An Inspection-based Technique for Verifying Module Correctness

by

Graeme Neal Jones
B. Sc., University of Alberta, 1987

A thesis submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Dr. D. M. Hoffman, Supervisor (Dept. of Computer Science)

[redacted]

Dr. W. W. Wadge, Departmental Member (Dept. of Computer Science)

[redacted]

Dr. C. G. Morgan, Outside Member (Dept. of Philosophy)

[redacted]

Dr. G. F. McLean, External Examiner (Dept. of Mechanical Engineering)

Supervisor: Dr. D. M. Hoffman

ABSTRACT

Current trends in programming methodology suggest that specification and verification should be as formal as possible. Yet, putting such methods into practice has proved difficult. We propose a mixture of formal and informal techniques, and rely more on human review than on formal derivation. We extend current work on module verification, providing new theoretical results as well as practical procedures designed for use in software inspections.

Our underlying theory is for pairs of infinite, nondeterministic Mealy machines. *Module state machines* (MSMs) are Mealy machines specialized for modeling software modules: the inputs are function calls and the outputs are return-value/exception pairs. We have defined three kinds of MSMs, corresponding to (1) declarative specifications using abstract state, (2) declarative specifications using concrete state, and (3) implementations. Both exceptions and nondeterminism are handled. Based on the Mealy machine theory, we have developed MSM verification procedures, specifically designed for proofs delivered in inspection meetings. Detailed examples of the three MSMs and the verification procedures are included.

Examiners:

[redacted]

Dr. D. M. Hoffman, Supervisor (Dept. of Computer Science)

[redacted]

Dr. W. W. Wadge, Departmental Member (Dept. of Computer Science)

[redacted]

Dr. C. G. Morgan, Outside Member (Dept. of Philosophy)

[redacted]

Dr. G. F. McLean, External Examiner (Dept. of Mechanical Engineering)

Table of Contents

List of Figures

Acknowledgment

I would like to thank my supervisor, Dr. Daniel Hoffman, for his guidance and advice during the course of this research, and for his help in the preparation of this manuscript. Financial support in the form of an NSERC Postgraduate Scholarship and a research assistantship from Dr. Daniel Hoffman made this work possible. Finally, I would like to thank my wife, Mary Jo. I could not have completed this thesis without her support, encouragement, and patience.

Chapter 1

Introduction

1.1 Problem

The task of developing a large software system can be simplified by decomposing it into *modules*. We view a module as a black box, accessible only through a fixed set of *access programs*. Three work products are associated with every module: *interface specification*, *program specification*, and *implementation*. These work products provide a progression from black-box specification to executable implementation.

An *interface specification* describes the service offered by a module without unnecessarily restricting the means used to provide that service. For most modules, some means for referring to "state" or "history" is needed. We use an abstract state representation to make the interface specification as simple and precise as possible. The *program specification* declares the concrete state, using the constructs of the implementation programming language. The effects of each of the access programs are specified declaratively, in terms of the concrete state and in accordance with the interface specification. The *implementation* provides the executable code used to operate on the concrete state, as required by the program specification.

Users of a module expect it to behave according to the interface specification. The implementation determines actual module behaviour; therefore, it is important that the implementation be correct with respect to the interface specification. Consider a module with interface specification I, program specification P, and implementation C. The task of proving that C meets I is simplified by proving that C meets P and that P meets I, as the two subproofs can be carried out independently. Each subproof typically involves induction on the length of traces (sequences of calls to access programs). We present techniques which can be used to eliminate the induction argument from correctness proofs. We are still left with the problem of how to conduct verification in industry.

Despite substantial effort, formal specification and verification research has had disturbingly little impact on industrial practice. This is due in part to an assumption in the prevalent Computer Science view of formal verification: formal verification will guarantee program correctness, just as mathematical proofs guarantee the correctness of theorems. According to DeMillo, et al. [1], there are two underlying problems. First, formal program proofs do not guarantee correctness. For non-trivial programs, such proofs are extremely complex and are thus error-prone themselves. Further, the assumptions on which the proofs are based often cannot be formally verified. Second, the appeal to mathematical practice is flawed. Mathematicians rarely resort to formal proofs, for good reason. According to Karp [2, pg. 1411]

> Once a mathematical truth is discovered, the 'proofs' used to communicate it from one human to another are never formalized completely, if only because such formalization is a hideously tedious process and because the resulting formal proofs would be opaque to human readers.

So, instead of relying on formal proofs, mathematicians use *informal* proofs,

and wide and repeated review by other mathematicians. It is the social process as much as the proof on which they base their confidence in the truth of the theorems.

The result of the academic insistence on formal specification and verification is a crippling stand-off between the "formalists" and the software developers. Few developers, even in academia, take formal methods seriously. In practice, even informal methods of specification and verification are rejected, leaving software developers struggling to control complex systems without the most basic mathematical support.

In summary, this thesis attempts to solve three problems associated with the verification of software modules:

1. What is module correctness and how can we define it precisely?

2. How can module correctness proofs be simplified?

3. How can module correctness be proven in practice?

The techniques that we use to solve these problems are described briefly in the next section.

1.2 Techniques

We have developed the theory of modules by viewing the three work products as *Mealy machines*. We define correctness in terms of the *language* of a Mealy machine, which captures the input/output behaviour of the machine.

Proving correctness typically involves induction on the length of traces. At first glance, one might suggest that the proof could be simplified by proving each access program correct using classical program verification techniques [3]. This poses special problems, however, because the interface spec-

ification and the program specification use different state spaces. Interface specification state variables are usually expressed in terms of abstract concepts, such as sets and sequences, while program specification state variables are expressed using the data types supported by the implementation language, such as arrays. Classical program verification involves proving a correspondence between pairs of functions defined on the same state space, and therefore must be extended to handle module verification.

We achieve the extension with *abstraction functions*. An abstraction function maps each state of the program specification to a state of the interface specification. We present a theorem which defines conditions on the abstraction function, program specification, and interface specification that are sufficient to conclude that the program specification meets the interface specification. Our theorem greatly simplifies module correctness proofs by factoring out the induction argument from these proofs, and by breaking up each proof into many smaller proofs which can be carried out independently of each other. The induction argument that would normally be required in these proofs. A corollary of the theorem applies when the concrete Mealy machine and the abstract Mealy machine have identical state spaces; in this case, the abstraction function is an identity function. This corollary can be used to prove that an implementation is correct with respect to a program specification. It is interesting that the conditions of the corollary can be established using classical program verification techniques [3]. Thus, program verification seems to be a special case of module verification.

We use pragmatic criteria for specification and proof, based on the following claim: *formality is not a suitable engineering goal.* While it is a powerful means for achieving engineering goals, such as reliability, it is dangerously inappropriate as an end in itself. There is no inherent value to the customer

in formality, only in the other characteristics that may, or may not, be best achieved by formal methods. Viewing formality as a means, rather than a goal, has influenced our criteria for proofs, as follows.

We use Fagan inspections [4], reviewing each work product in detail to identify faults, and using selected verification techniques [3, 5] in the inspection process. Consider a specification S, its implementation I, and a proof that I meets S. Evaluate the proof according to whether it convinces the inspection team. As a result, the quality of proofs will depend on the skill, experience, and discipline of the inspection team. But this is unavoidable — determining the soundness of a proof will always involve human judgement.

From the viewpoint of DeMillo, et al., we are using inspections to transform program verification: from a formal exercise whose goal is guaranteeing correctness, to a social process aimed at discovering faults and increasing confidence. The effect of this view on our research has been substantial: the focus is on methods within the grasp of industrial inspection teams, given modest training.

1.3 Results

Mealy machines have proven to be an excellent mathematical model for software modules. They have provided precise definitions of the interface specification, program specification, and implementation work products. We have adapted existing definitions of behavioural equivalence between Mealy machines to define our desired notion of module correctness. Similarly, using Mealy machines as the basis for modules allowed us to work out the requirements of abstraction functions while ignoring the details of how Mealy machines are specified in practice.

The verification procedures which we have developed are simple and easy

to follow in inspection meetings. Most of the assertions to be proven are obvious, focusing attention on the few that are not. We have found that a convincing proof can be presented to an inspection team verbally, with the help of diagrams. Obviously, care is required when using pictures in proofs. For example, it is sometimes easy to omit certain cases and to ignore important details. We use picture notations that have a precise semantics, such as those proposed by Gries [6].

1.4 Thesis organization

Chapter 2 defines key terms and concepts relating to modules and the three module work products, and defines notations that we use in specifications and proofs. Chapter 3 summarizes the related work. Chapter 4 presents our underlying theory, based on Mealy machines. Chapter 5 describes *module state machines*: Mealy machines whose inputs are access program calls and whose outputs are return-value/exception pairs. Chapter 6 presents verification procedures designed for use in inspection meetings, and Chapter 7 provides concrete examples of these procedures. Chapters 6 and 7 focus on proofs that a program specification is correct with respect to an interface specification. Chapter 8 presents our conclusions and future work.

Chapter 2

Key Concepts and Terminology

In this chapter, we define various concepts, terms, and notations used in subsequent chapters. Notations used in interface and program specifications are described in Section 2.1. Key terms relating to modules and the three work products that we associate with each module are defined in Section 2.2. Finally, the notations that we use to represent both interface and program specification states in the example proofs of Chapter 7 are described in Section 2.3.

2.1 Notation

A *concurrent assignment statement* [3, page 48] is an expression of the form

$$v_1, v_2, \ldots, v_n := e_1, e_2, \ldots, e_n$$

where the v_i's are distinct variables and each e_i is an expression of the same type as v_i. To evaluate the assignment statement, all of $e_1, e_2, \ldots, e_n$ are first evaluated, and then e_i is assigned to v_i, for $i \in [1, n]$. For example, the statement $x, y := y, x$ exchanges the values of x and y.

A *conditional rule* [3, page 29] is an expression of the form

$$(c_1 \rightarrow r_1 \mid c_2 \rightarrow r_2 \mid \ldots \mid c_n \rightarrow r_n)$$

where, for $i \in [1, n]$, c_i is a boolean expression and r_i may be any expression. To evaluate a conditional rule C, evaluate c_1; if it is true then r_1 is the value of C. Otherwise, evaluate c_2; if it is true then r_2 is the value of C, and so on. Where no condition is true, C is undefined. The condition *true* is always true, and therefore should only occur as the last condition of a rule. The expression I denotes an identity function, the domain of which should be clear from the context of the rule. An example of a conditional rule is $(x \leq y \rightarrow x \mid x \geq y \rightarrow y)$, which defines $min(x, y)$. Note that the conditions in a rule may overlap.

We make use of several simple notations on sequences. Let s be a sequence of n elements, indexed zero-relative, and let $i, j \in [0, n-1]$, with $i \leq j$. Then $s[i]$ is the ith element of s, $s[i..j]$ is the sequence $\langle s[i], s[i+1], \ldots, s[j] \rangle$, and $x \in s$ is equivalent to $(\exists j \in [0, n-1])(x = s[j])$.

2.2 Modules and work products

Following Parnas [7], a *module* is a programming work assignment, and a *module interface* is the set of assumptions that programmers using the module are permitted to make about its behavior. We view a module as a black box, accessible only through a fixed set of *access programs*. The module implementation is required to detect the occurrence of an illegal call and to signal the caller that the associated *exception* has occurred. By convention, in access program names the prefix *s_* (set) indicates calls that set internal module values, *g_* (get) indicates calls that retrieve those values, and *sg_* (set-get) indicates calls that do both.

Program name	Inputs	Outputs	Exceptions
s_init			
s_add	integer		*mem* *full*
s_del	integer		*notmem*
g_mem	integer	boolean	

Figure 2.1: *sset* access programs

We illustrate these ideas on the Simple Set (*sset*) module (see Figure 2.1), which provides access to a set of at most N integers. *s_init* initializes the module, with the set empty, and must be called before any other call. *s_add*(x) adds x to the set, signaling *mem* if x is in the set and *full* if the set has N elements. *s_del*(x) deletes x, signaling *notmem* if x is not in the set. *g_mem*(x) returns true or false according to whether x is in the set.

This thesis focuses on three work products, providing a progression from black-box specification to executable implementation.

1. An *interface specification* specifies the service offered without unnecessarily restricting the means used to provide that service. For most modules, including *sset*, some means for referring to "state" or "history" is needed. Here, the state representation is abstract, chosen to make the interface specification as simple and precise as possible.

2. The *program specification* declares the concrete state, using the constructs of the implementation programming language. The effects of each of the access programs are specified declaratively, in terms of the concrete state and in accordance with the interface specification.

3. The *implementation* provides the executable code used to operate on

the concrete state, as required by the program specification.

Our definition of module correctness is based on the notion of *behavioural equivalence* in automata theory. Shields [8] says two Mealy machines are behaviourally equivalent if, for each sequence of inputs and initial state, both machines produce the same sequence of outputs.

2.3 Representing module states in proofs

This section describes the various notations used to represent interface and program specification states in the module correctness proofs presented in Chapter 7. A state consisting of two or more variables can be represented conveniently as a tuple. For example, the *sset* program specification state consists of two variables, s and $scnt$. The elements of the set are stored in s, an array of N integers; the index of the first and last elements in s are 0 and $N-1$, respectively. $scnt$ is the number of elements in the set. Elements of the set are stored in positions 0 through $scnt-1$. A particular state can be represented as a tuple $\langle s, scnt \rangle$. This notation works well when it is not necessary to enumerate the values of the array s.

Arrays are frequently used in program specifications to store a collection of values for fast access. There are several notations for representing arrays. Since arrays are just (fixed-length) sequences, we could use an expression of the form $\langle a_0, a_1, \ldots a_n \rangle$. Such expressions can be used to represent arrays consisting of a small number of elements, but become unmanageable for larger arrays.

Gries [6] proposes notations that work well for representing arrays of any size. Arrays are represented as rectangles, arranged vertically or horizontally, with elements and indices labelled when convenient. For example, Figure 2.2

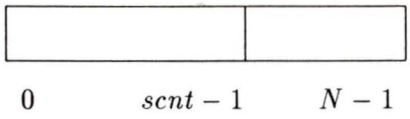


Figure 2.2: *sset* program specification state

shows the *sset* program specification state. The array s is represented as a rectangle with indices 0, $scnt - 1$, and $N - 1$ labelled.

Although the state notations presented in this section are effective in the correctness proofs presented in Chapter 7, other notations would be needed to represent more complex data structures such as linked lists.

2.4 Summary

In this chapter, we defined several notations that we use to specify the behaviour of access programs: the concurrent assignment statement, the conditional rule, and some simple notations on sequences. We presented some basic terms relating to modules and the three module work products on which this thesis focusses. Finally, we described the notations that we use to represent module states in our proof examples. In the next chapter, we compare our approach to module verification, which we outlined in Chapter 1, with approaches proposed by other authors.

Chapter 3

Comparison to Previous Work

We review the related work on module refinement, focusing on two issues. First, what class of modules is supported? In particular, are non-determinism, exceptions, and set-get calls handled? Second, what mathematical foundations are provided? Specifically, we seek precise answers to the following questions.

(i) *Semantics.* What is the meaning of a specification? What observable behavior does it specify?

(ii) *Definition of correctness.* What does it mean for a refinement to be correct? What correspondence is required between the observable behaviors of the abstract and concrete specifications?

(iii) *Sufficient conditions.* What are sufficient conditions for correctness?

(iv) *Correctness theorem.* Where is a proof that the conditions of (iii) are in fact sufficient?

The MSM approach is based directly on the *theory of modules* (TM) developed by Gannon, Hamlet, and Mills [5]. Below we discuss the TM proposal in detail and briefly describe other schemes for refinement.

3.1 Theory of modules

Within the *Cleanroom* approach [9] are methods for systems analysis, statistical testing, and verification. Only the latter is relevant to this paper. Using Cleanroom, specifications are based on state functions and relations [10, 11], and verification is done in inspection meetings [12].

Let us consider how the TM scheme answers each of the four questions presented in the overview of this chapter. The semantics of a specification consists of the following parts:

- a set S of states
- two access programs *in* and *out*

  $in : T_1 \times T_2 \times \ldots \times T_n \rightarrow S$

  $out : S \rightarrow T_1 \times T_2 \times \ldots \times T_n$

  where $T_i, i \in [1, n]$, is a type defined outside the module
- zero or more set programs $p_1, \ldots, p_k$ where $p_i : S \rightarrow S$, $i \in [1, k]$

Below we represent a TM specification compactly as a tuple of the form $\langle S, in, out, p_1, \ldots, p_k \rangle$.

Each state is a mapping from identifiers to values. In this discussion, identifiers appear in `typewriter` font. The following are examples of states:

$$\{\} \quad \{\langle \texttt{abc}, 2 \rangle\} \quad \{\langle \texttt{x}, -1 \rangle, \langle \texttt{y}, 3 \rangle\}$$

If s is a state and x is an identifier, then $s(x)$ is the value that s associates with x. Module users call *in* to load the module state, invoke set calls to modify that state, and call *out* to retrieve it.

Partial abstract and concrete specifications of a module providing operations on rational numbers are presented in the TM paper. In the remainder of this section, we shall refer to this module as *rat*. Components of the

abstract and concrete specifications are distinguished with the superscripts "a" and "c", respectively. The authors are unclear about the nature of both the abstract states and the concrete states of *rat*, and do not define the *in* and *out* functions for either the abstract or concrete specifications. To gain insight into TM specifications, we define these missing components.

Abstract states of *rat* contain exactly two elements, one element for each of the two identifers `r` and `n`. Values associated with `r` are rational numbers; values associated with `n` are integers. In other words, elements of S^a are subsets of

$$(\{\mathtt{r}\} \times Q) \cup (\{\mathtt{n}\} \times Z)$$

where Q is the set of rational numbers and Z is the set of integers.

We define in^a and out^a as follows:

$$in^a : Z \times Z \times Z \rightarrow S$$
$$in^a(x, y, z) = (y \neq 0 \rightarrow \{\langle \mathtt{r}, \tfrac{x}{y} \rangle, \langle \mathtt{n}, z \rangle\})$$

$$out^a : S \rightarrow Z \times Z \times Z$$
$$out^a(s) = \langle ax, y, z \rangle \text{ where}$$
$$x \text{ and } y \text{ are the smallest non-negative integers}$$
$$\text{such that } \mid s(\mathtt{r}) \mid = \tfrac{x}{y}$$
$$a = (s(\mathtt{r}) > 0 \rightarrow 1 \mid true \rightarrow -1) \text{ and}$$
$$z = s(\mathtt{n})$$

Some comments about these definitions of in^a and out^a are in order. First, the process of constructing these definitions was not trivial, and helped to clarify the nature of the abstract states of *rat*. In view of this, the authors of the TM approach probably should have included these definitions in their paper. Second, it is unlikely that implementations of in^a and out^a for *rat*

would ever be needed; however, a set program similar to in^a with the third argument dropped and a get program similar to out^a with the third component of the result dropped *would* be required in order to be able to use the module. The purpose of in^a and out^a is simply to establish correctness according the definition of correctness presented in the TM paper. We present this definition later in this section.

rat provides a single set program, *exprat*, which raises the rational number `r` to the power `n`. The following is the abstract specification of *exprat* as presented in the TM paper:

$$(\mathtt{n} > 1 \rightarrow \mathtt{r} := \mathtt{r}^{\mathtt{n}} \mid true \rightarrow I\ )$$

We created the following examples to illustrate the specifications in^a, out^a, and *exprat*:

$$in(1,2,3) = \{\langle \mathtt{r}, \tfrac{1}{2}\rangle, \langle \mathtt{n}, 3\rangle\}$$
$$in \circ out(1,-2,3) = \langle -1,2,3\rangle$$
$$in \circ exprat \circ out(1,-2,3) = \langle -1,8,3\rangle$$

Correctness is defined in terms of pairs of function compositions of the form

$$in^a \circ p_1^a \circ p_2^a \circ \cdots \circ p_n^a \circ out^a$$
$$in^c \circ p_1^c \circ p_2^c \circ \cdots \circ p_n^c \circ out^c$$

The refinement is correct if, for all such pairs, out^a and out^c return the same values.

Sufficient conditions for correctness are presented, based on an abstraction function and commuting diagrams, and a correctness theorem and proof are provided. A concrete specification $\langle S^c, in^c, out^c, p_1^c, \ldots p_k^c\rangle$, is correct with

respect to an abstract specification $\langle S^a, in^a, out^a, p_1^a, \ldots p_k^a \rangle$ if there exists an abstraction function $\mathcal{A}$ such that

1. $in^a \subseteq in^c \circ \mathcal{A}$

2. $\mathcal{A} \circ out^a \subseteq out^c$

3. for all $i \in [1, n]$, $\mathcal{A} \circ p_i^a \subseteq p_i^c \circ \mathcal{A}$

The strengths of the TM approach are its intuitive appeal, the effective use of state functions, and the thorough mathematical basis. The TM approach has several limitations which the MSM approach removes. First, the requirement for the *in* and *out* access programs is a severe limitation. Many modules simply do not have a call that loads the state and another to dump it. More often, the state is loaded and retrieved a portion at a time. The TM approach works reasonably well for the rational number example used in the TM paper. However, *sset*, *iset*, and many other modules are not handled.

Second, it would seem that no two set programs could have any parameter names in common, since there would be no way to load the values of these parameters simultaneously by *in*. The authors say little about how their approach would be applied to modules containing multiple access programs.

Third, the TM approach is unable to handle non-determinism, exceptions, set-get programs, and multiple get programs (only one get program, *out*, is allowed). Perhaps get programs could be modeled by assigning outputs to pseudo-variables which are returned by *out*, but again the authors do not say anything about this.

Finally, there are no verification procedures to handle the large number and variety of cases that occur in practice. These cases arise primarily from the details which were not considered in the TM paper: the three differ-

ent types of access programs, normal versus exceptional behaviour, varying degrees of nondeterminism, and so on.

The MSM approach removes all of these limitations.

3.2 Other approaches

The automata concepts of Section 6 were well established in the 1960's. For example, ignoring exceptions, the proof obligations of our verification procedures correspond exactly to the *transition homomorphism* on transducers described by Nelson [13, page 157].

Within the programming methodology community, Hoare [14] is usually given credit for the key ideas of data refinement. Following Hoare, many others have used the same basic ideas. For example, in their undergraduate text Wulf, et al. [15, page 316] use commuting diagrams to prove correctness, and include a module very similar to *iset* [15, page 519].

More recently, there has been considerable activity in the VDM and Z communities. Jones [16] has had extensive experience with data refinement (which he calls data *reification*). The sufficient conditions are precisely specified [16, page 190] and non-determinism and exceptions are handled. However, there is no explicit definition of correctness and no correctness theorem.

He, Hoare, and Sanders [17] provide methods for verification and derivation of refinements, and handle both non-determinism and exceptions. Again, precise sufficient conditions are provided but without a definition of correctness or a correctness theorem. Their scheme is very general and applicable to problems other than module refinement. However, this generality, and the requirement for a mechanism similar to the TM *in* and *out* routines, makes it hard to see how to apply the methods to conventional abstract data types. The paper provides no examples.

Chapter 4

Mealy machines

Infinite, nondeterministic Mealy machines serve as the foundation for our specification semantics and verification procedures. Definitions and a correctness theorem are presented below, in two versions: deterministic and nondeterministic. We present the deterministic theory first because it is far simpler to grasp. We present the nondeterministic version because it is needed for many proofs, including one in Section 7.2.

4.1 Deterministic machines

4.1.1 Definitions

An infinite, deterministic Mealy machine [18] is a 6-tuple $\langle Q, \Sigma, \Delta, \lambda, \delta, q_0 \rangle$ where Q is a set of states, Σ is the input alphabet, Δ is the output alphabet, $\lambda : Q \times \Sigma \rightarrow \Delta$ is the output function, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, and $q_0 \in Q$ is the initial state. Each of Q, Σ, and Δ may be infinite. Both λ and δ are defined on all of $Q \times \Sigma$.

The following example illustrates this definition. A *mod 2 recognizer* is a Mealy machine that outputs y if the number of ones received so far is even; otherwise, it outputs n. We can define a two-state mod 2 recognizer as

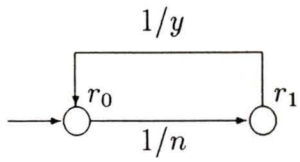


Figure 4.1: Mod 2 recognizer

follows.

$$Q = \{r_0, r_1\}$$
$$\Sigma = \{1\}$$
$$\Delta = \{y, n\}$$
$$\lambda(r_i, 1) = (i \text{ is even} \rightarrow n \mid true \rightarrow y)$$
$$\delta(r_i, 1) = r_{(i+1) \bmod 2}$$
$$q_0 = r_0$$

Note that we have used a conditional rule (defined in Chapter 2) to define the function λ. Henceforth, we shall refer to this machine as R^a.

A *transition diagram* is a graphical representation of a Mealy machine. Each diagram consists of an arrangment of nodes and arrows between nodes. Nodes correspond to states of the machine. There is an arrow from node q to node r labelled x/y if and only if $\delta(q, x) = r$ and $\lambda(q, x) = y$. q is called the *tail* of the arrow and r is called the *head*. The initial state is the head of the arrow with no tail. The transition diagram for R^a is shown in Figure 4.1.

We define correctness in terms of a language associated with a Mealy machine. Strings in the language are sequences of input/output pairs. More precisely, if $M = \langle Q, \Sigma, \Delta, \lambda, \delta, q_0 \rangle$ then strings in the language of M are of the form $w = \langle \langle x_1, y_1 \rangle, \ldots, \langle x_n, y_n \rangle \rangle$ where $x_i \in \Sigma \wedge y_i \in \Delta$ for all $i \in [1, n]$.

To define the language of a Mealy machine, we use the notion of a computation on a machine. Suppose $w = \langle\langle x_1, y_1\rangle, \ldots, \langle x_n, y_n\rangle\rangle$. A term of the form $\langle q_0, \langle x_1, y_1\rangle, q_1, \ldots, \langle x_n, y_n\rangle, q_n\rangle$ is a *computation of* w *on* M if $n \geq 0$ and

$$(\forall i \in [1, n])(x_i \in \Sigma \wedge y_i = \lambda(q_{i-1}, x_i) \wedge q_i = \delta(q_{i-1}, x_i))$$

For example, on R^a, $\langle r_0\rangle$ is a computation of the empty string $\langle\rangle$ and

$$\langle r_0, \langle 1, n\rangle, r_1, \langle 1, y\rangle, r_0\rangle$$

is a computation of the string $\langle\langle 1, n\rangle, \langle 1, y\rangle\rangle$.

The language of M, $L(M)$, is $\{w \mid$ there is a computation of w on $M\}$. It is easily shown that the language of R^a is

$$\{\langle\langle 1, y_1\rangle, \ldots, \langle 1, y_n\rangle\rangle \mid (\forall i)(i \in [1, n] \rightarrow y_i = (i \text{ is even } \rightarrow y \mid true \rightarrow n))\}$$

The language of a Mealy machine differs from the usual notion in automata theory, where "language" means the set of strings that leave a finite-state machine in one of several designated final states. If Σ is the input alphabet of such a machine, then the language accepted by the machine is a subset of $\Sigma^\star$. The language of a Mealy machine, on the other hand, is a subset of $(\Sigma \times \Delta)^\star$. To understand our definition of correctness, it is crucial to understand that the language of a Mealy machine consists of sequences of input/output pairs, not sequences of inputs alone.

Given machines M^a (abstract) and M^c (concrete), M^c is correct with respect to M^a if $L(M^c) = L(M^a)$. Proving that $L(M^c) = L(M^a)$ will typically involve induction on computations, each involving multiple applications of λ and δ. The following theorem shows how to establish correctness by considering only single applications of λ and δ. The theorem depends on an *abstraction function*: a mapping from M^c's state space to M^a's.

4.1.2 Theorem

Theorem 1 Let $M^a = \langle Q^a, \Sigma, \Delta, \lambda^a, \delta^a, q_0^a \rangle$ and $M^c = \langle Q^c, \Sigma, \Delta, \lambda^c, \delta^c, q_0^c \rangle$ be Mealy machines. If there exists a total function $\mathcal{A} : Q^c \rightarrow Q^a$ such that

1. *initial correctness*
   $q_0^a = \mathcal{A}(q_0^c)$
2. *output correctness*
   $\lambda^a(\mathcal{A}(q^c), x) = \lambda^c(q^c, x)$
3. *transition correctness*
   $\delta^a(\mathcal{A}(q^c), x) = \mathcal{A}(\delta^c(q^c, x))$

then $L(M^c) = L(M^a)$.

In this theorem, it is the hypotheses that deserve careful attention. The first one says, intuitively, that M^a and M^c are synchronized initially. Figure 4.2a uses a commuting diagram to illustrate the meaning of output correctness: from concrete state q^c, if you abstract and then apply λ^a, you get the same output as if you apply λ^c directly. Figure 4.2b illustrates transition correctness: from q^c, if you abstract and operate, you get the same abstract state as if you first operate and then abstract. A function A that satisfies the three properties listed above is called a *homomorphism.*

Proof

The proof is a straightforward application of Theorem 2 defined in Section 4.2.2. □

4.1.3 Example

We will use the theorem to prove that R^c is a correct implementation of R^a, where R^c is defined as follows:

(a) output correctness (b) transition correctness

Figure 4.2: Commuting diagrams

$$Q = \{s_0, s_1, s_2, s_3\}$$
$$\Sigma = \{1\}$$
$$\Delta = \{y, n\}$$
$$\lambda(r_i, 1) = (i \text{ is even} \rightarrow n \mid true \rightarrow y)$$
$$\delta(r_i, 1) = r_{(i+1) \bmod 4}$$
$$q_0 = s_0$$

The abstraction function A is defined as follows:

$$\mathcal{A}(s_i) = (i \text{ is even} \rightarrow r_0 \mid true \rightarrow r_1)$$

The transition diagram for R^c is shown at the bottom of Figure 4.3. The dashed lines associate each R^c state with an R^a state and thus represent the abstraction function.

In this example, we prove output and transition correctness by deriving the right-hand side of the appropriate equality from the left-hand side. The justification of each step of the derivation is obvious and will be omitted. In the proofs that follow, assume that δ^a and λ^a are the transition and output functions of R^a and that δ^c and λ^c are the transition and output functions of R^c.

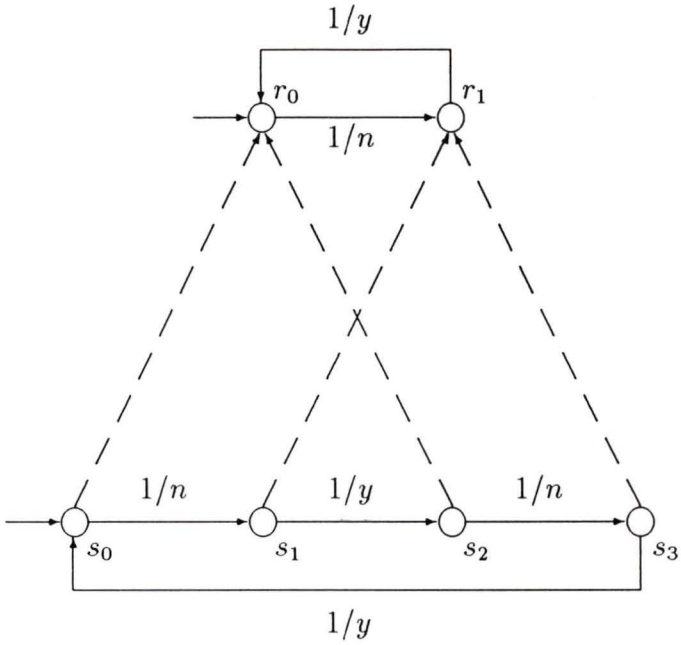


Figure 4.3: Abstract and concrete mod 2 counters

1. *initial correctness*

   Follows directly from the definition of $\mathcal{A}$.

2. *output correctness*

   Let s_i be an R^c state for some $i \in [0,3]$. There are two cases, depending on whether i is even or odd.

   (a) case 1: $i \in \{0,2\}$

   $$\lambda^a(\mathcal{A}(s_i),1) = \lambda^a(r_0,1) = n = \lambda^c(s_i,1)$$

   (b) case 2: $i \in \{1,3\}$

   $$\lambda^a(\mathcal{A}(s_i),1) = \lambda^a(r_1,1) = y = \lambda^c(s_i,1)$$

3. *transition correctness*

   Let s_i be an R^c state for some $i \in [0,3]$. There are two cases, depending on whether i is even or odd.

   (a) case 1: $i \in \{0,2\}$

   $$\delta^a(\mathcal{A}(s_i),1) = \delta^a(r_0,1) = r_1 = \mathcal{A}(s_{i+1}) = \mathcal{A}(\delta^c(s_i,1))$$

   (b) case 2: $i \in \{1,3\}$

   $$\delta^a(\mathcal{A}(s_i),1) = \delta^a(r_1,1) = r_0 = \mathcal{A}(s_{(i+1)mod4}) = \mathcal{A}(\delta^c(s_i,1))$$

4.2 Nondeterministic machines

4.2.1 Definitions

An infinite, nondeterministic Mealy machine is a 5-tuple $\langle Q, \Sigma, \Delta, f, Q_0 \rangle$ where Q is a set of states, Σ is the input alphabet, Δ is the output alphabet, $f : Q \times \Sigma \to 2^{Q \times \Delta} - \{\}$ is a total function, and $Q_0 \subseteq Q$.

Some explanation of the function f is required. A deterministic Mealy machine maps each state/input pair to exactly one state and one output; this behaviour is modelled by the functions δ and λ. A nondeterministic Mealy machine maps each state/input pair to *one or more* possible state/output pairs. We model this behaviour as a function f which maps each state/input pair to a *non-empty* subset of $Q \times \Delta$ (the set of all possible state/output pairs). A relation, such as a subset of $Q \times \Sigma \times Q \times \Delta$, could be used to model the behaviour of a nondeterministic Mealy machine.

If $M = \langle Q, \Sigma, \Delta, f, Q_0 \rangle$ and $w = \langle \langle x_1, y_1 \rangle, \ldots, \langle x_n, y_n \rangle \rangle$, then

$$\langle q_0, \langle x_1, y_1 \rangle, q_1, \ldots, \langle x_n, y_n \rangle, q_n \rangle$$

is a *computation of w on M* if

$$q_0 \in Q_0 \wedge (\forall i \in [1,n])(x_i \in \Sigma \wedge \langle q_i, y_i \rangle \in f(q_{i-1}, x_i))$$

The language of M, $L(M)$, is

$$\{w \mid \text{there is a computation of } w \text{ on } M\}$$

Our definition of the language of a Mealy machine suggests the following view of strings in the language. If an event is something which causes a transition from one state to another, then it seems appropriate to think of input/output pairs as events. It might seem more appropriate to view inputs, not input/output pairs, as events. The problem with that view is that, for a particular starting state of a nondeterministic Mealy machine, each input does not uniquely determine the "next" state; however, the next state *is* uniquely determined by an input/output pair. As we normally think of history as a sequence of events, we shall call the elements of the language of a Mealy machine *histories*.

Given machines M^a (abstract) and M^c (concrete), M^c is correct with respect to M^a if $L(M^c) \subseteq L(M^a)$. In some sense, if M^a defines the expected behaviour, then M^c must not generate any unexpected outputs. Note that the two definitions of correctness that we have presented are equivalent if both machines are deterministic.

A theorem that can be used to prove the correctness of nondeterministic Mealy machines is presented below. The two lemmas used in the proof of the theorem are presented later, because their conditions include those of the theorem and we do not wish to state these conditions more than once.

4.2.2 Theorem

Theorem 2 Let $M^c = \langle Q^c, \Sigma, \Delta, f^c, Q_0^c \rangle$ and $M^a = \langle Q^a, \Sigma, \Delta, f^a, Q_0^a \rangle$ be module state machines. If there exists a total function $\mathcal{A} : Q^c \rightarrow Q^a$ such that

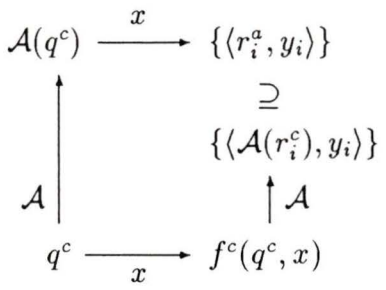


Figure 4.4: Transition-output correctness

1. *initial correctness*
   $\mathcal{A}(Q_0^c) \subseteq Q_0^a$
2. *transition-output correctness*
   $\langle r^c, y\rangle \in f^c(q^c, x) \rightarrow \langle \mathcal{A}(r^c), y\rangle \in f^a(\mathcal{A}(q^c), x)$

then $L(M^c) \subseteq L(M^a)$.

Initial correctness requires that all of M^c's initial states are also initial states of M^a. Figure 4.4 illustrates transition-output correctness: if $\langle r^c, y\rangle$ is a possible result of the concrete operation $f^c(q^c, x)$, then $\langle \mathcal{A}(r^c), y\rangle$ is a possible result of $f^a(\mathcal{A}(q^c), x)$.

Proof

Rephrase the conclusion as "$w \in L(M^c)$ implies $w \in L(M^a)$".

Assume $w = \langle\langle x_1, y_1\rangle, \ldots, \langle x_n, y_n\rangle\rangle \in L(M^c)$. Then there is a computation

$$\langle q_0^c, \langle x_1, y_1, \rangle, q_1^c, \ldots, \langle x_n, y_n, \rangle, q_n^c\rangle$$

of w on M^c. By Lemma 2, $\langle \mathcal{A}(q_0^c), \langle x_1, y_1, \rangle, \mathcal{A}(q_1^c), \ldots, \langle x_k, y_n, \rangle, \mathcal{A}(q_n^c)\rangle$ is a computation of w on M^a. Therefore, $w \in L(M^a)$. □

4.2.3 Lemmas

Lemma 1 If M is an MSM,

$w = \langle\langle x_1, y_1\rangle, \ldots, \langle x_n, y_n\rangle\rangle \in L(M)$, and

$\langle q_0, \langle x_1, y_1\rangle, q_1, \ldots, \langle x_n, y_n\rangle, q_n\rangle$ is a computation of w on M,

then, for all $k \in [0, n]$,

$w' = \langle\langle x_1, y_1\rangle, \ldots, \langle x_k, y_k\rangle\rangle \in L(M)$, and

$\langle q_0, \langle x_1, y_1\rangle, q_1, \ldots, \langle x_k, y_k\rangle, q_k\rangle$ is a computation of w' on MSM M.

Proof

Immediate. □

Lemma 2 If the conditions of the theorem hold and

$C^c = \langle q_0^c, \langle x_1, y_1\rangle, q_1^c, \ldots, \langle x_n, y_n\rangle, q_n^c\rangle$

is a computation of $w = \langle\langle x_1, y_1\rangle, \ldots, \langle x_n, y_n\rangle\rangle$ on M^c, then

$\langle \mathcal{A}(q_0^a), \langle x_1, y_1\rangle, \mathcal{A}(q_1^a), \ldots, \langle x_n, y_n\rangle, \mathcal{A}(q_n^a)\rangle$

is a computation of w on M^a.

Proof

The proof is by induction on n.

Base Case: $n = 0$.

$w = \langle\rangle$ and $C^c = \langle q_0^c\rangle$. By (1), $\mathcal{A}(q_0^c) \in Q_0^a$. Thus, $\langle \mathcal{A}(q_0^c)\rangle$ is a computation of w on M^a.

Induction Step: Assume the lemma is true when $n = k$, for some $k \geq 0$; show true for $n = k + 1$. Let w be the string $\langle\langle x_1, y_1\rangle, \ldots, \langle x_{k+1}, y_{k+1}\rangle\rangle$ and let $C^c = \langle q_0^c, \langle x_1, y_1\rangle, q_1^c, \ldots, \langle x_{k+1}, y_{k+1}\rangle, q_{k+1}^c\rangle$ be a computation of w on M^c.

By Lemma 1,

$$\langle q_0^c, \langle x_1, y_1\rangle, q_1^c, \ldots, \langle x_k, y_k\rangle, q_k^c\rangle$$

is a computation of $w' = \langle\langle x_1, y_1\rangle, \ldots, \langle x_k, y_k\rangle\rangle$ on M^c. By the induction hypothesis,

$$\langle \mathcal{A}(q_0^c), \langle x_1, y_1\rangle, \mathcal{A}(q_1^c), \ldots, \langle x_k, y_k\rangle, \mathcal{A}(q_k^c)\rangle$$

is a computation of w' on M^a.

By the definition of a computation, $\langle q_{k+1}^c, y_{k+1}\rangle \in f^c(q_k^c, x_{k+1})$. By (2), $\langle \mathcal{A}(q_{k+1}^c), y_{k+1}\rangle \in f^a(\mathcal{A}(q_k^c), x_{k+1})$. Therefore,

$$\langle \mathcal{A}(q_0^c), \langle x_1, y_1\rangle, \mathcal{A}(q_1^c), \ldots, \langle x_k, y_k\rangle, \mathcal{A}(q_k^c), \langle x_{k+1}, y_{k+1}\rangle, \mathcal{A}(q_{k+1}^c)\rangle$$

is a computation of w on M^a. □

4.2.4 Corollary

Corollary 1 Let $M^c = \langle Q, \Sigma, \Delta, f^c, Q_0\rangle$ and $M^a = \langle Q, \Sigma, \Delta, f^a, Q_0\rangle$ be Mealy machines. If $f^c(q, c) \subseteq f^a(q, c)$ for all states $q \in Q$ and inputs c, then $L(M^c) \subseteq L(M^a)$.

Proof

Let $\mathcal{A}$ be the identity function on Q. It is clear that $\mathcal{A}$ satisfies the initial correctness and transition-output correctness conditions of Theorem 2. □

4.3 Summary

In this chapter, we presented the theory of nondeterministic Mealy machines on which our verification techniques are based. Our notions of computation and language are used to define precisely what we mean by module correctness. We presented a theorem which shows how abstraction functions can be used to simplify correctness proofs. This theorem is applied in Chapter 6 to define procedures for verifying that a program specification is correct with respect to an interface specification. We presented a corollary which applies

when two Mealy machines have the same state space; this corollary serves as the basis for a procedure for proving that implementations are correct with respect to program specifications. In the next chapter, we present notations for specifying *module state machines*: Mealy machines corresponding to each of our three module work products.

Chapter 5

Module state machines

The Mealy machines of the previous section provide the theoretical basis for our approach. For proofs about properties of Mealy machines, the notations of that section are effective. However, for use in software documentation other notations are preferable.

In this section, we present *module state machines* (MSM), a subset of the class of Mealy machines, and document formats for specifying three types of MSM. These formats provide the structure needed when specifying complex MSMs and save time when reviewing specifications. Further, the formats are restrictive, eliminating many undesirable Mealy machines and shortening specifications by making much implicit.

5.1 Interface specification

In this section, we describe *interface specifications*, summarizing the standard document sections and illustrating them on an *sset* (see Section 1.3) specification. We then provide the semantics by giving a mapping to the underlying Mealy machine.

5.1.1 Standard specification sections

The **state** section defines the specification state space, by declaring a collection of typed variables. The types *integer*, *real*, *character*, and *boolean* are assumed; new types are declared using the type constructors *set*, *sequence*, and *tuple*.

The **access program semantics** section contains one subsection for each access program. Each of these subsections contains an **exceptions** entry, specifying the situations in which each exception must be signaled. The exceptions are specified by an assignment to the variable *exc*, or by "none," indicating that no exception is ever signaled.

There is a **transition** entry for each set program, specifying the new state resulting from its invocation. The transition applies only if the call is exception-free — otherwise the state is unchanged.

An **output** entry is required for each get program, specifying its return value. The return value is specified by an assignment to the variable *out*, and applies only if the call is exception-free. If the call generates an exception, then the output is *dontcare* — any value of the correct type may be returned.

A **transition-output** entry is required for each set-get program, specifying the state transition and return value corresponding to an exception-free call. As for set and get calls, when an exception is generated, there is no state change and the output is *dontcare*. Note that set-get programs cannot, in general, be specified with independent **transition** and **output** sections. Consider a nondeterministic set-get call where the new-state/output pairs $\langle s_1, o_1 \rangle$ and $\langle s_2, o_2 \rangle$ are permitted, while $\langle s_1, o_2 \rangle$ is not. The **transition-output** entry allows the specifier to allow the first two while prohibiting the third. Chapter 7 provides a simple example of such a set-get program.

We illustrate the above ideas on the *sset* interface specification shown in

state
    s: set of integer
access program semantics
    s_init:
        transition: $s := \{\}$
        exception: none
    $s\_add(x)$:
        transition: $s := s \cup \{x\}$
        exception: $exc := (\mid s \mid = N \rightarrow full \mid x \in s \rightarrow mem)$
    $s\_del(x)$:
        transition: $s := s - \{x\}$
        exception: $exc := (x \notin s \rightarrow notmem)$
    $g\_mem(x)$:
        output: $out := x \in s$
        exception: none

Figure 5.1: *sset* interface specification

Figure 5.1. The specification state is s, a set of integers. To illustrate the access program semantics of *sset*, consider the trace $s\_init.s\_add(3).g\_mem(3)$. The state, output, and exception generated by each call of the trace is shown in Figure 5.2.

5.1.2 Specification semantics

In practice, we use a variety of notations in the interface specification entries. Below we use functions and relations to precisely characterize what is required in all specifications. We then give a mapping from interface specifications to their corresponding Mealy machines.

Let S be the specification state space, O the union of the output types for all the get and set-get programs, O_p the output type for get or set-get program p, and E the set of all exceptions.

Call	State	Output	Exception
s_init	{}	none	none
s_add(3)	{3}	none	none
g_mem(3)	{3}	*true*	none

Figure 5.2: Execution of $s\_init.s\_add(3).g\_mem(3)$

Consider access program p, with parameters of type $T_1, T_2, \ldots, T_n$. The **exception** section for p must specify the function

$$e_p : S \times T_1 \times T_2 \times \ldots \times T_n \rightarrow E$$

e_p is normally partial. If p is a set program, the **transition** section for p must specify the function

$$t_p : S \times T_1 \times T_2 \times \ldots \times T_n \rightarrow 2^S - \{\}$$

While t_p may be partial, it must be defined where e_p is not. That is

$$[(S \times T_1 \times T_2 \times \ldots \times T_n) - dom(e_p)] \subseteq dom(t_p)$$

We intepret t_p informally as follows. Let X denote a sequence $x_1, x_2, \ldots x_n$ of values such that $x_i \in T_i$ for all $i \in [1, n]$. If a set-call call $p(X)$ is exception-free in state q, then $t_p(q, X)$ is the set of states that could possibly be generated by the call. Since the new state resulting from an exception-free set-call must be defined, the set $t_p(q, X)$ cannot be empty. Therefore, the range of t_p is $2^S - \{\}$.

Suppose p is a get program with return value type O_p. The **output** section for p must specify the function

$$o_p : S \times T_1 \times T_2 \times \ldots \times T_n \rightarrow 2^{O_p} - \{\}$$

Like t_p, o_p may be partial but must be defined where e_p is not. If a call $p(X)$ is exception-free in state q, then $o_p(q, X)$ is the set of possible outputs that could be returned by the call. Since each get-call must always return a value, the range of o_p excludes the empty set.

If p is a set-get program with return value type O_p, then the **transition-output** section for p must specify the function

$$to_p : S \times T_1 \times T_2 \times \ldots \times T_n \rightarrow 2^{S \times O_p} - \{\}$$

Like t_p, to_p may be partial but must be defined where e_p is not.

Note that a number of the restrictions implied in the functions and relations just described are not always appropriate. For example, by choosing E rather than 2^E for the range of e_p, we have restricted exception signaling to be deterministic. Many adjustments are possible: to reduce or increase nondeterminism, to allow transitions when exceptions occur, etc.

In the next section, we define a mapping from interface specifications to Mealy machines; the mapping from program specifications to Mealy machines is virtually identical. We use these mappings in Section 5.3 to define precisely what we mean by the correctness of a program specification with respect to an interface specification.

5.1.3 Mapping to Mealy machines

Recall that a nondeterministic Mealy machine is a 5-tuple $\langle Q, \Sigma, \Delta, f, Q_0 \rangle$. The Mealy machine corresponding to an interface specification is as follows.

$Q = S$

$\Sigma =$ the set of all access program calls

$\Delta = (O \cup \{\perp\}) \times (E \cup \{\perp\})$, assuming that $\perp \notin O \cup E$

$Q_0 =$ the set of states possibly established by *s_init*

	p is a set call	p is a get call	p is a set-get call
$\langle q, X\rangle \in dom(e_p)$	$r = q$ $v = \perp$ $e = e_p(q, X)$	$r = q$ $v \in O_p$ $e = e_p(q, X)$	$r = q$ $v \in O_p$ $e = e_p(q, X)$
$\langle q, X\rangle \notin dom(e_p)$	$r \in t_p(q, X)$ $v = \perp$ $e = \perp$	$r = q$ $v \in o_p(q, X)$ $e = \perp$	$\langle r, v\rangle \in to_p(q, X)$ $e = \perp$

Figure 5.3: f defined by an interface specification

Where an access program signals no exception or returns no value, we use $\perp$ in the corresponding Mealy machine. For state s and access program call $p(x_1, \ldots, x_n)$, $f(s, p(x_1, \ldots, x_n))$ is a set whose elements are of the form $\langle s', \langle v, e\rangle\rangle$, where s' is a state, v is an output, and e is an exception. The constraints on s', v, and e are defined in Figure 5.3.

To illustrate the mapping, let $M^I = \langle Q^I, \Sigma, \Delta, f^I, Q_0^I\rangle$ be the nondeterministic Mealy machine represented by the *sset* interface specification. Let Z denote the set of integers. Then,

$$Q^I = 2^Z$$
$$\Sigma = \{c(x) \mid c \in \{s\_add, s\_del, g\_mem\} \wedge x \in Z\}$$
$$\Delta = (Z \cup \{true, false, \perp\}) \times \{full, mem, notmem, \perp\}$$
$$Q_0^I = \{\{\}\}$$

While f^I is as defined by Figure 5.3, it has no simple defining expression. However, we can illustrate f^I by showing that the history

$$w = \langle\langle s\_add(3), \langle\perp, \perp\rangle\rangle, \langle g\_mem(3), \langle true, \perp\rangle\rangle\rangle$$

is an element of the language of M^I. The task is reduced to finding a computation of w on M^I.

First, we use Figure 5.3 to hand-execute the call $s\_add(3)$ in the initial state $\{\}$. Since the initial state is empty and $N > 0$, $s\_add(3)$ is exception-free in the initial state. Therefore, according to Figure 5.3,

$$
\begin{aligned}
&f^I(\{\}, s\_add(3)) \\
&\quad = \{\langle r, \langle \perp, \perp \rangle\rangle \mid r \in t_{s\_add}(\{\}, 3)\} \\
&\quad = \{\langle r, \langle \perp, \perp \rangle\rangle \mid r \in \{\{\} \cup \{3\}\}\} \\
&\quad = \{\langle r, \langle \perp, \perp \rangle\rangle \mid r \in \{\{3\}\}\} \\
&\quad = \{\langle \{3\}, \langle \perp, \perp \rangle\rangle\}
\end{aligned}
$$

Thus, f^I maps the initial state and input $s\_add(3)$ to exactly one state/output pair.

Next, we hand-execute $g\_mem(3)$ in the state $\{3\}$ generated by the previous call. Since g_mem signals no exceptions, $dom(e_{g\_mem}) = \{\}$ and $g\_mem(3)$ is exception-free in state $\{3\}$. Therefore, according to Figure 5.3,

$$
\begin{aligned}
&f^I(\{3\}, g\_mem(3)) \\
&\quad = \{\langle \{3\}, \langle v, \perp \rangle\rangle \mid v \in o_{g\_mem}(\{3\}, 3)\} \\
&\quad = \{\langle \{3\}, \langle v, \perp \rangle\rangle \mid v \in \{3 \in \{3\}\}\} \\
&\quad = \{\langle \{3\}, \langle v, \perp \rangle\rangle \mid v \in \{true\}\} \\
&\quad = \{\langle \{3\}, \langle true, \perp \rangle\rangle\}
\end{aligned}
$$

As $\langle \{\}, \langle s\_add(3), \langle \perp, \perp \rangle\rangle, \{3\}, \langle g\_mem(3), \langle true, \perp \rangle\rangle, \{3\}\rangle$ is a computation of w on M^I, we conclude that $w \in L(M^I)$.

5.2 Program specification

Program specifications are nearly identical in form and semantics to interface specifications. The key difference is in the **state** section. The program

specification state is declared using the constructs of the implementation programming language, C for our purposes.

Thus the program specification records one important design decision not present in the interface specification: the run-time representation of the module state. Here the state is chosen on the basis of implementation efficiency or simplicity. In the interface specification, the state is chosen to make the service description as simple and precise as possible.

In the program specification, one additional item is required. The **state invariant** entry contains a predicate on the state variables, designed to eliminate "meaningless" states. The program specification state space is the set of values permitted by the declarations that also satisfy the state invariant. It is invariant in the sense that it must be established by *s_init*, and maintained by the other access programs: if it holds on entry, then it must hold on exit. The state invariant restricts the state space so that the **access program semantics** entries are well-defined; it also supports many of the verification steps discussed in the next section.

The *sset* program specification is shown in Figure 5.4, where C state variables are shown in the `typewriter` font. The set size is maintained in `scnt` and the set elements are stored in `s`[0..`scnt` − 1]. The state invariant requires that `scnt` be a legitimate set size and that `s`[0..`scnt` − 1] contains no duplicates. *s_init* initializes the set to empty, *s_add*(x) adds x at the end, *s_del*(x) deletes x by swapping (unless the rightmost element is being deleted), and *g_mem*(x) returns the value of $x \in$ `s`[0..`scnt` − 1]. For *s_add*, *s_del*, and *g_mem* the value of the expression $x \in$ `s`[0..`scnt` − 1] must be computed — the method used is part of the implementation.

To illustrate the access program semantics of the *sset* program specifica-

state

`int s[N];`
`int scnt;`
state invariant
$\texttt{scnt} \in [0, N] \wedge (\forall i, j \in [0, \texttt{scnt} - 1])(i \neq j \rightarrow \texttt{s}[i] \neq \texttt{s}[j])$

access program semantics

s_init() :
transition: $\texttt{scnt} := 0$
exception: none

s_add(x) :
transition: $\texttt{s[scnt]}, \texttt{scnt} := x, \texttt{scnt} + 1$
exception: $exc := (\texttt{scnt} = N \rightarrow full \mid x \in \texttt{s}[0..\texttt{scnt} - 1] \rightarrow mem)$

s_del(x) :
transition:
$(pos(x) < scnt - 1 \rightarrow \texttt{s}[pos(x)], \texttt{scnt} := \texttt{s}[\texttt{scnt} - 1], \texttt{scnt} - 1$
$\mid pos(x) = scnt - 1 \rightarrow \texttt{scnt} := \texttt{scnt} - 1)$
where $pos(x)$ is the index of x in $\texttt{s}[0..\texttt{scnt} - 1]$
exception: $exc := (x \notin \texttt{s}[0..\texttt{scnt} - 1] \rightarrow notmem)$

g_mem(x) :
output: $out := x \in \texttt{s}[0..\texttt{scnt} - 1]$
exception: none

Figure 5.4: *sset* program specification

tion, consider the trace

$$s\_init.s\_add(3).g\_mem(3)$$

The state, output, and exception generated by each call in the trace is shown in Figure 5.5. Program specification states are represented as tuples of the form $\langle s, scnt \rangle$. We assume that $N = 3$ so that instances of s may be expressed compactly. Elements of s that are irrelevant (more precisely, do not determine the behaviour of access program calls) are identified by "?". For example, the initial state is $\langle \langle ?, ?, ? \rangle, 0 \rangle$.

In Section 5.1.3, we illustrated the mapping from MSMs to Mealy ma-

Call	State	Output	Exception
s_init	$\langle\langle ?, ?, ? \rangle, 0\rangle$	none	none
s_add(3)	$\langle\langle 3, ?, ? \rangle, 1\rangle$	none	none
g_mem(3)	$\langle\langle 3, ?, ? \rangle, 1\rangle$	*true*	none

Figure 5.5: Execution of *s_init*.*s_add*(3).*g_mem*(3)

chines by defining, in part, the Mealy machine represented by the *sset* interface specification. To further illustrate the mapping, we shall partially construct the Mealy machine represented by the *sset* program specification. Let $M^P = \langle Q^P, \Sigma, \Delta, f^P, Q_0^P \rangle$ be the Mealy machine represented by the *sset* program specification. Let Z denote the set of integers. Since the invariant determines the program specification state-space, $Q^P = Z^N \times [1, N]$. According to the **transition** section of *s_init*, $Q_0^P = Z^N \times \{0\}$. Σ and Δ are the same for both M^I and M^P.

We illustrate f^P by showing that

$$w = \langle\langle s\_add(3), \langle \perp, \perp \rangle\rangle, \langle g\_mem(3), \langle true, \perp \rangle\rangle\rangle \in L(M^P)$$

Thus, we must find a computation of w on M^P.

First, we use Figure 5.3 to hand-execute the call *s_add*(3) in the initial state $\langle\langle ?, ?, ? \rangle, 0\rangle$. Since $N > 0$ and $scnt = 0$, *s_add*(3) is exception-free in the initial state. Therefore, according to Figure 5.3,

$$\begin{aligned} & f^P(\langle\langle ?, ?, ? \rangle, 0\rangle, s\_add(3)) \\ & \quad = \{\langle r, \langle \perp, \perp \rangle\rangle \mid r \in t_{s\_add}(\langle\langle ?, ?, ? \rangle, 0\rangle, 3)\} \\ & \quad = \{\langle r, \langle \perp, \perp \rangle\rangle \mid r \in \{\langle\langle 3, ?, ? \rangle, 1\rangle\}\} \\ & \quad = \{\langle\langle\langle 3, ?, ? \rangle, 1\rangle, \langle \perp, \perp \rangle\rangle\} \end{aligned}$$

Since $dom(e_{g\_mem}) = \{\}$, *g_mem*(3) is exception-free in state $\langle\langle 3, ?, ? \rangle, 1\rangle$. Therefore,

$$
\begin{aligned}
&f(\langle\langle 3, ?, ?\rangle, 1\rangle, g\_mem(3)) \\
&\quad = \{\langle\langle\langle 3, ?, ?\rangle, 1\rangle, \langle v, \perp\rangle\rangle \mid v \in o_{g\_mem}(\langle\langle 3, ?, ?\rangle, 1\rangle, 3)\} \\
&\quad = \{\langle\langle\langle 3, ?, ?\rangle, 1\rangle, \langle 3 \in s[0..scnt - 1], \perp\rangle\rangle\} \\
&\quad = \{\langle\langle\langle 3, ?, ?\rangle, 1\rangle, \langle 3 \in \langle 3\rangle, \perp\rangle\rangle\} \\
&\quad = \{\langle\langle\langle 3, ?, ?\rangle, 1\rangle, \langle true, \perp\rangle\rangle\}
\end{aligned}
$$

Having constructed a computation of w on M^P, we have demonstrated that w is in the language of $L(M^P)$.

In the next section, we use the mapping defined in Section 5.1.3 to define precisely what we mean by the correctness of a program specification with respect to an interface specification.

5.3 Correctness

A program specification P is correct with respect to an interface specification I, if $L(M^c) \subseteq L(M^a)$, where M^c and M^a are the Mealy machines represented (according to the mapping defined in the previous section) by P and I, respectively.

5.4 Implementation

Our implementations are written in C and correspond closely to the program specifications. The two operate on the same state, with the implementation computing the exception checks, state transitions, and outputs defined in the program specification. The *sset* implementation is shown in Figure 5.6.

Defining, in general, the transition, output, exception, and transition-output functions of access program implementations is beyond the scope of this thesis. It is expected, however, that these functions can be defined

```
static int findpos(int x)
{    int i;
     for (i = scnt-1; i >= 0; i--) if (s[i] == x) return(i);
     return(scnt); /*not found*/ }

void s_init() { scnt = 0; }

void s_add(int x)
{    if (scnt == N) full();
     else if (findpos(x) != scnt) mem();
     else s[scnt++] = x; }

void s_del(int x)
{    int i;
     i = findpos(x);
     if (i == scnt) notmem(); else if (i < --scnt) s[i] = s[scnt]; }

int g_mem(int x) { return(findpos(x) != scnt); }
```

Figure 5.6: *sset* implementation

precisely in terms of program functions [3], and some additional assumptions about the structure of access program implementations. For example, we might require that each access program implementation consists of exception-checking code (with no side effects) followed by code for computing the transition, output, or transition-output. We would also need to model the behaviour of return statements and calls to exception-handlers, perhaps in terms of assignments to the pseudo-variables *out* and *exc*. We propose to explore these issues in future work.

5.5 Summary

In this chapter, we presented notations for specifying interface and program specifications. These notations eliminate many undesirable Mealy machines (for example, Mealy machines whose transition-output function is not defined for all state/input pairs), and shorten specifications by leaving much implicit (such as the behaviour of an access program when an exception occurs). Both work products contain a section that describes the module state, and a section that describes the effects of the access programs on the state. Program specifications often contain an additional section that specifies a state invariant: an assertion that is established by *s_init* and maintained by every other access program. We defined a mapping from interface and program specifications to Mealy machines. In the next chapter, we present a procedure for proving the correctness of program specifications with respect to interface specifications.

Chapter 6

Verification procedures

The procedure for verifying that a module implementation is correct with respect to an interface specification consists of four steps.

1. Prove that the interface specification is well-formed.
2. Prove that the program specification is well-formed.
3. Prove that the program specification meets the interface specification.
4. Prove that the implementation meets the program specification.

Steps 1 and 2 involve proving that the interface specification and the program specification are well-defined according to the rules presented in Chapter 5. For the interface specification, it is necessary to prove that the behaviour (**transition**, **output**, and **transition-output**) of each access program is well-defined for exception-free calls in any state. For the program specification, it is necessary to prove that (a) the state invariant is established by *s_init* and maintained by every other call and (b) the behaviour of each access program is well-defined for exception-free calls in states satisfying the invariant.

Steps 3 and 4 require proving that one work product is correct with respect to another. Step 3 is described in detail in Section 6.1. Step 4 is discussed in Section 6.2.

Henceforth, I, P, and C denote a well-formed interface specification, program specification, and implementation, respectively. In addition, we shall assume that the three MSMs define exactly the same set of access programs, with the same signatures. As a consequence, the corresponding Mealy machines have the same input and output alphabets.

6.1 Program specification meets interface specification

The procedure for verifying that a program specification P meets an interface specification I consists of five steps.

1. Define the abstraction function $\mathcal{A}$ and prove that it is defined on every P state that satisfies P's state invariant.

2. *initial correctness*
   Prove that each initial P state abstracts to an initial I state.

3. For each set call c

   (a) *exception correctness*
   Prove that c signals an exception in P state q if and only if c signals the same exception in I state $\mathcal{A}(q)$.

   (b) *transition correctness*—assuming c is exception-free
   Abstract the P state; calculate the I transition; calculate the P transition; show that the final P states abstract to a subset of the final I states.

4. For each get call c

   (a) *exception correctness*
   Identical to 3a.

   (b) *output correctness*—assuming c is exception-free
   Abstract the P state; compute the I outputs; compute the P outputs and show that these are a subset of the I outputs.

5. For each set-get call c

   (a) *exception correctness*
   Identical to 3a.

   (b) *transition-output correctness*—assuming c is exception-free
   Abstract the P state. Compute the set S of state/output pairs defined by the I transition-output. Compute the set R of state/output pairs defined by the P transition-output. Prove that, after abstracting the state, each element of R is an element of S.

In the next section, we show that the verification procedure never leads to incorrect conclusions.

6.1.1 Soundness

Given a well-formed program specification P and a well-formed interface specification I, we say that the verification procedure is *successful* if an abstraction function mapping P states to I states can be found that passes all of the steps of the verification procedure. The verification procedure is *sound* if, whenever it is successful, it is also true that P is correct with respect to I.

The following notation is used in the soundness argument. Let M^P and M^I denote the nondeterministic Mealy machines corresponding to P and I, respectively, according to the mapping defined in Section 5.1.2. The superscripts "P" and "I" are used to distinguish between components of the MSMs P and I as well as between components of the associated Mealy machines, M^P and M^I. For example, e_p^P is the program specification exception function for access program p of P and Q_0^I is the set of initial states of M^I.

The soundness of the verification procedure is established by showing that, if the verification procedure is successful when applied to P and I, then M^P and M^I satisfy the conditions of Theorem 2. The following are the conditions of the theorem as it applies to M^I in the role of M^a and M^P in the role of M^c.

I. *well-formedness*

(a) $Q_0^P \subseteq Q^P$

(b) $Q_0^I \subseteq Q^I$

(c) f^P and f^I are total

II. $\mathcal{A}$ is total

III. *initial correctness*

$\mathcal{A}(Q_0^P) \subseteq Q_0^I$

IV. *transition-output correctness*

$\langle r, y \rangle \in f^P(q, c) \rightarrow \langle \mathcal{A}(r), y \rangle \in f^I(\mathcal{A}(q), c)$

The proof of soundness is straightforward but tedious. First, we assume that P and I are well-formed according to the rules presented in Chapter 5. The precise results established by the verification procedure are listed in Figure 6.1. To prove soundness, we must show that these assumptions, together with the definition of the mapping from MSMs to Mealy machines, imply conditions I - IV.

1. $\mathcal{A}$ *total*: $S^P \subseteq dom(\mathcal{A})$
2. *initial correctness*: $\mathcal{A}(S_0^P) \subseteq S_0^I$
3. for each set call c
   (a) *exception correctness*:
      (i) $\langle q, c\rangle \in dom(e^P) \leftrightarrow \langle \mathcal{A}(q), c\rangle \in dom(e^I)$
      (ii) $\langle q, c\rangle \in dom(e^P) \rightarrow e^P(q, c) = e^I(\mathcal{A}(q), c)$
   (b) *transition correctness*:
      $\langle q, c\rangle \notin dom(e^P) \rightarrow \mathcal{A}(t^P(q, c)) \subseteq t^I(\mathcal{A}(q), c)$
4. for each get call c
   (a) *exception correctness*: Identical to 3a.
   (b) *output correctness*:
      $\langle q, c\rangle \notin dom(e^P) \rightarrow o^P(q, c) \subseteq o^I(\mathcal{A}(q), c)$
5. for each set-get call c
   (a) *exception correctness*: Identical to 3a.
   (b) *transition-output correctness*:
      $\langle q, c\rangle \notin dom(e^P) \wedge \langle r, v\rangle \in to^P(q, c) \rightarrow \langle \mathcal{A}(r), v\rangle \in to^I(\mathcal{A}(q), c)$

Figure 6.1: Assertions verified

Proof of soundness

Since P is well-formed, each possible state established by *s_init* satisfies the invariant. Since the states of M^P are exactly those states of P that satisfy the invariant, I.a holds. I.b and I.c are true because P and I are well-formed. II and III follow directly from steps 1 and 2, respectively, of Figure 6.1.

The proof of IV is divided into six cases, one for each cell of Figure 5.3. We shall consider only the two set-call cases; proofs of the remaining four cases are similar.

1. Exception: $\langle q, c\rangle \in dom(e^P)$
   Assume $\langle r, y\rangle \in f^P(q, c)$; we must show that $\langle \mathcal{A}(r), y\rangle \in f^I(\mathcal{A}(q), c)$. By the definition of M^P's output alphabet, $y = \langle v, e\rangle$ for some $v \in O \cup \{\perp\}$ and $e \in E \cup \{\perp\}$, where O is the union of all possible return types of access programs and E is the set of all possible exceptions.

By the definition of f^P, $r = q$, $v = \perp$, and $e = e^P(q, c)$. By step 3.a of Figure 6.1, $\langle \mathcal{A}(q), c \rangle \in dom(e^I)$ and $e = e^I(\mathcal{A}(q), c)$. Thus, by the definition of f^I,

$$\langle \mathcal{A}(r), y \rangle = \langle \mathcal{A}(r), \langle v, e \rangle \rangle = \langle \mathcal{A}(q), \langle \perp, e^I(\mathcal{A}(q), c) \rangle \rangle \in f^I(\mathcal{A}(q), c)$$

2. Exception-free: $\langle q, c \rangle \notin dom(e^P)$

   Assume $\langle r, y \rangle \in f^P(q, c)$; we must show that $\langle \mathcal{A}(r), y \rangle \in f^I(\mathcal{A}(q), c)$. By the definition of M^P's output alphabet, $y = \langle v, e \rangle$ for some $v \in O \cup \{\perp\}$ and $e \in E \cup \{\perp\}$.

   By the definition of f^P, $r = t^P(q, c)$ and $v = e = \perp$. By step 3.a of Figure 6.1, $\langle \mathcal{A}(q), c \rangle \notin dom(e^I)$. By step 3.b,

$$\mathcal{A}(t^P(q, c)) = t^I(\mathcal{A}(q), c)$$

   Therefore, by the definition of f^I,

$$\begin{aligned} \langle \mathcal{A}(r), y \rangle &= \langle \mathcal{A}(r), \langle v, e \rangle \rangle \\ &= \langle \mathcal{A}(t^P(q, c)), \langle \perp, \perp \rangle \rangle \\ &= \langle t^I(\mathcal{A}(q), c), \langle \perp, \perp \rangle \rangle \\ &\in f^I(\mathcal{A}(q), c) \end{aligned}$$

6.2 Implementation meets program specification

To prove that an implementation meets a program specification, we may follow the steps defined in the last section; however, as the state spaces of the two MSMs are identical, the abstraction function is always identity. Therefore, step 1 is no longer required, and the remaining steps can be simplified. For example, to prove exception correctness for an access program p, one proves that the two exception functions are identical, i.e. $e_p^C = e_p^P$; to prove

transition correctness for a set program p, one proves that $t_p^C(q,c) \in t_p^P(q,c)$ for all states q satisfying the invariant and for all calls c to p.

As the transition, output, transition-output, and exception functions of the implementation are defined directly in terms of program functions, we may use standard program verification techniques in each of the verification steps. For example, suppose the code segment S computes the exception checks for an access program p. As proposed in Section 5.4, we define e_p^C to be the function computed by S. Therefore, to prove exception correctness for p, we must show that e_p^P is the function computed by S. The Correctness Theorem [3] can be applied directly to prove this.

The soundness argument is similar to the soundness argument of the last section, using Corollary 1 instead of Theorem 2. The proof is straightforward and will not be included.

6.3 Summary

In this chapter, we presented a procedure for verifying that an implementation is correct with respect to an interface specification, focussing on the task of proving that the program specification meets the interface specification. Our procedure greatly simplifies the proof task by breaking it up into simple steps which can be carried out independently. We proved that the procedure is sound with respect to (a) the mapping from work products to Mealy machines, and (b) the definition of correctness for Mealy machines. Finally, we sketched a procedure for verifying the correctness of implementations with respect to program specifications. In the next chapter, we illustrate the procedure by applying it to two modules: the simple-set (*sset*) module and the iterator-set module, which is an extension of *sset* that allows elements of the set to be retrieved sequentially.

Chapter 7

Verification examples

In this chapter, we apply the procedure for verifying the correctness of a program specification with respect to an interface specification to two modules: *sset* and *iset*. In Section 7.1, we prove that the *sset* program specification meets the interface specification; both of these work products were presented in Chapter 5. The *iset* module is an extension of the *sset* module that allows elements of the set to be retrieved sequentially. In Section 7.2, we present the *iset* interface specification and program specification, and a proof that the program specification meets the interface specification. This chapter does not contain any proofs involving implementations.

7.1 Simple set proof

We have chosen to represent *sset* interface specification states using set notation. The *sset* program specification state is represented using the array picture notation described in [6]. Arrays are represented as rectangles arranged vertically with elements and indices labelled when convenient. Primed labels, such as `s'` and `scnt'`, denote the values of state variables after a transition.

7.1.1 Abstraction function and lemmas

Consider the *sset* interface specification (Figure 5.1) and program specification (Figure 5.4). We define the abstraction function as follows:

$$\mathcal{A}(\langle \mathtt{s}, \mathtt{scnt} \rangle) = \{\mathtt{s}[i] \mid i \in [0, \mathtt{scnt} - 1]\}$$

A hypothesis of each of the following lemmas is that the P state $\langle \mathtt{s}, \mathtt{scnt} \rangle$ satisfies the invariant.

content lemma

$$x \in \mathcal{A}(\langle \mathtt{s}, \mathtt{scnt} \rangle) \leftrightarrow x \in \mathtt{s}[0..\mathtt{scnt} - 1]$$

size lemma

$$\mathtt{scnt} = \mid \mathcal{A}(\langle \mathtt{s}, \mathtt{scnt} \rangle) \mid$$

The proof of the content lemma follows directly from the definition of the abstraction function. The proof of the size lemma uses the fact that all of the first `scnt` elements of `s` are unique—as required by the invariant.

7.1.2 Proof

1. *initial correctness*

   Because $\mathtt{scnt} = 0$, $\mathcal{A}(\langle \mathtt{s}, \mathtt{scnt} \rangle) = \{\}$, as required.

2. *s_add*

   (a) *transition correctness*

   The commuting diagram for *s_add* is shown in Figure 7.1. Assume that the leftmost abstraction holds. The effect of the call on both the I state and P state is shown. Since `s[scnt]` is assigned x and no other elements of `s` are changed, the rightmost abstraction holds.

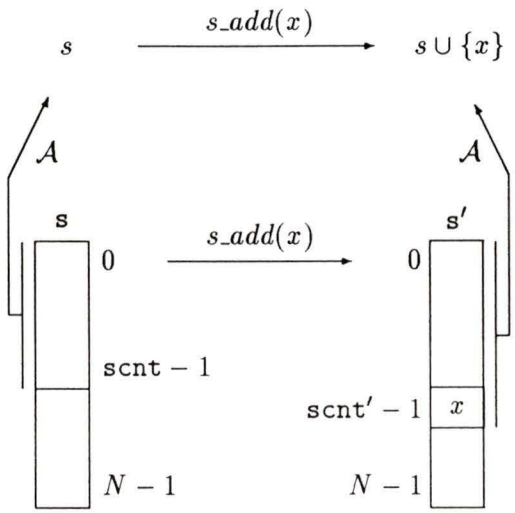


Figure 7.1: Transition correctness: *s_add*

(b) *exception correctness*

i. *full*

Follows directly from the size lemma.

ii. *mem*

Follows directly from the content lemma.

3. *s_del*

(a) *transition correctness*

In the program specification, the transition for *s_del* is specified as a two-part conditional rule. Hence, the proof is divided into two cases. The corresponding commuting diagrams are shown in Figure 7.2.

i. $pos(x) < \texttt{scnt} - 1$

Let S be the set consisting of the first scnt elements of s,

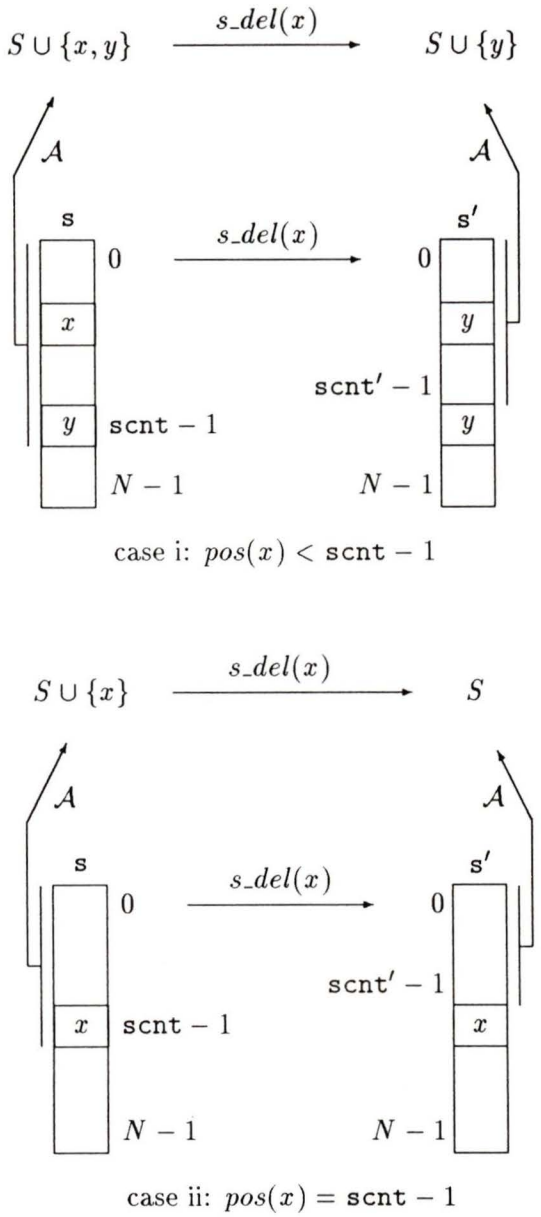


Figure 7.2: Transition correctness: *s_del*

excluding x, the element to be deleted, and y, the element replacing x. By definition of $\mathcal{A}$, $s = S \cup \{x, y\}$. Since $x \neq y$ and $x \notin S$, the I transition yields the state $S \cup \{y\}$. The P transition replaces x by y in the array `s`, leaves all other elements of `s` alone, and decrements `scnt` by one. It follows that the rightmost abstraction holds.

ii. $pos(x) = \texttt{scnt} - 1$

Let S be the set consisting of the first $\texttt{scnt} - 1$ elements of `s`. By definition of $\mathcal{A}$, $s = S \cup \{x\}$. The effect of the call on both the I state and P state is shown. Since `s` is not changed and `scnt` is decremented by one, the rightmost abstraction holds.

(b) *exception correctness*

Follows directly from the content lemma.

4. *g_mem*

(a) *output correctness*

Follows directly from the content lemma.

(b) *exception correctness*

Trivially true because *g_mem* signals no exceptions.

7.2 Iterator set proof

We use the Iterator Set (*iset*) module to illustrate the verification of non-deterministic modules. *iset* is an extension of *sset* that allows elements of the set to be retrieved sequentially. The *iset* access programs are shown in Figure 7.3. The module is accessed in set mode (*SET*) or sequential mode (*SEQ*). *s_init* initializes the module, in access mode *SET*. *s_mod*(*mod*) sets

Program name	Inputs	Outputs	Exceptions
s_init			
s_add	integer		*mod* *mem* *full*
s_del	integer		*mod* *notmem*
s_mod	{ *SET*, *SEQ* }		
sg_next		integer	*mod* *end*
g_mem	integer	boolean	*mod*
g_end		boolean	*mod*

Figure 7.3: *iset* access programs

the access mode to *mod*. *s_add*, *s_del*, and *g_mem* behave as specified in *sset* when called in set mode, signalling *mod* when called in sequential mode. Followed by the call *s_mod*(*SEQ*), successive calls to *sg_next* return different elements of the set. *sg_next* signals *mod* when called in set mode, and *end* if there are no more elements to return. *g_end* returns true if all of the elements have been returned, and false otherwise. *g_end* signals *mod* when called in set mode.

Excerpts from the *iset* interface and program specifications are shown in Figure 7.4 and Figure 7.5, respectively. The full work products, including the implementation, are provided in the appendix. We are interested primarily in *sg_next* for the reasons described below. *s_mod* is important because it must be called with the argument *SEQ* before *sg_next* is called.

The proof of transition/output correctness of *sg_next* is complicated by the fact that, as the interface specification shows, this access program is nondeterministic: a call to *sg_next* may return *any* element of the set *is*.

state

s, is: set of integer

m: { SET, SEQ }

access program semantics

s_mod(*mod*):

transition: $(mod = SEQ \rightarrow is, m := s, SEQ$
$\mid mod = SET \rightarrow is, m := \{\}, SET)$

exception: none

sg_next:

transition: $(x \in is \rightarrow is, out := is - \{x\}, x)$

exception: $exc := (m \neq SEQ \rightarrow mod \mid is = \{\} \rightarrow end)$

Figure 7.4: *iset* interface specification

state

`int s[`N`];`

`int scnt,iscnt;`

`enum { SET,SEQ } m;`

state invariant

$\texttt{scnt} \in [0, N] \wedge (\forall i, j \in [0, \texttt{scnt} - 1])(i \neq j \rightarrow \texttt{s}[i] \neq \texttt{s}[j]) \wedge$
$(\texttt{m} = SET \rightarrow \texttt{iscnt} = 0 \mid \texttt{m} = SEQ \rightarrow \texttt{iscnt} \in [0, scnt])$

access program semantics

s_mod(*mod*) :

transition: $(mod = SET \rightarrow \texttt{iscnt}, \texttt{m} := 0, mod$
$\mid mod = SEQ \rightarrow \texttt{iscnt}, \texttt{m} := \texttt{scnt}, mod)$

exception: none

sg_next :

transition-output: $\texttt{iscnt}, out := \texttt{iscnt} - 1, \texttt{s}[\texttt{iscnt} - 1]$

exception: $exc := (\texttt{m} \neq SEQ \rightarrow mod \mid \texttt{iscnt} = 0 \rightarrow end)$

Figure 7.5: *iset* program specification

The program specification (Figure 7.5), on the other hand, is deterministic: a call to *sg_next* always produces the same state/output combination from any given state.

7.2.1 Abstraction function and lemmas

The *iset* abstraction function is defined as follows:

$$\mathcal{A}(\langle \mathtt{s}, \mathtt{scnt}, \mathtt{iscnt}, \mathtt{m} \rangle) = \langle s, is, m \rangle \text{ where}$$
$$s = \{\mathtt{s}[i] \mid i \in [0, \mathtt{scnt} - 1]\} \wedge is = \{\mathtt{s}[i] \mid i \in [0, \mathtt{iscnt} - 1]\} \wedge m = \mathtt{m}$$

The *sset* content lemma is extended as follows.

$$x \in is \leftrightarrow x \in \mathtt{s}[0..\mathtt{iscnt} - 1] \text{ where } \langle s, is, m \rangle = \mathcal{A}(\langle \mathtt{s}, \mathtt{scnt}, \mathtt{iscnt}, \mathtt{m} \rangle)$$

The proof follows directly from the state invariant and the abstraction function.

7.2.2 Proof

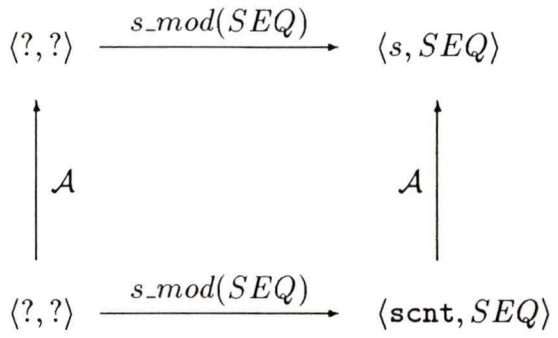
1. *initial correctness*

   Follows directly from the size lemma.

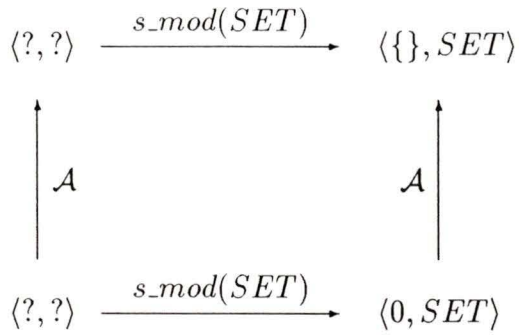
2. *s_mod*

   (a) *transition correctness*

   The state variables s, s, and scnt are not changed by the operation, and the abstraction function defines s strictly in terms of s and scnt; therefore, the task is reduced to proving that the diagram commutes for the remaining variables is, m, iscnt, and m. For convenience, we will represent interface specification states



case i: sequential mode



case ii: set mode

Figure 7.6: Transition correctness: *s_mod*

restricted to is and m as pairs of the form $\langle is, m \rangle$. Similarly, program specification states will be represented as pairs of the form $\langle \texttt{iscnt}, \texttt{m} \rangle$.

There are two cases to consider: changing to sequential mode and changing to set mode. The transition commuting diagram for each case is shown in Figure 7.6. The question marks in the diagram indicate that the initial values of the state variables are irrelevant, since the new values do not depend on them.

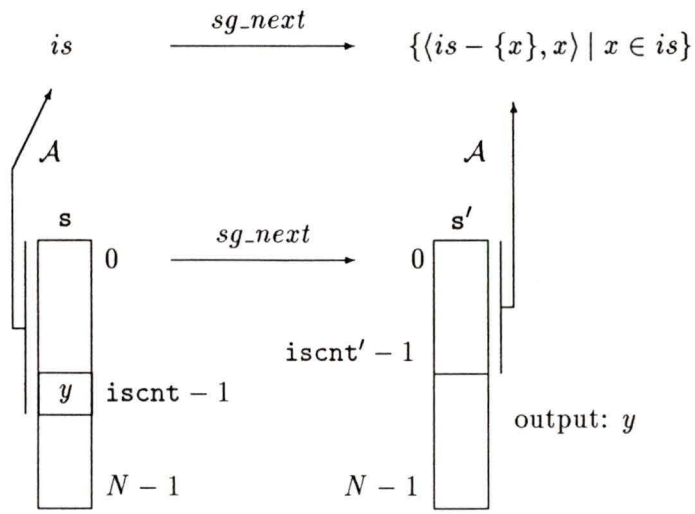


Figure 7.7: Transition-output correctness: *sg_next*

By the size lemma, the final abstraction holds in both cases.

(b) *exception correctness*

Trivially true because *s_mod* signals no exceptions.

3. *sg_next*

(a) *transition/output correctness*

The only state variables effected by *sg_next* are *is* and `iscnt`; all other variables except `s` can be ignored. `s` is relevant because it determines *is*. The transition commuting diagram restricted to these three variables is shown in Figure 7.7.

Let y = `s[iscnt − 1]`, the output defined by the P transition-output. y is one of the elements of *is*, by the state invariant and the extended content lemma. Therefore, y is a potential output of the I transition-output.

P does not change the contents of `s` and decrements `iscnt` by one; therefore, abstracting the new P state yields the set consisting of the first `iscnt` -1 elements of `s`. By the extended content lemma, this set is just $is - \{y\}$, precisely the I state associated with the output y.

(b) *exception correctness*

i. *mod*

Obvious, since the access modes of the program specification state and the abstract state are identical.

ii. *end*

Follows directly from the size lemma.

Chapter 8

Summary and Future Work

8.1 Summary and Discussion

We have pursued three themes: (1) Mealy machines as models for software modules, (2) extensions to the theory of module verification, and (3) verification in inspection meetings.

Many researchers have intuitively viewed modules as state machines. We have proceeded more rigorously, and have found that Mealy machines, as found in the typical automata text, provide an excellent model. Module state machines support software development effectively and map smoothly to Mealy machines. Our notations allow exceptions and nondeterminism to be specified conveniently.

We have exploited Mealy machines to extend the theory of module verification. We view the interface specification, program specification, and implementation work products associated with each module as Mealy machines. We define correctness in terms of the language of a Mealy machine. Our notion of language differs from the usual notion in automata theory. Proving correctness typically involves induction on the length of traces.

Our correctness theorem simplifies correctness proofs by eliminating the

need to use induction, and by dividing the proof up into many smaller assertions which can be proven independently of each other. Our correctness theorem defines conditions, about the abstraction function, interface specification, and program specification, that are sufficient to conclude that the program specification is correct with respect to the interface specification. A commuting diagram is a convenient way of presenting these conditions, and is a useful tool for explaining specific proofs in inspections. Our correctness theorem supports exceptions and nondeterminism. While the work of Gannon, et al. [5] was our starting point, their approach cannot handle many common abstract data types, including *sset* and *iset*.

We have developed verification procedures specifically designed for use in inspection meetings. Our module state machines accurately model interface specifications, program specifications, and implementations from a proven software development approach. We use inspections to provide the social process that DeMillo, et al. [1] claim is essential to successful program proving. Our experience to date suggests that the key contributions to inspections are broad proof strategies. We decompose the proof problem into independent subproofs, with a standard procedure for each. For module M, we separately prove that M's program specification meets its interface specification, and that M's implementation meets its program specification. We attack one access program at a time, separately considering transition, output, and exception correctness. Furthermore, access programs are usually specified as conditional rules with multiple cases which can be proven independently of each other. In practice, the majority of the subproofs are simple enough to be considered obvious, focusing attention on the few that are not.

Commuting diagrams have worked well for presenting proofs of transition

and transition-output correctness. Although we have not used commuting diagrams in proofs of output and exception correctness, they may well be useful when the output or exception specification or code is more complex. When commuting diagrams are used, not all state variables need to be considered. For example, a program specification state variable that does not determine (according to the abstraction function) any interface specification state variable can be ignored. The proof of the access programs *sg_next* and *s_mod* of the *iset* module illustrate this idea.

8.2 Future work

In Chapter 5, we describe a language for specifying module state machines, in particular, interface specifications and program specifications. A number of restrictions are implied by the semantics of this language. For example, we assume that transitions do not occur when exceptions are signaled. To relax this constraint, we might provide a **transition-exception** section or we might simply allow assignments to state variables to occur in the **exception** section. Changing the semantics of our notations would necessitate changing the mapping from MSMs to Mealy machines; it would then be necessary to modify the verification procedures because the soundness of these procedures depends on how we define the mapping.

This thesis has concentrated on program specifications and interface specifications. In the future, we will create procedures for proving the correctness of implementations with respect to program specifications. Clearly, such procedures would depend on the choice of implementation language, C in our case.

We would like to create a taxonomy of verification procedures, based on various combinations of properties of the three work products. To illustrate

this idea, consider an interface specification I, program specification P, abstraction function $\mathcal{A}$, and get program p. A *benevolent side-effect* [14] is a transition from P state q to P state q', caused by a call to p, such that $\mathcal{A}(q) = \mathcal{A}(q')$. Although p is a pure get program in the interface specification, it is a set-get program in the program specification. Thus, we would like to have two verification procedures for get programs. The choice of which one to use depends upon whether the access program is a get program or a set-get program, according to the program specification. We can imagine many other verification procedures that would be useful.

Some additional work on the theory of modules is required. According to our verification procedures, it is necessary to construct a suitable abstraction function to prove that a program specification is correct with respect to an interface specification. For the modules examined in this thesis, this has not been a difficult task; however, there might be some modules for which an abstraction function is difficult, if not impossible, to find. We would like to develop techniques that could identify such modules, and techniques for designing modules in a way that would simplify the task of constructing an abstraction function. For example, we believe that if the number of reachable states of the interface specification is finite and greater than the number of reachable states of the program specification, then an abstraction function that satisfies the conditions of our correctness theorem does not exist.

The notations that we use to specify the effects of access programs (in both the interface and program specifications) could be improved. For example, we have no convenient way of specifying, say in the transition section of an access program, that we do not care which value is assigned to a particular variable. Let us illustrate this problem by considering the *iset* interface specification. Recall that there are two modes, *SET* and *SEQ*, and that the

state variable *is* is used in *SEQ* mode to keep track of which elements of the set have not yet been returned by the iterator, *sg_next*. According to the interface specification, every *SET* mode operation leaves *is* unchanged, as it does not appear on the left-hand side of any assignment statement in the transition sections of these operations. On the other hand, the value of *is* in *SET* mode has no effect on the value assigned to it upon entering *SEQ* mode; therefore, we do not care how *SET* mode operations effect *is*. Thus, we have actually overspecified the *SET* mode operations.

We would like to gain more experience with inspection-based verification. In particular, we do not really understand which kind of people should form a team for inspecting module correctness proofs. Furthermore, it would be beneficial to identify various types of errors that arise in proofs, and to record over a period of time the number of instances of each type. Obviously, a list of the most frequently occurring errors would be beneficial to inspection teams, module designers, programmers, and people responsible for proving module correctness.

Bibliography

[1] R.A. DeMillo, R.J. Lipton, and A.J. Perlis. Social processes and proofs of theorems and programs. *Commun. ACM*, 22(5), May 1979.

[2] R.M. Karp. Reply to 'On the cruelty of really teaching computer science'. *Commun. ACM*, 32(12):1410–1412, December 1989.

[3] R.C. Linger, H.D. Mills, and B.I. Witt. *Structured Programming: Theory and Practice.* Addison-Wesley Publishing Co., 1979.

[4] M.E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, July 1976.

[5] J.D. Gannon, R.G. Hamlet, and H.D. Mills. Theory of modules. *IEEE Trans. Soft. Eng.*, SE-13(7):820–829, July 1987.

[6] David Gries. *The Science of Programming.* Springer-Verlag New York Inc., 1981.

[7] D.L. Parnas and P.C. Clements. A rational design process: How and why to fake it. *IEEE Trans. Soft. Eng.*, SE-12(2):251–257, February 1986.

[8] M.W. Shields. *An Introduction to Automata Theory.* Blackwell Scientific Publications, 1987.

[9] H.D. Mills, M. Dyer, and R.C. Linger. Cleanroom software engineering. *IEEE Software*, pages 19–25, September 1987.

[10] H.D. Mills. The new math of computer programming. *Commun. ACM*, 18(1):43–48, January 1975.

[11] H.D. Mills, V.R. Basili, J.D. Gannon, and R.G. Hamlet. *Principles of Computer Programming: A Mathematical Approach.* Allyn and Bacon, 1986.

[12] M. Dyer and A. Kouchakdjian. Correctness verification: alternative to structural software testing. *Information and Software Technology*, 32(1):53–59, 1990.

[13] R.J. Nelson. *Introduction to Automata.* John Wiley and Sons, Inc., 1968.

[14] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[15] W.A. Wulf, M. Shaw, P.N. Hilfinger, and L. Flon. *Fundamental Structures of Computer Science.* Addison-Wesley, 1981.

[16] C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall, 1986.

[17] J. He, C.A.R. Hoare, and J.W. Sanders. Data refinement refined. In *Proceedings of ESOP86: Lecture Notes in Computer Science No. 213*, pages 187–196. Springer-Verlag, 1986.

[18] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *Data Structures and Algorithms.* Addison-Wesley Publishing Co., 1983.

Appendix - iset work products

Interface specification

state

s, is: set of integer

m: { SET, SEQ }

access program semantics

s_init:

transition: $s, is, m := \{\}, \{\}, SET$

exception: none

$s\_add(x)$:

transition: $s := s \cup \{x\}$

exception: $exc := (m \neq SET \rightarrow mod \mid\mid s \mid = N \rightarrow full \mid x \in s \rightarrow mem)$

$s\_del(x)$:

transition: $s := s - \{x\}$

exception: $exc := (m \neq SET \rightarrow mod \mid x \notin s \rightarrow notmem)$

$g\_mem(x)$:

output: $out := x \in s$

exception: $exc := (m \neq SET \rightarrow mod)$

$s\_mod(mod)$:

transition: $(mod = SEQ \rightarrow is, m := s, SEQ$
$\mid mod = SET \rightarrow is, m := \{\}, SET)$

exception: none

g_end:

output: $is = \{\}$

exception: $exc := (m \neq SEQ \rightarrow mod\ )$

sg_next:

transition: $(x \in is \rightarrow is, out := is - \{x\}, x)$

exception: $exc := (m \neq SEQ \rightarrow mod \mid is = \{\} \rightarrow end\ )$

Program specification

state

`int s[`N`];`
`int scnt,iscnt;`
`enum { SET,SEQ } m;`
state invariant
$\texttt{scnt} \in [0, N] \wedge (\forall i, j \in [0, \texttt{scnt} - 1])(i \neq j \rightarrow \texttt{s}[i] \neq \texttt{s}[j]) \wedge$
$(\texttt{m} = SET \rightarrow \texttt{iscnt} = 0 \mid \texttt{m} = SEQ \rightarrow \texttt{iscnt} \in [0, scnt])$

access program semantics

s_init :
transition: $\texttt{scnt}, \texttt{iscnt}, \texttt{m} := 0, 0, SET$
exception: none

$s\_add(x)$:
transition: $\texttt{s}[\texttt{scnt}], \texttt{scnt} := x, \texttt{scnt} + 1$
exception: $exc := (\texttt{m} \neq SET \rightarrow mod \mid \texttt{scnt} = N \rightarrow full$
$\mid x \in \texttt{s}[0..\texttt{scnt} - 1] \rightarrow mem)$

$s\_del(x)$:
transition:
$(pos(x) < scnt - 1 \rightarrow \texttt{s}[pos(x)], \texttt{scnt} := \texttt{s}[\texttt{scnt} - 1], \texttt{scnt} - 1$
$\mid pos(x) = scnt - 1 \rightarrow \texttt{scnt} := \texttt{scnt} - 1)$
where $pos(x)$ is the index of x in $\texttt{s}[0..\texttt{scnt} - 1]$
exception: $exc := (\texttt{m} \neq SET \rightarrow mod$
$\mid x \notin \texttt{s}[0..\texttt{scnt} - 1] \rightarrow notmem)$

$g\_mem(x)$:
output: $out := x \in \texttt{s}[0..\texttt{scnt} - 1]$
exception: $exc := (\texttt{m} \neq SET \rightarrow mod)$

$s\_mod(mod)$:
transition: $(mod = SET \rightarrow \texttt{iscnt}, \texttt{m} := 0, mod$
$\mid mod = SEQ \rightarrow \texttt{iscnt}, \texttt{m} := \texttt{scnt}, mod)$
exception: none

g_end :
output: $out := \texttt{iscnt} = 0$
exception: $exc := (\texttt{m} \neq SEQ \rightarrow mod)$

sg_next :
transition-output: $\texttt{iscnt}, out := \texttt{iscnt} - 1, \texttt{s}[\texttt{iscnt} - 1]$
exception: $exc := (\texttt{m} \neq SEQ \rightarrow mod \mid \texttt{iscnt} = 0 \rightarrow end)$

Implementation

In this implementation, the following exception-signaling method is used: exception e is signaled by invoking the C function e, to be implemented by the module user.

```
static int findpos(int x)
{    int i;
     for (i = scnt-1; i >= 0; i--) if (s[i] == x) return(i);
     return(scnt); /*not found*/ }

void s_init() { scnt = 0; iscnt = 0; m = SET }

void s_add(int x)
{    if (mod != SET) mod();
     else if (scnt == N) full();
     else if (findpos(x) != scnt) mem();
     else s[scnt++] = x; }

void s_del(int x)
{    int i;
     if (mod != SET) mod();
     else {
          i = findpos(x);
          if (i == scnt) notmem();
          else if (i < --scnt) s[i] = s[scnt]; }

int g_mem(int x)
{    if (mod != SET) { mod(); return(0); }
     return(findpos(x) != scnt); }

void s_mod(mod)
modtyp mod;
{    m = mod;
     if (m == SET) iscnt = 0;
     else iscnt = scnt; }

int g_end()
{    if (m != SEQ) { mod(); return(0); }
     return(!iscnt); }
```

```
void sg_next()
{    if (mod != SEQ) { mod(); return(0); }
     if (iscnt == 0) { end(); return(0); }
     iscnt--; return(s[iscnt]); }
```

VITA

Surname: Jones Given Names: Graeme Neal

Place of Birth: Fairview, Alberta Date of Birth: 24 January 1966

Educational Institutions Attended:

University of Victoria	1990 to 1992
University of Alberta	1983 to 1987

Degrees Awarded:

B.Sc. (Honours)	University of Alberta	1987

Honours and Awards:

NSERC Postgraduate Scholarship	1987
I.P. Sharp Associates Limited Scholarship	1985
Norcen Energy Resources Scholarship	1984

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis: An Inspection-based Technique for Verifying Module Correctness

Author: [redacted]

GRAEME NEAL JONES

April 23, 1992