t]$ or $b'[t]$? It is traditional to choose the first alternative, allowing the modification of c only at $b_0[t]$, and we follow this tradition. However, the possibility that $b_0[t]$ may be different from $b'[t]$ causes a difficulty, which we will now describe.

Consider the following artificial procedure that returns from a reader not the current character but the second character:

```
proc secondChar(rd: Rd.T): int
  requires valid[rd]
  modifies state[rd]
impl secondChar(rd: Rd.T) is
    result := rd.getChar() ;
    result := rd.getChar()
  end
```

We certainly hope that this implementation will verify: since *rd.getChar*()'s specification requires *valid*[*rd*] and modifies *state*[*rd*], that same specification should be satisfied by two calls in a row. Indeed, it does verify in a scope where only *Rd* is imported.

Unfortunately, as we discovered when using our ESC checker on the Modula-3 I/O system, the implementation does not verify if *RdRep* is imported! The problem is that, in the presence of the dependencies declared in *RdRep*, the checker will issue a spurious warning because of the possibility that the first call to *getChar* changes *buff*[*rd*] and the second call changes the contents of the new *buff*[*rd*]. Thus the net effect is inconsistent with our interpretation of *secondChar*'s specification

$$\textbf{modifies } state[rd]$$

since this desugars to

$$\textbf{modifies } buff[rd], elems[buff[rd]], \ldots$$

which allows changing $elems[buff_0[rd]]$, but not $elems[buff'[rd]]$.

Reversing the tradition does not help: if *secondChar*'s specification were desugared to allow modification of *elems* at $buff'[rd]$ instead of at $buff_0[rd]$, then the checker would warn about the possibility that the first call to *getChar* changes the contents of the buffer and the second call changes the buffer pointer.

This problem is a failure of modular soundness, which can only be rectified in two ways: by changing the proof system so that *secondChar* does not verify when only *Rd* is imported, or by changing it so that *secondChar* does verify when *RdRep* is imported. Our engineering judgment is that the latter course is the right one. The best way we have found to achieve this is to impose a rather strict requirement that we call the *swinging pivots restriction*: a procedure specified to modify a pivot field is allowed to change it only to **nil** or to a value newly allocated within the procedure. This discipline is enforced formally by adding, for each pivot field b, a conjunct to the postcondition of every procedure:

$$\langle\, \forall\, s \;::\; b_0[s] = b'[s] \;\vee\; b'[s] = \mathbf{nil} \;\vee\; \neg alloc_0[b'[s]] \,\rangle \tag{16}$$

where $alloc_0[x]$ means the object x was allocated in the pre-state (Section 8.1 provides more details about *alloc*). With this postcondition of *getChar*, the spurious warnings will not occur, since the problematic control paths are inconsistent with the strengthened postcondition.

As we have described it, the swinging pivots restriction is too strict. For example, the restriction forbids an initialization method from assigning one of its parameters to a pivot field in the object being initialized, which is occasionally necessary. It is straightforward to revise the swinging pivots restriction to accommodate such assignments, by adding a disjunct to (16), but we won't describe it in this paper since it requires the nomenclature defined in *Wrestling with rep exposure* [Detlefs et al. 1998].

We can envision situations where even the revised restriction is too strict, for example, a double-buffered reader implementation in which one buffer is being filled while the other is being emptied. But the swinging pivots restriction is the best solution to the problem that we know.

8. REASONING ABOUT TYPES AND ALLOCATION

A central issue described in this paper is the rewriting of specifications found in a modular program into specifications about which one can reason using standard techniques for verifying one-scope programs. Although those techniques have been described widely in the literature, there are some areas where we have had to innovate in order to build the Modula-3 Extended Static Checker, in particular in the areas of reasoning about types and allocation. We describe these techniques here, both because some of the techniques related to allocation are new, and because this material interacts with the modularity issues discussed in Section 9.

8.0 Reasoning about types

Conditions that the language guarantees can be assumed by the methodology without proof. We call such conditions "freeconditions". For example, in checking a procedure implementation like

$$\mathbf{impl}\ P(t\colon T,\ n\colon \mathbf{nat})\ \mathbf{is}\ \ldots\ \mathbf{end}$$

we get the free preconditions $t \neq \mathbf{nil}$ and $n \geq 0$, from the language semantics for method calls and the language definition of **nat**. The full story is more complicated, since the value of type **nat** might be a field of some object rather than a simple parameter.

Therefore, every verification condition R in a scope D is discharged under the *background predicate* for D:

$$BackgroundPred_D \;\Rightarrow\; R$$

The background predicate is a conjunction of axioms formed from the declarations that are visible. This subsection gives a flavor of what the background predicate contains.

For every type T, the background predicate contains the definition of a predicate symbol $is\$T$, which asserts that its argument is of type T. In addition, for each object type T, the background predicate contains a constant $tc\$T$ representing the *typecode* of T. If T is declared to be a subtype of an object type U, the background predicate will contain the conjunct

$$subtype1(tc\$T, tc\$U)$$

For an object type T, $is\$T$ is defined by the conjunct

$$\langle\, \forall\, t \;::\; is\$T(t) \;\equiv\; t = \mathbf{nil} \;\vee\; subtype(typecode(t), tc\$T) \,\rangle$$

where *subtype* is the reflexive, transitive closure of *subtype*1.

Data fields are treated as maps from objects to values. For every map type $T \rightarrow U$ occurring in the program, the background predicate defines a predicate symbol $field\$T\$U$. One of the axioms about $field\$T\$U$ asserts that applying a map to a value in its domain produces a value in its range:

$$\langle\, \forall\, f, t \;::\; field\$T\$U(f) \;\wedge\; is\$T(t) \;\wedge\; t \neq \mathbf{nil} \;\Rightarrow\; is\$U(f[t]) \,\rangle$$

For the full details of the background predicate of a small object-oriented language, we refer the reader to the axiomatic semantics of Ecstatic [Leino 1997]. The background predicate used for Modula-3 in the Extended Static Checker is similar, but more complicated, because, for example, Modula-3 has a larger variety of types.

8.1 Reasoning about allocation

Consider the following specification puzzle: A procedure P takes a filename as a parameter, opens the named file, reads four bytes, and returns their value as an integer. We would like to specify P with an empty modifies list, since P is essentially functional from the point of view of the client. However, it is impossible to implement P without side effects on allocated data. For example, if a file reader is used, its buffer will be changed.

Our solution is to make it implicit in the specification of every procedure that modifications to newly allocated state are allowed. Thus, although P's modifies list is empty, its implementation is allowed to change the fields of the file reader, since it allocates that reader (but if P used a pre-existing reader rd, it would have to mention $state[rd]$ in the modifies list, as usual). We say that by convention we allow "free modification of unused state". In fact, we have already used this convention: *BlankRd.init* modifies the contents of the buffer. This is allowed by our convention,

because the buffer is newly allocated, but it would have been inconsistent with the modifies list otherwise.

We believe this convention is sound with respect to the standard operational semantics, but we have neither proved it nor noticed that anyone else has.

The convention affects the desugaring of specifications. To describe this in more detail, we must explain the semantics of allocation. Since successive calls to the storage allocator return different results, it must be that the calls have some side effect. Informally, the side effect is to extend the set of allocated objects. In the formal semantics, the side effect is to change the "allocated" property of the returned object from *false* to *true*. We model this property with the predeclared boolean object field *alloc*.

The program expression **new**(T) is sugar for

$$\begin{array}{l}\textbf{var } x \textbf{ in}\\ \quad x \neq \textbf{nil} \ \ \wedge \ \ \neg alloc[x] \ \ \wedge \ \ x \in T\\ \rightarrow\\ \quad alloc[x] := true \ ; \textbf{result} := x\\ \textbf{end}\end{array}$$

that is, nondeterministically choosing any non-**nil**, unallocated object of type T, and allocating and returning it.

The modifies list of every procedure implicitly contains *alloc*, and the postcondition of every procedure implicitly includes

$$\langle\, \forall\, s \ ::\ alloc[s] \ \Rightarrow\ alloc'[s] \,\rangle$$

that is, the procedure can allocate objects, but not deallocate them (we assume the usual fiction of garbage collected languages wherein objects are allocated but never deallocated).

Recall that, for every field g, a modifies list desugars to a conjunct in the postcondition of the form

$$\langle\, \forall\, s \ ::\ g[s] = g'[s] \ \vee\ s = E_0 \ \vee\ s = E_1 \ \vee\ \ldots \,\rangle$$

where the E's are the modification points allowed for g by the modifies list. With our allocation convention, this conjunct becomes

$$\langle\, \forall\, s \ ::\ g[s] = g'[s] \ \vee\ \neg alloc[s] \ \vee\ s = E_0 \ \vee\ s = E_1 \ \vee\ \ldots \,\rangle$$

This allows the procedure to modify g at any newly allocated object.

The specification language admits assertions that quantify over all objects of a particular type. Such assertions are considered by convention to apply to allocated objects only. For example, a universal quantification $\langle\, \forall\, x\colon T \ ::\ P(x) \,\rangle$ occurring in a specification is desugared into

$$\langle\, \forall\, x\colon T \ ::\ alloc[x] \ \Rightarrow\ P(x) \,\rangle$$

except if it occurs in a postcondition, in which case it is desugared into

$$\langle\, \forall\, x\colon T \ ::\ alloc'[x] \ \Rightarrow\ P(x) \,\rangle$$

These kinds of assertions are not common in pre- or postconditions, but they are common in program invariants, which will be discussed in Section 9.3.

Unlike the mini-language used in this paper, many programming languages allow declarations to specify *default values* for object fields. These will become important when we discuss program invariants in Section 9.3. Taking default values into account, the desugaring of **new** must be altered slightly from the version given above. Suppose, for example, that f is one of T's fields, and that the default value of f is the constant C. Then $\mathbf{new}(T)$ is sugar for

$$
\begin{array}{l}
\mathbf{var}\ x\ \mathbf{in} \\
\quad x \neq \mathbf{nil} \ \wedge \ \neg alloc[x] \ \wedge \ x \in T \ \wedge \ f[x] = C \\
\rightarrow \\
\quad alloc[x] := true\ ;\ \mathbf{result} := x \\
\mathbf{end}
\end{array}
$$

This desugaring nondeterministically chooses an object whose f field has the right value. (We prefer this to an alternative desugaring which assigns $f[x] := C$ after choosing x. Our version reduces the number of assignments, which speeds mechanical checking.)

The story we have told so far about **new** is not new. For example, our story is essentially equivalent to that given by Hoare and Wirth [1973] in their classic paper on an axiomatic semantics for Pascal. We were surprised to find, when applying our checker to the Modula-3 library, that the story doesn't work. The following artificial program illustrates the problem:

```
type T, U
var f: T → U
proc P(t: T) requires t ≠ nil
impl P(t: T) is
    var u: U in
      u := new(U) ;
      assert f[t] ≠ u
    end
  end
```

Procedure P, which takes an object t as a parameter and allocates a new object u, will crash if the f field of t is u. As programmers, we know this won't ever happen, but nothing we have said so far allows this procedure to be verified. We have ensured that **new** returns a previously unallocated object, but we have not ensured that all reachable objects are allocated. This problem seems to be less appreciated than the more easily solved problem of ensuring that **new** returns a previously unallocated object.

The background predicate helps, since we can arrange that it provide the assumption $alloc[t]$ for each parameter or global variable of an object type. But as the example shows, this is not sufficient, since $alloc[f[t]]$ does not follow logically from $alloc[t]$. The basic idea of our solution is to introduce into the methodology the free assumption that fields of allocated objects are themselves allocated, that is, that for every declared field f whose range type is an object type, *alloc* is closed under f. It is not enough to assume this condition once and for all in the background predicate, since both *alloc* and f are mutable. Instead, the closure condition is an implicit pre- and postcondition of every procedure, including **new**. We will not describe

the details here, since they are not particularly relevant to modular verification. Instead, we refer interested readers to the axiomatic semantics of Ecstatic [Leino 1997].

9. FURTHER CHALLENGES

Static and dynamic dependencies allow us to check many parts of the Modula-3 run-time library that we were unable to check without them. But there remain programming paradigms that are used in practice and seem sound and modular to which our approach does not apply. This section describes some of these challenges and some tentative ideas we have for addressing them.

9.0 Cyclic dependencies

Dynamic dependencies give rise to the possibility of *cyclic dependencies*, that is, an abstract variable may depend on itself indirectly, via some pivot fields. Indeed, this happens in the case of a "filter" object that "forwards" method calls to an instance of one of its supertypes. For example, consider a $DOSRd.T$ subtype of $Rd.T$ that returns all the characters of a given child reader, but with carriage return characters filtered out:

$$\begin{array}{l}\textbf{unit } DOSRd \textbf{ import } Rd\\ \quad \textbf{type } T <: Rd.T\\ \quad \textbf{proc } init(drd\colon T,\ rd\colon Rd.T)\colon T\\ \quad\quad \textbf{requires } rd \neq \textbf{nil } \wedge\ valid[rd]\\ \quad\quad \textbf{modifies } valid[drd]\\ \quad\quad \textbf{ensures } valid'[drd]\ \wedge\ \textbf{result} = drd\end{array}$$

(For simplicity, we're ignoring *state*.) The expression $\textbf{new}(DOSRd.T).init(rd)$ allocates, initializes, and returns a new DOS reader with child reader rd. The implementation of DOS readers will need to store the child reader in some field of the DOS reader, say ch:

$$\textbf{var } ch\colon DOSRd.T \to Rd.T$$

The implementation will also have to give the representation of *svalid* for DOS readers, which will include a conjunct expressing that the child is valid:

$$\textbf{rep } svalid[drd\colon DOSRd.T] \equiv\ \ldots\ \wedge\ valid[ch[drd]]$$

This requires the dynamic dependency

$$\textbf{depends } svalid[drd\colon DOSRd.T] \textbf{ on } valid[ch[drd]]$$

Combined with the static dependency of $valid[rd]$ on $svalid[rd]$ in *RdRep*, this produces a cycle of dependencies.

To accommodate cyclic dependencies, we make two changes to our proof system. We will describe the two changes for the case that there is exactly one pivot field involved in any cycle. This is the only case that we have implemented in ESC, although we believe that the ideas could be generalized.

The first change is in taking the closure of a modifies list. We need to make some change to prevent the closure from being infinite. We introduce two new notations allowed in closures: $f^*[t]$ and $f^{-*}[t]$. Intuitively, they represent the set of terms

$$t,\ f[t],\ f[f[t]],\ \ldots$$

and the set of terms

$$t,\ f^{-1}[t],\ f^{-1}[f^{-1}[t]],\ \ldots$$

respectively. These notations appear in the closures of modifies list, but they are fictions that are eliminated when the closures are transformed into postconditions. Since we assume only one pivot field per cycle, the infinite set of terms produced by the closure rules described previously can be summarized in a finite set of terms involving the new notations. For example, in the context of the implementation of DOS readers, the modifies list

$$\textbf{modifies}\ \mathit{valid}[\mathit{drd}]$$

has the closure

$$\begin{array}{l}\mathit{valid}[\mathit{ch}^*[\mathit{drd}]],\ \mathit{svalid}[\mathit{ch}^*[\mathit{drd}]],\\ \mathit{valid}[\mathit{ch}^{-*}[\mathit{drd}]],\ \mathit{svalid}[\mathit{ch}^{-*}[\mathit{ch}^{-1}[\mathit{drd}]]]\end{array}$$

Recall that modifies lists are closed, and then closed modifies lists are turned into modification constraints in postconditions. Thus, to eliminate our new notations, we must show how to rewrite them into modification constraints. The license to modify $a[b^*[t]]$ gives rise to the postcondition contribution

$$\langle\, \forall\, s\ ::\ a[s] = a'[s]\ \vee\ t \xrightarrow[\textbf{nil}]{b} s\ \rangle$$

where the notation $t \xrightarrow[x]{b} s$, read "t reaches s via (applications of) b, not going through x", is defined by Nelson [1983]. Similarly, the license to modify $a[b^{-*}[t]]$ gives rise to the postcondition contribution

$$\langle\, \forall\, s\ ::\ a[s] = a'[s]\ \vee\ s \xrightarrow[\textbf{nil}]{b} t\ \rangle$$

The second change to our proof system is to the pointwise axiom for any abstract variable involved in a cycle of dependencies. We will describe the change by means of an example. To set the stage, we consider first an example with a dynamic but non-cyclic dependency, say

$$\begin{array}{l}\textbf{depends}\ a[t]\ \textbf{on}\ e[t]\\ \textbf{depends}\ a[t]\ \textbf{on}\ c[b[t]]\end{array}$$

The pointwise axiom for a (leaving out residues) is

$$\begin{array}{rl}\langle\, \forall\, s, e0, e1, c0, c1, b0, b1\ ::\ & e0[s] = e1[s]\ \wedge\ c0[b0[s]] = c1[b1[s]]\\ \Rightarrow\ & \mathcal{F}.a(e0, c0, b0)[s] = \mathcal{F}.a(e1, c1, b1)[s]\ \rangle\end{array}$$

Now let the dynamic dependency be cyclic:

$$\begin{array}{l}\textbf{depends}\ a[t]\ \textbf{on}\ e[t]\\ \textbf{depends}\ a[t]\ \textbf{on}\ a[b[t]]\end{array}$$

The new pointwise axiom for a (leaving out residues) is

$$\begin{array}{l}\langle\, \forall\, s, e0, e1, b0, b1\ ::\\ \qquad \langle\, \forall\, r\ ::\ s \xrightarrow[\textbf{nil}]{b0} r\ \Rightarrow\ e0[r] = e1[r]\ \wedge\ b0[r] = b1[r]\ \rangle\\ \qquad \Rightarrow\ \ \mathcal{F}.a(e0, b0)[s] = \mathcal{F}.a(e1, b1)[s]\ \rangle\end{array}$$

That is, $a[t]$'s value depends only on the e and b fields of objects reachable from t via b.

We will illustrate this pointwise axiom by showing the verification of the *init* method of DOS readers, implemented as:

impl *init*(*drd*: T, *rd*: $Rd.T$): T **is**
  $ch[drd] := rd$; $lo[drd] := 0$; ... ; **result** := *drd*
end

where we assume the elided code initializes the *cur*, *hi*, and *buff* fields of *drd* to satisfy the validity requirements given in *RdRep*. The first part of this verification is showing that the assignment to the *ch* field establishes $svalid[drd]$. This is easy since the *init* method requires $valid[rd]$ as a precondition. The second part is showing that the assignment does not affect the validity of any other reader (except as allowed by the modifies list). As we have already remarked, the closure of the modifies list includes

$$valid[ch^*[drd]],\ valid[ch^{-*}[drd]]$$

which produces the postcondition

$$\langle\, \forall\, s\ ::\ valid[s] = valid'[s]\ \vee\ drd \xrightarrow[\mathbf{nil}]{ch} s\ \vee\ s \xrightarrow[\mathbf{nil}]{ch} drd\ \rangle$$

which is functionalized to

$$\langle\, \forall\, s\ ::\ \mathcal{F}.valid(ch, lo, \ldots)[s] = \mathcal{F}.valid(ch', lo', \ldots)[s] \vee\ drd \xrightarrow[\mathbf{nil}]{ch} s\ \vee\ s \xrightarrow[\mathbf{nil}]{ch} drd\ \rangle$$

which follows from the pointwise axiom for *valid*, which is

$$\begin{aligned}&\langle\, \forall\, s, ch0, ch1, lo0, lo1, \ldots\ ::\\ &\quad \langle\, \forall\, r\ ::\ s \xrightarrow[\mathbf{nil}]{ch0} r\ \Rightarrow\ ch0[r] = ch1[r]\ \wedge\ lo0[r] = lo1[r]\ \wedge\ \ldots\ \rangle\\ &\quad \Rightarrow\\ &\quad \mathcal{F}.valid(ch0, lo0, \ldots)[s] = \mathcal{F}.valid(ch1, lo1, \ldots)[s]\ \rangle\end{aligned}$$

We leave the proof to the reader.

In the verification of the *init* method of DOS readers, no properties of the reachability predicate were used: it might as well have been an uninterpreted predicate. Properties of the reachability predicate come into play when verifying a non-trivial operation on the DOS reader whose implementation modifies the child reader (for example the *refill* method, which recursively invokes the *refill* method of the child).

In summary, we have described the essential ideas of a proof system for cyclic dependencies. More details are described by Rajeev Joshi [1997]. At least two problems still remain: Cyclic dependencies with more than one pivot field per cycle require some generalization. Also, even with just one pivot field per cycle, our rewriting produces verification conditions that are beyond the limit of what our automatic theorem prover can handle efficiently.

9.1 Yet more dependencies

We have concentrated on static and dynamic dependencies because they play a central role in the patterns of abstractions in the library programs we took as test

cases in the ESC project, not because we can't imagine other kinds of dependencies. In this section, we sketch what we know about other dependencies.

If a global abstract variable (not a field) depends on a global concrete variable (not a field), we call it an *entire dependency*. For example,

$$\begin{array}{l}\textbf{spec var } k\colon \textbf{int}\\ \textbf{var } m, n\colon \textbf{nat}\\ \textbf{rep } k \equiv m - n\\ \textbf{depends } k \textbf{ on } m, n\end{array}$$

This kind of abstraction occurs frequently in papers on data refinement, but in practice we have found static and dynamic dependencies far more frequent. One place in which entire dependencies are useful is in reasoning about module initialization, which we will address in Section 9.2. We have a soundness theorem for entire dependencies, and the modularity requirements are essentially the same as those for static dependencies, that is, the dependency of a on c must be placed in the unit that declares c [Leino 1995].

If an abstract field (not a global variable) depends on a global concrete variable (not a field), we have a dependency of the form

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

As an example of how this might come up, consider an abstract type whose instances contain unique id fields. Each id field is initialized from a global counter, *gcount*. This might well lead to a representation of validity of the form

$$\textbf{rep } \mathit{valid}[t] \equiv \ \ldots \ \wedge \ \mathit{id}[t] < \mathit{gcount}$$

which in turn would require a dependency of the form

$$\textbf{depends } \mathit{valid}[t] \textbf{ on } \mathit{gcount}$$

However, the soundness of these dependencies is problematical, and our current view is that they are not useful and should be forbidden. To specify a data type containing unique identifiers, we recommend using program invariants, as will be described in Section 9.3.

If an abstract field depends on concrete fields of the elements of an array, we have an *array dependency*. We would suggest overloading the notation for dynamic dependencies: if $b[t]$ has type $\textbf{array}[U]$ and c is a field with index type U, then

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

is an array dependency that allows $a[t]$ to depend on the sequence of values

$$c[b[t][0]],\ c[b[t][1]],\ \ldots$$

So an array dependency seems akin to a dynamic dependency, but with an array of pivots instead of just one.

As an example of an array dependency, consider a type T representing sets of elements of type E. Suppose that the implementation automatically enlarges itself when necessary, and that enlarging requires rehashing the current elements, and that rehashing an element requires that the element be valid. Then, the validity of

the set will require the validity of all its elements. If the elements are held in an array, say b, then the validity of the set will have the form

$$\begin{aligned}&\textbf{rep } T.valid[t\colon T] \equiv \\ &\qquad \ldots \wedge \langle \forall\, i :: 0 \le i < \textbf{number}(b[t]) \Rightarrow E.valid[b[t][i]] \rangle\end{aligned}$$

which involves the array dependency

$$\textbf{depends } T.valid[t\colon T] \textbf{ on } E.valid[b[t]]$$

We suspect that array dependencies are a straightforward generalization of dynamic dependencies, but we have not investigated them thoroughly.

One can imagine many other kinds of dependencies, for example,

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

But we have never been able to make a strong case that such dependencies are useful.

9.2 Checking initialization order

Initializing global data is more complicated in a multi-module program than in a single-module program and is a common source of programming errors. Some of the procedures in a module require that the module's globals be initialized, but generally not all of them: for example, any procedure that is used in performing the initialization. Thus, there are two classes of procedures: those that require prior initialization of the module and those that don't. A common error is to inadvertently call a procedure of the first class before initialization is complete, either through confusion over which class a procedure is in, or because the linker initializes the modules in an unexpected order.

We suggest that data abstraction can help in solving this problem. The idea is to introduce into the interface of each module a boolean abstract variable, called an *init variable*, which means the module has been initialized. Procedures of the first class require the init variable as a precondition, while those of the second class do not. The purpose of a module body is to ensure an init variable as a postcondition; to achieve this, it may call other procedures that modify and ensure the init variable.

A programmer can also require one or more init variables of other modules as preconditions of the module body. The linker calls the module bodies in an order such that each body's precondition is established before it is called, or reports a cycle if this is impossible. Each module provides a **rep** declaration that connects its init variable to the globals of the module, so occurrences of init variables in specifications are desugared like any other abstract variable.

An init variable generally depends on the global variables in the module. These dependencies satisfy the modularity requirements for entire dependencies since they are placed in the same units as the declarations of the globals, and thus present no problem to modular verification. An init variable may also depend on other init variables, since

$$\textbf{rep } \mathit{Minit} \equiv \mathit{Ninit} \wedge \ldots$$

where *Minit* and *Ninit* are the init variables of two modules M and N, is a simple way of giving M's procedures of the first class the right to call N's procedures of

```
unit M
  type T
  spec var a: T → ...
    ⋮
  proc P(t: T) modifies a[t]
unit N
  type U
  spec var c: U → ...
    ⋮
  proc R(u: U) modifies c[u]
unit MImpl import M, N
  var b: T → U
  depends a[t: T] on c[b[t]]
    ⋮
  impl P(t: T) is ... R(b[t]) ... end
```

Fig. 10. A prototypical example involving a dynamic dependency.

the first class. Unfortunately, the dependency of *Minit* on *Ninit* is most naturally placed in the implementation of module M, a unit that declares neither *Minit* nor *Ninit*. Thus, this dependency violates both the visibility and top-down requirements, which in general destroys soundness. We have several ideas for restoring soundness while allowing init variables to depend on one another. These ideas are based on the observation that init variables change only from *false* to *true*. But we have not proved a soundness theorem.

9.3 Invariants

In practice, almost all pivot fields are injective (one-to-one), that is, if b is a pivot field and u and v are distinct objects in the domain of b, then $b[u]$ and $b[v]$ are distinct (or they are both **nil**). The reason for this is easily seen by considering the prototypical example involving a dynamic dependency, shown in Figure 10. The call to R from P modifies $c[b[t]]$. This affects the value of $a[t]$. If the pivot field b were not injective, it would also affect $a[u]$ for any u such that $b[u] = b[t]$. In general, when $a[t]$ is modified by changing part of its representation $c[b[t]]$, the only hope for showing that the modification obeys the modifies list

$$\textbf{modifies } a[t]$$

is to require the injectivity of b.

Note that although we find injectivity necessary to be able to verify interesting programs, we have not found injectivity to be a requirement for soundness.

By the way, it is surprisingly difficult to verify a procedure that initializes an injective field. While showing that a command like

$$b[t] := \textbf{new}(U)$$

maintains the injectivity of b is easy, a command like

$$b[t] := \mathit{NewU}()$$

does not verify, even if procedure *NewU* is specified to ensure $\neg alloc[\mathbf{result}] \wedge alloc'[\mathbf{result}]$. The checker dreams up the possibility that *NewU* allocates a new U object, squirrels it away into some b field, and then returns it. To cope with this problem, we enrich the specification language with the expression $virgin[x]$, which means that x is not, and has never been, the value of any object field or global variable. The details are found in a paper by Leino and Stata [1999b].

How should a programmer use the specification language to record the design decision that a field is to be injective? One might first try to include this as part of the representation of an object's validity, producing a **rep** declaration like

$$\mathbf{rep}\ valid[t\colon T] \equiv \ \ldots \ \wedge\ (b[t] = \mathbf{nil} \vee \langle\, \forall\, s\colon T \ ::\ s \neq t \ \Rightarrow\ b[s] \neq b[t] \,\rangle)$$

But this seems problematical. It makes $valid[t]$ depend not just on $b[t]$, but on $b[s]$ for all s of the appropriate type. It seems perverse to think of this unbounded collection of $b[s]$'s to be part of the "representation" of $valid[t]$.

A simpler and better strategy is to extend the specification language with the notion of a *program invariant*: a declaration of the form

$$\mathbf{invariant}\ J$$

records the intention that the predicate J hold at every procedure call and return in the entire program. For example, to specify the injectivity of b, the following program invariant can be used:

$$\begin{aligned}\mathbf{invariant}\ \langle\, \forall\, t, u\colon T \ ::\ & t \neq \mathbf{nil} \ \wedge\ u \neq \mathbf{nil} \ \wedge\ t \neq u \\ & \Rightarrow\ b[t] \neq b[u] \ \vee\ b[t] = \mathbf{nil} \,\rangle\end{aligned}$$

The methodology enforces program invariants with two requirements. First, it requires that J be true at the "beginning of time". Second, it requires that every procedure respect J (assuming that all the procedures it calls respect J), that is, it conjoins J to the pre- and postcondition of every procedure implementation and procedure call.

The beginning-of-time test is straightforward and presents no modularity problems. It consists of the following proof obligation for each declared program invariant J:

$$\langle\, \forall\, t \ ::\ t = \mathbf{nil} \ \vee\ \neg alloc[t] \,\rangle \ \Rightarrow\ J$$

That is, J must hold in a state in which no non-**nil** objects have been allocated. More precisely, this proof obligation must follow from the background predicate. If J contains free variables of primitive types like integers, then it must hold regardless of their values. To enforce invariants about global variables, the init-vars technique described in Section 9.2 is more useful. In our experience, we mostly use program invariants to assert universally quantified properties of objects of a certain type, like injectivity. In this case, the beginning-of-time test passes trivially.

The second test, that every procedure respects J, involves subtle modularity issues. The basic idea is simple: when in a scope D the methodology desugars a specification (either in reasoning about a procedure call or in checking a procedure implementation), it adds to the pre- and postcondition all invariants whose declarations are in D. If the program consists of a single global scope, then the soundness of this approach is clear: the change to the pre- and postconditions is the

same for reasoning about the calls as for checking the implementations. However, if the program consists of many scopes, then modularity requirements must be imposed to achieve soundness, by ensuring that primitive steps in a scope where the invariant is not visible cannot falsify the invariant. We will build up to the correct modularity requirements in stages. To begin with, we assume that the invariant contains concrete variables only.

The first modularity requirement for invariants that comes to mind is:

> a program invariant must be declared near all of its free variables.

Two declarations are *near* one another if they are contained in the same unit. It follows that they are visible in the same scopes.

This simple modularity requirement achieves soundness because an invariant cannot be falsified except by modifying its free variables. Thus, those procedures whose implementation lies outside the scope of the invariant preserve the invariant because they cannot mention any of its free variables. The rest of the procedures are proved to maintain the invariant.

Unfortunately, this simple requirement is too strong because of the special concrete variable *alloc*, which represents the set of allocated objects and occurs implicitly in almost all invariants: recall from Section 8.1 that a quantification

$$\langle \forall\, t\colon T \;::\; \ldots \rangle$$

is desugared to

$$\langle \forall\, t\colon T \;::\; \mathit{alloc}[t] \;\Rightarrow\; \ldots \rangle$$

Consequently, it is necessary to loosen the simple rule to allow program invariants to mention *alloc*. This introduces the danger of a procedure falsifying an invariant invisible to it by modifying *alloc*. We address this difficulty by observing that the only way a procedure can directly modify *alloc* is by performing an allocation, and we can demand of an invariant that it be maintained by any allocation in any portion of the program in which it is not visible. To this end, we say that an invariant J passes the *blind allocation test* for a type T if J is invariant under $\mathbf{new}(T)$.

This brings us to the second version of the modularity requirement for invariants:

(0) a program invariant must be declared near all of its free concrete variables, except *alloc*, and

(1) for all types T, either (a) T is declared near the invariant, or (b) the invariant passes the blind allocation test for T, or (c) T is not mentioned in the invariant.

Here's a sketch of a justification for this version of the modularity requirement: Because of (0), the only invariant-falsifying primitive steps that we need to worry about are those that modify *alloc*, that is, expressions of the form $\mathbf{new}(T)$ for some type T. But it is impossible for the expression $\mathbf{new}(T)$ to falsify the invariant, because for such a T, neither (a) nor (b) nor (c) could hold: not (a), since if T is declared near the invariant, the invariant is visible wherever $\mathbf{new}(T)$ can be called; not (b), since the blind allocation test explicitly checks that $\mathbf{new}(T)$ maintains the invariant; and not (c), since $\mathbf{new}(T)$ cannot falsify the invariant if the invariant doesn't mention T and passes the blind allocation test for T.

```
unit U
  type T
  spec var valid: T → bool
  proc init(t: T): T
    modifies valid[t]
    ensures valid'[t] ∧ result = t
unit UImpl import U
  var id: T → int
  ... (other fields) ...
  rep valid[t: T] ≡ ...
  var gcount: int
  impl init(t: T): T is
      id[t] := gcount ; gcount := gcount + 1
      ...
      result := t
    end
```

Fig. 11. An example module that uses unique identifiers.

In order to pass the blind allocation test, a programmer must choose appropriate default values for the fields of an object type. For example, if a pivot is specified to be injective, its default value should be **nil**.

Let us return to a problem that we touched on in Section 9.1, namely the problem of declaring a data type containing unique identifiers, see Figure 11. To record the design decisions about *id* and *gcount*, one can add to *UImpl* the program invariants:

$$\begin{aligned}&\textbf{invariant } \langle\, \forall\, t\colon T \;::\; t \neq \textbf{nil} \;\wedge\; valid[t] \;\Rightarrow\; id[t] < gcount \,\rangle\\ &\textbf{invariant } \langle\, \forall\, t, u\colon T \;::\\ &\qquad t \neq \textbf{nil} \;\wedge\; u \neq \textbf{nil} \;\wedge\; valid[t] \;\wedge\; valid[u] \;\wedge\; t \neq u\\ &\qquad \Rightarrow\; id[t] \neq id[u] \,\rangle\end{aligned}$$

In this approach, the statements about *id* and *gcount* that were problematical to place in the **rep** declaration (see Section 9.1) have been moved into program invariants. The **rep** declaration for $valid[t]$ concerns only fields of t. This seems an improvement, but this approach still has two problems: one is giving the public *init* method the license to modify the private variable *gcount*, the other is allowing the abstract variable *valid* to appear in a program invariant.

To solve the first problem, we can introduce an abstract variable, say *istate* for internal state, in the interface U:

$$\textbf{spec var } istate\colon \textbf{any}$$

We then allow *init* to modify *istate*, but *istate* has no other occurrences in the interface:

$$\begin{aligned}&\textbf{proc } init(t\colon T)\colon T\\ &\quad \textbf{modifies } valid[t], istate\\ &\quad \textbf{ensures } valid'[t] \;\wedge\; \textbf{result} = t\end{aligned}$$

Finally, we add the entire dependency of *istate* on *gcount* to the module *UImpl*:

$$\textbf{depends } istate \textbf{ on } gcount$$

which by downward closure gives *init* the license to modify *gcount*.

The second problem is that the invariants mention $valid[t]$, but so far we have considered invariants containing concrete variables only. We cannot just eliminate the occurrences of $valid[t]$, since no default value for id will make the second invariant pass the blind allocation test for T. The blind allocation test is needed, since T is mentioned in the invariant but T and the invariant are not declared near one another.

One way to solve the second problem is to allow abstract variables in program invariants. We believe that it is sound to do so, provided that the invariant satisfies (0) and (1) from above, and also, for each abstract variable a appearing in the invariant:

(2) all dependencies of a are static, and

(3) either (a) the invariant is declared near a, or (b) the invariant is declared near every **rep** declaration for a and near every dependency of a.

However, this story is getting more complicated than we like. Perhaps it is best simply to forbid abstract variables from appearing in program invariants. If we do, we need some other way of dealing with the occurrences of $valid[t]$ in the program invariants in the unique identifiers example. This we can do simply by inlining them, that is, by replacing $valid[t]$ by whatever expression is given as its rep. Although awkward, this entails no loss of modularity or information hiding, since the invariants occur in a scope (*UImpl*) where the representation of $valid[t]$ is visible.

10. IMPLEMENTATION STATUS

Almost everything described in this paper has been implemented in the Modula-3 Extended Static Checker. Exceptions are:

(0) the checker implements only the individual residues, not the shared residue *sres* described in Section 6 beginning on page 29,

(1) the checker does not enforce the dependency segregation restriction of Section 7.1 on page 39, but instead uses a more general way of computing the dynamic closure ("upward closure of dynamic predecessors"), which does not necessitate the restriction,

(2) the checker does not enforce the disjoint ranges requirement of Section 7.2 (and as mentioned in that section, we leave it to the programmer to avoid abstract aliasing), and

(3) the checker does not implement the initialization order checking of Section 9.2.

Our experience with the checker is described in more detail in our companion paper [Detlefs et al. 1998]. We have applied the checker to thousands of lines of code, both from the Modula-3 libraries and from programs that use the libraries. In specifying the libraries, we constantly used static and dynamic dependencies.

After experimenting with our Modula-3 checker, we embarked on another project to build an extended static checker for Java [Extended Static Checking for Java ; Leino et al. 2000]. In the ESC/Java project, we circumvented most of the difficulties described in this paper by omitting data abstraction from the annotation language. To partially make up for the omission, we provide object invariants [Leino and

Stata 1997] and ghost variables, but the fundamental basis of our decision was to accept less thorough checking in order to produce a simpler checker. For example, without any feature for abstraction (like abstract variables presented here or like data groups [Leino 1998a]), it is not possible to soundly specify and verify modifies lists.

11. RELATED WORK

Most work on data abstraction seems to be directed at one of two goals: algorithm design or structuring large systems.

When data abstraction is used for algorithm design, the representation is "inlined" into the site of use as the refinement step of the design [Back 1980; Gries and Prins 1985; Lamport and Schneider 1985; Jones 1986; Morris 1989; Gries and Volpano 1990; Gardiner and Morgan 1993]. Consequently, the work on this kind of data abstraction is largely unconnected with the large system structuring problems that we are concerned with in this paper. This is not to deny that the underlying mathematics of data abstraction applies to both enterprises. Indeed, our first verification condition generator did not use explicit functionalization of abstract variables but instead used the "change of coordinates" approach common in algorithm refinement. However, we found that the result was that our theorem-prover was constantly forced to apply the "one-point rule" and that for our purposes, explicit functionalization is preferred.

Turning to data abstraction for the purpose of structuring large systems, the earliest treatments were in contexts where there was no independent information-hiding mechanism (like our units) and therefore the problems addressed in the present paper did not arise, or were ignored in the semi-formal treatments in the literature. These treatments include Milner's definition of simulation [Milner 1971], Hoare's classic treatment of abstraction functions [Hoare 1972], and the influential work of Liskov and Guttag and the rest of the CLU community [Liskov and Guttag 1986].

The first programming language to support information hiding in the way our units do was Mesa [Mitchell et al. 1979], with its definition modules and implementation modules. The Mesa designers appear to have been influenced by Parnas's classic paper on decomposing systems into modules [Parnas 1972]. Mesa in turn influenced Modula [Wirth 1977], Modula-2 [Wirth 1982], Modula-3 [Nelson 1991], Oberon-2 [Mössenböck and Wirth 1991], and Ada [American National Standards Institute, Inc. 1983]. Ernst et al. [1994] have studied the problem of specifying Modula-2 programs where the objects of a module may share some global state. These authors share our concern for modular verification, but the possible scopes they consider are not rich enough to allow subtypes or the *RdRep* interface of our example.

Another, rather different, approach of hiding information is to classify declarations as public or private. This approach is used in Oberon [Wirth 1988], C++ [Ellis and Stroustrup 1990], and Java [Gosling et al. 1996]. In the course of the ESC/Java project [Extended Static Checking for Java ; Leino et al. 2000], we used the modularity requirements of the units approach to guide our design for visibility of invariants in the public/private approach [Leino and Stata 1997].

One of the central ideas of this paper, explicit dependency declarations, were introduced in Leino's PhD thesis in 1995 [Leino 1995]. Between that time and this, they have been applied in a number of contexts: they played a central role in ESC for Modula-3 [Detlefs et al. 1998], and they were incorporated in the specification languages JML [Leavens et al. 1999] and Larch/C++ [Leavens 1996] and in the programming logic of Müller and Poetzsch-Heffter [2000]. Indeed, as we mentioned at the end of Section 6.4, the PhD thesis of Müller [2001] features a simpler formalization of dependencies than ours. The formalization also covers a broader range of dependencies, including dynamic dependencies, for which Müller provides a soundness proof, provided the program uses the Universe Type System [Müller and Poetzsch-Heffter 2001]. Another application (or reformulation) of dependency declarations is Leino's technique of Data Groups [1998a].

As described in Section 7.2, our first attempt at a solution to the problem of abstract aliasing [Detlefs et al. 1998] is not fully satisfactory. We do find that our framework of modular soundness and dynamic dependencies has allowed us to give a more incisive definition of the problem than other approaches in the literature, such as Hogg's Islands [1991], Almeida's Balloons [1997], Utting's Extended Local Stores [1995], the Flexible Aliasing Protection of Noble et al. [1998], and Boyland's Alias Burying [2001]. Perhaps the most promising approach to this problem is the Universe Type System of Müller and Poetzsch-Heffter [2001], which was also designed in conjunction with abstraction dependencies. We would also like to mention a recent proposal with some promise, the "owner exclusion requirement", invented by Leino and Stata [1999a] and named by Arnd Poetzsch-Heffter.

A few other researchers have employed declarations similar to our **depends** declaration connecting an abstract variable to the (more) concrete variables in its representation. Daniel Jackson's Aspect system [Jackson 1995] features dependencies much like ours, but his motivation seems to be to avoid the need for reasoning about the details of the actual representation, whereas we have argued that dependency declarations are useful also in conjunction with full representation declarations. The COLD specification language of Jonkers [1991] includes abstract variables (called functions) and dependency declarations between them, but COLD seems not to allow an abstract variable to appear in a modifies list, so it doesn't address many of the problems we have wrestled with.

12. CONCLUSIONS

We have applied precise formal methods to systems programs that are typical examples of the programming techniques used by careful and experienced contemporary programmers. We found that the formal methods described in the verification literature are inadequate to deal with the patterns of data abstraction and modularization in these programs. We have developed new formal methods to address these shortcomings.

Central to the new methods is the concept of an *abstraction dependency*, which is a kind of abstraction of an abstraction function, in the same sense that an opaque type is an abstraction of a concrete type. A dependency specifies one or more of the variables that occur in an abstraction function, but hides the detailed definition of the function. Just as an opaque type may be widely visible in a multi-module program, while the corresponding concrete type may be visible only narrowly, we

discovered that it is often useful to make a dependency more widely visible than the abstraction function itself.

Different kinds of abstraction dependencies occur in different styles of design. Top-down programming leads to static dependencies, where an abstract field of an object is represented in terms of other fields of that same object. Bottom-up programming with reusable libraries leads to dynamic dependencies, where an abstract field of an object is represented in terms of fields of other objects, reachable indirectly from the first object.

We have shown how to verify programs in the presence of static and dynamic dependencies by rewriting modifies lists, preconditions, and postconditions.

For static dependencies, we have two simple *modularity requirements*, which are laws for the placement of dependency declarations in a multi-module program. The requirements do not seem to preclude any useful designs, and we have a formal proof of modular soundness for the requirements. The formal proof makes use of our identification of modular soundness with the monotonicity of verifiability with respect to scope. For dynamic dependencies, we have several modularity requirements, but no soundness theorem, nor any confidence that the list of requirements is complete.

In our experience with static checking of contemporary program libraries, we have found that we use dependencies constantly in our annotations. We have also found that dependencies provide a new perspective on old problems like the problem of encapsulation and rep exposure.

ACKNOWLEDGMENTS

At several points in the paper, we have remarked that implementing our ideas in a realistic program checker was critical to many of our discoveries. Here we will remark that Dave Detlefs was critical: he wrote the majority of the code, and he was often the first to see the methodological implications of practical issues.

In Section 6, we attributed the subtle example program to Jim Saxe's penetrating intelligence; we also would like to thank him for his help with cyclic dependencies and other thorny problems.

Every project owes a debt to its devil's advocates, and to our ESC project, Mark Lillibridge contributed both valuable scepticism and helpful ideas.

Rajeev Joshi worked with us as a research intern, and resolved a problem in our initial approach to cyclic dependencies.

We also thank Raymie Stata, who shared his methodological wisdom in many helpful discussions, and the anonymous referees for their many helpful comments.

Finally, we thank our colleagues who commented on earlier drafts of this paper: Dave Detlefs, Cynthia Hibbard, Mark Lillibridge, Annabelle McIver, Carroll Morgan, Raymie Stata, and Mark Vandevoorde.

REFERENCES

Aho, A. V., Sethi, R., and Ullman, J. D. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley.

Almeida, P. S. 1997. Balloon types: Controlling sharing of state in data types. In *ECOOP'97—Object-oriented Programming: 11th European Conference*, M. Akşit and S. Matsuoka, Eds. Lecture Notes in Computer Science, vol. 1241. Springer, 32–59.

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

Back, R. J. R. 1980. *Correctness Preserving Program Refinements: Proof Theory and Applications*. Mathematical Centre Tracts, vol. 131. Mathematical Centre, Amsterdam.

Boyland, J. 2001. Alias burying: Unique variables without destructive reads. *Softw. Pract. Exper. 31,* 6 (May), 533–553.

Detlefs, D. L., Leino, K. R. M., and Nelson, G. 1998. Wrestling with rep exposure. Research Report 156, Digital Equipment Corporation Systems Research Center. July.

Detlefs, D. L., Leino, K. R. M., Nelson, G., and Saxe, J. B. 1998. Extended static checking. Research Report 159, Compaq Systems Research Center. Dec.

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

Ernst, G. W., Hookway, R. J., and Ogden, W. F. 1994. Modular verification of data abstractions with shared realizations. *IEEE Trans. Softw. Eng. 20,* 4 (Apr.), 288–307.

Extended Static Checking for Java. Extended Static Checking for Java home page. On the web at `http://research.compaq.com/SRC/esc/`.

Extended Static Checking for Modula-3. Extended Static Checking for Modula-3 home page. On the web at `http://research.compaq.com/SRC/esc/EscModula3.html`.

Gardiner, P. H. B. and Morgan, C. 1993. A single complete rule for data refinement. *Formal Aspects of Computing 5,* 4, 367–382.

Gosling, J., Joy, B., and Steele, G. 1996. *The Java™ Language Specification*. Addison-Wesley.

Gries, D. and Prins, J. 1985. A new notion of encapsulation. In *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*. Number 7 in SIGPLAN Notices 20. ACM, 131–139.

Gries, D. and Volpano, D. 1990. The transform — a new language construct. *Structured Programming 11,* 1, 1–10.

Hoare, C. A. R. 1972. Proof of correctness of data representations. *Acta Inf. 1,* 4, 271–81.

Hoare, C. A. R. and Wirth, N. 1973. An axiomatic definition of the programming language PASCAL. *Acta Inf. 2,* 4, 335–355.

Hogg, J. 1991. Islands: Aliasing protection in object-oriented languages. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '91)*, A. Paepcke, Ed. Number 11 in SIGPLAN Notices 26. ACM, 271–285.

Jackson, D. 1995. Aspect: Detecting bugs with abstract dependences. *ACM Transactions on Software Engineering and Methodology 4,* 2 (Apr.), 109–145.

Jones, C. B. 1986. *Systematic Software Development using VDM*. International Series in Computer Science. Prentice-Hall, Englewood Cliffs, NJ.

Jonkers, H. B. M. 1991. Upgrading the pre- and postcondition technique. In *VDM '91 Formal Software Development Methods: 4th International Symposium of VDM Europe. Volume 1: Conference Contributions*, S. Prehn and W. J. Toetenel, Eds. Lecture Notes in Computer Science, vol. 551. Springer-Verlag, 428–456.

Joshi, R. 1997. Extended static checking of programs with cyclic dependencies. In *1997 SRC Summer Intern Projects*, J. Mason, Ed. Technical Note 1997-028. Digital Equipment Corporation Systems Research Center.

Lamport, L. and Schneider, F. B. 1985. Constraints: A uniform approach to aliasing and typing. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*. ACM, 205–216.

Leavens, G. T. 1996. An overview of Larch/C++: Behavioral specifications for C++ modules. In *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, H. Kilov and W. Harvey, Eds. Kluwer Academic Publishers, Chapter 8, 121–142.

Leavens, G. T., Baker, A. L., and Ruby, C. 1999. Preliminary design of JML: A behavioral interface specification language for Java. Tech. Rep. 98-06f, Iowa State University, Department of Computer Science. July. Available at `ftp://ftp.cs.iastate.edu/pub/techreports/TR98-06/`.

Leino, K. R. M. 1995. Toward reliable modular programs. Ph.D. thesis, California Institute of Technology. Technical Report Caltech-CS-TR-95-03.

Leino, K. R. M. 1997. Ecstatic: An object-oriented programming language with an axiomatic semantics. In *The Fourth International Workshop on Foundations of Object-Oriented Languages*. Proceedings available from `http://www.cs.williams.edu/~kim/FOOL/`.

Leino, K. R. M. 1998a. Data groups: Specifying the modification of extended state. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '98)*. Number 10 in SIGPLAN Notices 33. ACM, 144–153.

Leino, K. R. M. 1998b. Recursive object types in a logic of object-oriented programs. *Nordic Journal of Computing 5,* 4 (Winter), 330–360.

Leino, K. R. M. and Nelson, G. 2000. Data abstraction and information hiding. Research Report 160, Compaq Systems Research Center. Nov.

Leino, K. R. M., Nelson, G., and Saxe, J. B. 2000. ESC/Java user's manual. Technical Note 2000-002, Compaq Systems Research Center. Oct.

Leino, K. R. M. and Stata, R. 1997. Checking object invariants. Technical Note 1997-007, Digital Equipment Corporation Systems Research Center. Jan.

Leino, K. R. M. and Stata, R. 1999a. Smothering rep exposure with reads clauses. Internal manuscript KRML 90, Compaq Systems Research Center.

Leino, K. R. M. and Stata, R. 1999b. Virginity: A contribution to the specification of object-oriented software. *Inf. Process. Lett. 70,* 2 (Apr.), 99–105.

Liskov, B. and Guttag, J. 1986. *Abstraction and Specification in Program Development*. MIT Electrical Engineering and Computer Science Series. MIT Press.

Milner, R. 1971. An algebraic definition of simulation between programs. Tech. Rep. Stanford Artificial Intelligence Project Memo AIM-142, Computer Science Department Report No. CS-205, Stanford University. Feb.

Mitchell, J. G., Maybury, W., and Sweet, R. 1979. The Mesa language manual, version 5.0. Tech. Rep. CSL-79-3, Xerox PARC. Apr.

Morris, J. M. 1989. Laws of data refinement. *Acta Inf. 26,* 4 (Feb.), 287–308.

Mössenböck, H. and Wirth, N. 1991. The programming language Oberon-2. *Structured Programming 12,* 4, 179–195.

Müller, P. 2001. Modular specification and verification of object-oriented programs. Ph.D. thesis, FernUniversität Hagen. Available from `http://www.informatik.fernuni-hagen.de/pi5/publications.html`.

Müller, P. and Poetzsch-Heffter, A. 2000. Modular specification and verification techniques for object-oriented software components. In *Foundations of Component-Based Systems*, G. T. Leavens and M. Sitaraman, Eds. Cambridge University Press, Chapter 7, 137–159.

Müller, P. and Poetzsch-Heffter, A. 2001. Universes: A type system for alias and dependency control. Tech. Rep. 279, FernUniversität Hagen. Available from `http://www.informatik.fernuni-hagen.de/pi5/publications.html`.

Nelson, G. 1983. Verifying reachability invariants of linked structures. In *Conference Record of the Tenth Annual ACM Symposium on Principles of Programming Languages*. ACM, 38–47.

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

Noble, J., Vitek, J., and Potter, J. 1998. Flexible alias protection. In *ECOOP'98—Object-oriented Programming: 12th European Conference*, E. Jul, Ed. Lecture Notes in Computer Science, vol. 1445. Springer, 158–185.

Parnas, D. L. 1972. On the criteria to be used in decomposing systems into modules. *Commun. ACM 15,* 12 (Dec.), 1053–1058. Reprinted as `http://www.acm.org/classics/may96/`.

Stoy, J. E. and Strachey, C. 1972. OS6—an experimental operating system for a small computer. Part II: Input/output and filing system. *The Computer Journal 15,* 3, 195–203.

Utting, M. 1995. Reasoning about aliasing. In *Proceedings of the Fourth Australasian Refinement Workshop (ARW-95)*. School of Computer Science and Engineering, The University of New South Wales, 195–211.

Wirth, N. 1977. Modula: a language for modular multiprogramming. *Softw. Pract. Exper. 7,* 1 (Jan.–Mar.), 3–35.

Wirth, N. 1982. *Programming in Modula-2.* Springer-Verlag.

Wirth, N. 1988. The programming language Oberon. *Softw. Pract. Exper. 18,* 7 (July), 671–690.

Received 29 November 2000. Accepted 4 June 2001.