

**SOFTWARE VERIFICATION RESEARCH CENTRE  
DEPARTMENT OF COMPUTER SCIENCE  
THE UNIVERSITY OF QUEENSLAND**

**Queensland 4072  
Australia**

**TECHNICAL REPORT**

**No. 94-38**

**Supporting module reuse in refinement**

**Ian Hayes**

**September 1994**

**Phone: +61 7 365 1003  
Fax: +61 7 365 1533**

# Supporting module reuse in refinement

Ian J. Hayes\*

September 23, 1994

## Abstract

For the rigorous development of larger-scale software, the use of data abstractions is essential. For formal development in methods such as VDM, Z and the refinement calculus such abstractions are encapsulated in a module with state and operations. The principle of information hiding suggests that the state of a module should be inaccessible to the rest of a program. As an approach to refinement where an existing module is to be reused, we recommend relaxing the principle of information hiding to allow the program being developed to access the *abstract* state of the module directly. Eventually all references to the module state must be replaced by calls to module operations, but in the initial stages of refinement only allowing module operations is too restrictive. By allowing access to the module state the development method may support the reuse of existing modules more easily.

## Introduction

The principle of information hiding [4] —not allowing external access to the internal state of a module— is important in developing well-structured programs because it allows a program to be developed using abstract data structures. These structures may not be directly supported by the programming language but are easier to reason about than the data structures provided by the programming language. Information hiding allows data refinement to be used to change the internal representation of the state of a module to allow more efficient implementation of the module's operations. The module interface specification provides a firewall between the user of the module — who may assume only the properties specified in the interface but not those additional properties of a particular implementation— and the implementer of the interface — who may choose to implement the interface how they see fit without any specific knowledge of how the module is to be used (provided the use conforms to the specification).

In [3, Chapter 17] Morgan considers program development for the case where there are no existing modules to be reused. The approach to developing a program that will

---

\*Department of Computer Science, University of Queensland, Brisbane, Queensland 4072, Australia. E-mail: Ian.Hayes@cs.uq.oz.au.

eventually contain a module is to begin by introducing the state of the module-to-be as part of the program's state. The program is then refined. During this process procedures are introduced to manipulate the state of the module-to-be. In the initial stages of the refinement the module's state is used directly by the program, but as the refinement proceeds access to the module's state is localised to a set of procedures whose purpose is to manipulate the module's state. Once all references to the module state are localised to these procedures, a new module can be created and the state and procedures can be encapsulated within the module.

This is an excellent approach to take to developing a new module from scratch. One starts with the abstract state and during the program refinement process introduces new procedures as necessary. However, it does not suit the situation where one would like to reuse an existing module, or where one has a clear idea of the interface of a desired module *a priori*. In the latter case one may have a range of implementation strategies in mind for the module. Such considerations can affect the choice of operations provided by the interface.

## Module interface reuse

It is quite common for the interface of a module to be already defined (often because one wishes to reuse an existing module) before refinement of a program that uses the module takes place. The aim of this paper is to introduce an approach to handling refinement of programs in such cases.

The approach is simply to allow the program to reference the *abstract* state of the module directly. The program may include statements (typically specification statements [2, 3]) that examine and update the abstract state of the module. The motivation behind this is that it is easier to think about manipulating the abstract state directly rather than via the module's operations, especially if the module's operations are quite low level.

In this stage of the refinement process we say that the module interface is *open*. The program is then refined with the goal of removing the direct references to the module's (abstract) state, replacing them with calls to the operations provided by the module. Once this goal is reached, the abstract state of the module may be hidden: the module interface may be *closed*. Only at this stage is it valid to use a data refinement of the module as an implementation of it. As with any development, during the refinement process one must be aware of the capabilities of the module one is using in order to avoid performing refinement steps that lead up a blind alley.

To some, our approach may seem obvious, but to others who take information hiding as an inviolable principle of software design, our approach may seem heretical. However, our approach is consistent with the information-hiding principle once the module interface has been closed. The significance of encapsulating the representation used in a module is that it allows data refinement of the representation. Our approach allows this, but only after the module has been closed.

**An example** Consider a program fragment that is required to swap the values associated with two symbols stored in a symbol table module. The symbol table is represented abstractly as a finite partial function from symbols of type  $S$  to values of type  $V$ ,

$$st : S \dashrightarrow V,$$

with operations to look up the value associated with a symbol in the table,  $LookUp(s,v)$ , and to replace (or add) the value associated with a symbol,  $Replace(s,v)$ .

We specify the fragment as a specification statement which has the form ' $v : [pre, post]$ ' where  $v$  is the statement's frame (a list of variables that may be modified by the statement),  $pre$  is its pre-condition and  $post$  is its post-condition. To specify a swap of the values of symbols  $x$  and  $y$  in the symbol table, we need to refer to the symbol table state:

$$st : [\{x, y\} \subseteq \text{dom } st, st = st_0 \oplus \{x \mapsto st_0(y), y \mapsto st_0(x)\}]$$

where in the post-condition  $st_0$  stands for the value of the symbol table before the operation,  $st$  for the value after, and ' $\oplus$ ' for function overriding.

This specification statement may be refined to a sequence of operations by introducing local variables  $vx$  and  $vy$  to hold the values stored in the table for  $x$  and  $y$ . During this process direct reference to the module state is required.

```

var  $vx, vy$  •
   $vx : [x \in \text{dom } st, vx = st(x)];$ 
   $vy : [y \in \text{dom } st, vy = st(y)];$ 
   $st : [true, st = st_0 \oplus \{x \mapsto vy\}];$ 
   $st : [true, st = st_0 \oplus \{y \mapsto vx\}]$ 

```

Because we are reusing an existing module we need to realise that our target program can only make use of the operations provided by the module. This motivates the structure we have derived so far. The specification statements in this program can now be replaced by equivalent module operation calls:

```

var  $vx, vy$  •
   $LookUp(x, vx);$ 
   $LookUp(y, vy);$ 
   $Replace(x, vy);$ 
   $Replace(y, vx)$ 

```

This is a simple example for illustrative purposes. Assuming the symbol table module is under one's control, one could contemplate the alternative approach of adding a swap operation to the symbol table module. However, this approach does not scale up to more complex operations. By adding more operations or, alternatively, more parameters to an existing operation, one risks the 'solution' of introducing a baroque module interface with many operations with multiple parameters, not all of which are relevant to any one application. This can make the module interface more difficult

to understand, and hence make reuse more difficult. Further, more of the program development problem is moved inside the module. This can lead to an unbalanced partitioning of the program development problem between the module and the rest of the program.

A better approach is to avoid adding operations (or additional parameters) to a module for the sake of a single application, and instead to design the module interface to provide the primitives from which more complex operations can be built. It is for constructing such complex operations that the approach suggested by this paper is most relevant. The more complex the operation the harder it is to do without the approach —an indication of its usefulness in scaling up refinement methods.

**A second example** Consider the development of an algorithm to manipulate a tree structure. To specify the algorithm, one needs to refer to the abstract state of the tree. It is possible that one will want to make use of an existing tree module because, for example, other parts of the program may be using the tree module to set up and perform other operations on the tree.

Consider developing an algorithm that requires multiple traversals of the tree. For example, it may be necessary to calculate the maximum height of the tree in one traversal before performing other manipulations of the tree in a second traversal. In the initial refinement steps, the specification can be transformed to a level in which each complete traversal of the tree by the algorithm is given only as a single specification statement operating on the abstract tree as a whole. In general, the particular tasks performed by the traversals are specific to the problem at hand and are best not considered primitives to be provided by the tree module. Hence the specification of each traversal as a specification statement requires reference to the abstract state of the tree module. The next stage of the development is to refine each of the traversals until all references to the tree are via the operations provided by the tree module.

These examples illustrate that the approach suggested does seem well-suited to handling refinement of more complex operations.

## Reasoning

In state-based specification and refinement methods, used for example with VDM [1], Z [5] and the refinement calculus [3], the specification of a module includes its (abstract) state. The state is visible to the person reading the specification and provides a basis for understanding the behaviour of the module's operations. The state is also visible when it comes to reasoning about use of the module. The convention usually followed is that the person can see the abstract state in order to reason about the module, but is constrained not to refer directly to the module state within the program they are developing. Our proposal is to relax this restriction during program development. The program may refer directly to the state during refinement —the program is not

yet *code*<sup>1</sup>— but the references must be eliminated eventually by replacing them with module operation calls. The endpoint of this development phase satisfies the usual convention that the program does not directly refer to the state of the module. The module can be closed off, and only at this stage can it be replaced by a data refinement.

A possible alternative approach to reasoning about a program using a module is to prove properties about the externally visible behaviour of the module's operations as theorems within the module. The theorems can include properties of sequences of operation calls (including the parameters to the calls). Although a proof of a theorem may reference the module's state, the statement of the theorem should have no explicit references to the state.

It is easy to see how such theorems can be devised for simple or systematic sequences of calls, but for more complex applications the sequence of calls may be quite complicated and intimately interwoven with the order in which the program calls module's operations. In these cases reasoning about the sequences is tantamount to reasoning about the particular application program and its use of the abstract module.

It would appear to be a mistake to consider all such reasoning to be internal to the module. It is more appropriate to consider it reasoning about the abstract program that uses the module. On the other hand, if certain types of simple sequences of calls occur frequently within many applications, then it would be useful to prove properties of such sequences just once within the bounds of the module.

## Conclusions

A simple approach to the rigorous development of software that reuses existing modules is initially to suspend the principle of information hiding and allow the program to refer to the abstract state of the module during the early stages of the development process. The module interface is open at this stage. The goal of the development is to remove the explicit references to the module's abstract state. Once this goal is attained the module interface may be closed. Only at this stage does the principle of information hiding come into play in order to allow changes of representation of the module's state via data refinement.

Our approach allows access to the abstract state of a module during the initial stages of the refinement of a program that uses the module, but at no stage does it allow external access to the state associated with data refinements of the module. All data refinements of the state remain internal to the module and may not be accessed directly by the remainder of the program.

The approach allows one to write programs that use the abstract structures of modules directly in the early stages of the program development process. This is essential if the program to be developed is complex and the operations of the module that is

---

<sup>1</sup>Following the convention used in [3] we use the term *code* to refer to programs that can be compiled and executed. The term *program* is more general, while including code, it is also used for programs that may contain (non-executable) specification statements, or other non-executable constructs.

being reused are relatively low level. Overall the approach allows more straightforward refinement of programs that reuse existing module interfaces.

**Acknowledgements** This paper was motivated by discussions with Cliff Jones, John Fitzgerald and Luke Wildman while I was visiting the University of Manchester. I would like to acknowledge the hospitality of the Department of Computer Science there and the financial assistance of the Special Studies Program of the University of Queensland. My thanks go to David Carrington, Peter Kilpatrick and Peter Lindsay for fruitful discussions on earlier drafts of this paper.

## References

- [1] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International, second edition, 1990.
- [2] Carroll Morgan. The specification statement. *ACM Transactions on Programming Languages and Systems*, 10(3), July 1988.
- [3] Carroll Morgan. *Programming from Specifications*. Prentice Hall, 1990.
- [4] D. Parnas. On the criteria to be used in decomposing systems into modules. *CACM*, 15(2):1053–58, 1972.
- [5] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice Hall International, second edition, 1992.