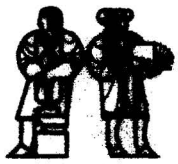
LABORATORY FOR
COMPUTER SCIENCE



MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-202

A FORMALIZATION OF THE STATE MACHINE SPECIFICATION TECHNIQUE

Robert N. Principato, Jr.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

This blank page was inserted to preserve pagination.

MIT/LCS/TR-202

A Formalization of the State Machine Specification Technique

by

Robert Nelson Principato, Jr.

May 1978

This research was supported in part by the National Science Foundation under grant MCS74-21892 A01.

Massachusetts Institute of Technology
Laboratory for Computer Science

Cambridge Massachusetts 02139

A Formalization of the State Machine Specification Technique

by

Robert Nelson Principato, Jr.

Submitted to the Department of Electrical Engineering and Computer Science
on May 16, 1978 in partial fulfillment of the requirements for the
degrees of Master of Science and Electrical Engineer

Abstract

This thesis develops the state machine specification technique, a formal specification technique for data abstractions based on Parnas' work on specifying software modules. When using the state machine technique, each data object is viewed as the state of an abstract (and not necessarily finite) state machine and, in the state machine, this state set is implicitly defined. The basic idea is to separate the operations of the data abstraction into two distinct groups; those which do not change the state but allow some aspect of the state to be observed, the value returning or V-functions, and those which change the state, the operation or O-functions. The specifications are then written by stating the effect of each O-function on the result of each V-function. This implicitly defines the smallest set of states necessary to distinguish the variations in the results of the V-functions. It also determines the transitions among these states caused by the O-functions.

An abstract model for the semantics of state machine specifications is presented and then used to formalize the semantics of a concrete specification language. Furthermore, a methodology for proving the correctness of an implementation of a data abstraction specified by a state machine is discussed and illustrated.

Key Words and Phrases: state machine specifications, data abstractions, formal specification technique, proofs of correctness

Name and Title of Thesis Supervisor: Barbara H. Liskov
Associate Professor of Computer Science and Engineering

Acknowledgements

I wish to express my thanks to all the people who helped and encouraged this work: to Professor Barbara Liskov who patiently read my many drafts and helped me clarify my ideas; to Deepak Kapur who took the time to read several drafts of this thesis and made so many valuable suggestions; to Bob Scheifler who taught me how to use ITS and wrote R macros for me; and finally to Bart DeWolf and Frank Bamberger who helped me in the early phases of this work.

CONTENTS

FIGURES

1. Introduction

1.1 Motivation

In the development of our understanding of complex phenomena, the most powerful tool available to enhance our comprehension is abstraction. Abstraction arises from the recognition of similarities between certain objects or processes, and the decision to concentrate on these correspondences and to ignore, for the present, their differences [Hoare 72b]. In focusing on similarities, one tends to regard them as fundamental and intrinsic, and to view the differences as trivial.

One of the earliest recognized and most useful aids to abstraction in programming is the self-contained subroutine or procedure. Procedures appeared as early as 1945 in Zuse's programming language, Plancalculus [Knuth 76]. Besides, early developers of programming languages recognized the utility of the concept of a procedure. Curry, in 1950, described the advantages of including procedures in the programming languages being developed at that time by pointing out that the decomposition mechanism provided by a procedure would allow keener insight into a problem by permitting consideration of its separate, distinct parts [Curry 50].

The existence of procedures goes quite far toward capturing the meaning of abstraction [Liskov and Zilles 74]. At the point of its invocation, a procedure may be treated as a "black box", that performs a specific function by means of an unprescribed algorithm. Thus, at the level of its invocation, a procedure separates the relevant detail of what it accomplishes from the irrelevant detail of how it is implemented. Furthermore, at the level of its implementation, a procedure facilitates understanding of how it accomplishes its task

by freeing the programmer from considering why it is invoked.

However, procedures alone do not provide a sufficiently rich vocabulary of abstractions [Liskov and Zilles 75]. Procedures, while well suited to the description of abstract processes or events, do not accommodate the description of abstract objects. To alleviate this problem, the concept of a *data abstraction* was introduced. This comprises a group of related functions or operations which act upon a particular class of objects with the constraint that objects in this class can only be observed or modified by the application of its related operations [Liskov and Zilles 75].

A typical example of a data abstraction is an *integer push down stack*. Here, the class of objects consists of all possible stacks and the collection of related operations includes the usual stack operations, like *push* and *pop*, an operation to create new stacks, and an operation, *top*, to return the integer on top of the stack.

The set of operations associated with a data abstraction will, in general, include operations to create objects of the data abstraction, operations to modify objects of the data abstraction and operations to obtain information about the structure or contents of objects of the data abstraction. The first two categories of operations, which include *push* and *pop*, are the *constructors* of the data abstraction. Operations in the last category are *inquiry* operations as they provide information about the data abstraction. *Top* belongs to this category.

Constructors can be further classified into two different groups; *information adding operations* and *information removing operations*. Information adding operations place new information in the data abstraction. For example, *push* is an information adding operation for *integer push down stack*. Its complement, *pop*, is an information removing operation.

This type of operation removes information from an object of the data abstraction and results in a new object of the data abstraction whose information content is a subset of the information content of the original object [Kapur 78].

A data abstraction provides the same aids to abstraction as a procedure and allows one to separate the implementation details of a data abstraction from its behavior. The behavior of a data abstraction can be described by a *specification*. A specification of a data abstraction specifies the names and defines the abstract meaning of the associated operations of the data abstraction. It describes what the data abstraction does but not how it is done. This latter task is accomplished by an *implementation*. An implementation of a data abstraction describes the representation of objects of the data abstraction and the implementation of the operations that act upon these objects. Though these different attributes of specification and implementation are, in practice, highly interdependent, they represent logically independent concepts [Guttag 75].

The main concern of this thesis is the specification of data abstractions. Specification is important because it describes the abstract object which has been conceived in someone's mind. It can be used as a communication medium among designers and implementors to insure that an implementor understands the designer's intentions about the data abstraction he is coding [Liskov and Zilles 75].

Moreover, if a formal specification technique, one with an explicitly and precisely defined syntax and semantics, is used, even further benefits can be derived. Formal specifications can be studied mathematically so that questions, such as the equivalence of two different specifications, may be posed and rigorously answered. Also, formal specifications can serve as the basis for proofs of correctness of programs. If a programming language's

semantics are defined formally [Milne and Strachey 76], properties of a program written in this language can be formally proved. The correctness of the program can then be proved by establishing the equivalence of these properties and the specification. Finally, formal specifications can be meaningfully processed by a computer [Liskov and Zilles 75], [Liskov and Berzins 77]. Since this processing can be done in advance of implementation, it can provide design and configuration guidelines during program development.

1.2 Parnas's Approach to Specification

The information contained in the specification of a data abstraction can de divided into a syntactic part and a semantic part [Liskov and Zilles 75]. The syntactic part provides a vocabulary of terms or symbols that are used by the semantic part to express the actual meaning or behavior of the data abstraction. Two different approaches are used in capturing this meaning; either an explicit, abstract model is supplied for the class of objects and its associated operations are defined in terms of this model, or the class of objects is defined implicitly via descriptions of the operations [Liskov and Zilles 76].

Parnas [Parnas 72] has developed a technique and notation for writing specifications based on the implicit approach. His specification scheme was devised with the following goals in mind [Parnas 72]:

> 1) The specification must provide to the intended user all the information that he will need to correctly use the object specified, and nothing more.

> 2) The specification must provide to the implementor all the information about the intended use of the object specified that he needs to implement the specification, and no additional information.
>
> 3) The specification should discuss the object specified in the terms normally used by user and implementor alike rather than in some other area of discourse.

When using Parnas's technique, each data object is viewed as the state of an abstract (and not necessarily finite) state machine and, in Parnas's specifications, this state set is implicitly defined. The basic idea is to separate the operations of the data abstraction into two distinct groups; those which do not change the state but allow some aspect of the state to be observed, the value returning or *V-functions*, and those which change the state, the operation or *O-functions*. The specifications are then written by stating the effect of each O-function on the result of each V-function. This implicitly defines the smallest set of states necessary to distinguish the variations in the results of the V-functions [Liskov and Zilles 75]. It also determines the transitions among these states caused by the O-functions.

Returning to the *integer push down stack* example, consider the operations *top* and *push*. *Top* is a V-function that is defined as long as the stack is not empty, and *push* is an O-function that effects the result of *top*. These operations might be specified as in Figure 1, where *depth* is another V-function whose definition is not shown here, but reflects the number of integers in the stack. Quotes around a V-function are used to indicate its value after the O-function is executed.[1]

A problem with this approach is that certain O-functions may have delayed effects

---

1. This interpretation of quotes differs from that in [Parnas 72, 75].

Figure 1. Top and Push

```
top = V-function( ) returns integer
        Applicability Condition: depth ≠ 0
        Initial Value: undefined
        end top

push = O-function(a:integer)
        Applicability Condition: depth < 100
        Effects Section: 'top' = a
        'depth' = depth + 1
        end push
```

---

on the V-functions. In other words, some property of the state will be observed by a V-function only after some O-function has been used. For example, *push* has a delayed effect on *top* in the sense that after a new element has been pushed on the stack, the former top of the stack element is no longer observable by *top* but it will be if *pop* is used.

Parnas used an informal language to express these delayed effects [Parnas 72,75]. In his specifications, he included a section, called *module properties*, for describing delayed effects in English, at times interlaced with simple mathematical formulae [Parnas 75]. For example, to specify the interaction of *push* and *pop* on a stack, Parnas used the phrase "The sequence PUSH(a);POP has no net effect if no error calls occur" [Parnas 75].

One method to formally describe delayed effects is to introduce hidden V-functions [Price 73] to represent aspects of the state which are not immediately observable. Hidden V-functions are not operations associated with the data abstraction being defined. They are introduced to store values of other V-functions and in this manner they solve the representational problems caused by delayed effects. Since they are not operations of the

data abstraction, users of the abstraction should not be able to use them. As an example, in the specification of a push down stack, one could introduce a hidden V-function *stack* to store the former top of the stack element.

This approach has been followed by researchers at the Stanford Research Institute [Robinson 77], [Spitzen 76]. However, their main concern with Parnas's approach to specification is its use in a general methodology for the design, implementation and proof of large software systems [Robinson 75], [Neumann 74]. With this goal in mind, they have designed a specification language, called SPECIAL, for describing Parnas-type specifications [Roubine 76]. But, no formal semantics have been provided for SPECIAL.

1.3 State Machine Specifications

This thesis develops a formal specification technique based on Parnas' ideas. The specifications written using this technique are called *state machine specifications* and employ hidden V-functions. The specification technique described in this thesis is similar to work being done at the Stanford Research Institute. No attempt is made to formalize Parnas' notion of a modular properties section.

An example of a state machine specification is given below in Figure 2. Here, the data abstraction defined is a *bounded integer stack* with the following operations. *Top* is a V-function that is defined as long as the stack is not empty and returns the top of the stack. *Depth* is another V-function that reflects the number of integers in the stack. *Push* and *pop* are O-functions that insert and delete, respectively, integers from the top of the stack.

Notice that there are three different types of V-functions included in the specification. The *hidden* V-functions are used to represent aspects of the state that are not

Figure 2. Bounded Integer Stack

```
bounded_stack = state machine is push, pop, top, depth

        depth = non-derived V-function( ) returns integer
                Appl. Cond.: true
                Initial Value: 0
                end depth

        stack = hidden V-function(i:integer) returns integer
                Appl. Cond.: 1 ≤ i ≤ depth
                Initial Value: undefined
                end stack

        top = derived V-function( ) returns integer
                Appl. Cond.: ~(depth = 0)
                Derivation: top = stack(depth)
                end top

        pop = O-function( )
                Appl. Cond.: ~(depth = 0)
                Effects: 'depth' = depth - 1
                end pop

        push = O-function(a:integer)
                Appl. Cond.: depth < 100
                Effects: 'depth' = depth + 1
                'stack'(depth + 1) = a
                end push

end bounded_stack
```

---

immediately observable. Recall the delayed effect of *push* on *top*. When a new element is pushed on the stack, the former top of stack element is no longer observable by *top* but it will be if *pop* is used. This value is stored in the hidden V-function *stack*. Hidden V-functions are not directly accessible to users of the data abstraction, but limited access to them is provided by the *derived* V-functions, which are defined in terms of the hidden and

non-derived V-functions. Non-derived V-functions are also accessible to users of the data abstraction. They are inquiry operations that reveal intrinsic aspects of the data abstraction defined by the specification.

Note that the specification in Figure 2 uses two data abstractions, namely the integers and Booleans, which are distinct from the data abstraction defined by the machine. These data abstractions are called the *defining abstractions.* They are not restricted to contain only the integers and Booleans and can consist of an entire collection of data abstractions. The defining abstractions are usually simple abstractions that are used to construct more complicated state machine specifications.

The defining abstractions are used in the domain and range of the V-functions and O-functions. They constitute the information that the O-functions, the constructors, add or remove from the data abstraction. They are also the results that the V-functions, the inquiry operations, return. The defining abstractions are assumed to be defined elsewhere either by state machines or some other formal specification technique.

The semantics of a state machine can be defined by giving the following interpretation to the V-functions and O-functions. In every state of the machine, some mapping is associated with each V-function. These mappings characterize the state. They represent the information that the V-functions reveal about each state. In fact, since the derived V-functions are defined in terms of the non-derived and hidden V-functions, the state of a state machine is completely characterized by the mappings of the non-derived and hidden V-functions. The O-functions change the state of the machine by redefining these mappings.

1.4 Uses of State Machines

As was previously discussed, formal specifications can be studied mathematically. So, state machine specifications can be used to prove properties of data abstractions or the equivalence of different specifications. Furthermore, they can be used as an unambiguous communications medium among programmers due to their precisely defined semantics. But one of their most important uses will be to serve as the basis for proofs of program correctness.

Establishing program correctness can be described as a two step process with the overall goal of showing that a program correctly implements a concept that exists in someone's mind. First, a formal description of the concept is needed. This can be done by a formal specification. Then, the program is proved equivalent to the specification by formal, analytic means. [Hoare 72a] has described a method to accomplish this latter task.

However, Hoare's method requires some adaptations to meet the special needs of state machines. Accordingly, this thesis also discusses these changes and how to perform proofs of correctness using state machines.

1.5 The Outline of the Thesis

Chapter 2 presents a model for the semantics of state machine specifications. First, the basic components that every state machine must contain are discussed. Then these basic components are used to develop a model for the semantics of a state machine. The discussion in this chapter is abstract, presenting only the objects that the basic components of any state machine must specify but not discussing an actual language to specify these objects.

Hence, the model developed is quite general and not tied to a particular specification language. However, this model is restricted to state machines that only contain unary operations on the data abstraction defined by the machine.

Chapter 3 details an actual specification language for state machines. It is a complement to the abstract discussion in Chapter 2 and uses the model developed in Chapter 2 to formalize the semantics of this concrete specification language.

Chapters 4 and 5 discuss and illustrate a method to prove the correctness of an implementation of a data abstraction specified by a state machine.

Chapter 6 extends the model for the semantics of state machines described in Chapter 2 by lifting the restriction to unary operations.

Chapter 7 concludes this thesis with an evaluation of the work presented and some suggestions for extensions to the state machine specification technique.

2. A Model for State Machines

This chapter presents a model for the semantics of state machine specifications. In Section 2.1, the basic components that every state machine specification must contain are discussed. Section 2.1 only defines the syntactic constraints that a state machine specification must satisfy. Semantic issues concerning whether the machine is well-defined or consistent are discussed in Section 2.2, which shows how these basic components can be used to develop a model for the semantics of a state machine. Here, each machine is modelled by a set of states, where each state is modelled by a set of functions corresponding to the hidden and non-derived V-functions; O-functions define transitions between states.

The discussion here is abstract, presenting only the objects that the basic components of any state machine must specify but not discussing the actual language used to specify these objects. Hence, the model developed here is quite general and applicable to any state machine specified using a combination of V-functions and O-functions. It is not, however, applicable to state machines specified using something similar to Parnas's modular properties section.

2.1 The Basic Components of a State Machine

The state machines considered here are specified using V-functions and O-functions. In principle, one could define a state machine without any V-functions. Such a specification, however, would be singularly uninteresting. Without V-functions there would be no way to observe the state of the machine and, hence, no way to distinguish one member of the data abstraction defined by the machine from any other member. So, we

shall assume all state machines have one or more V-functions.

Furthermore, most interesting state machine specifications will contain one or more O-functions since, without O-functions, a state machine can only specify a data abstraction containing exactly one element.

2.1.1 V-functions

As was discussed in Chapter 1, there are three types of V-functions; the non-derived V-functions and the hidden V-functions, which are primitive, and the derived V-functions, which are not primitive but are defined in terms of the other two.

2.1.1.1 Non-derived and Hidden V-functions

Non-derived and hidden V-functions are specified analogously. Each non-derived or hidden V-function v has three sections in its definition: a mapping description, an applicability condition and an initial value section.

Figure 3. Non-derived or hidden V-function v

Mapping Description: D_v: R_v
Applicability Condition: $\mathbb{A}_v$: $\mathbb{D} \times D_v \rightarrow$ Boolean
Initial Value: $\underline{init}_v \epsilon [D_v \rightarrow R_v]$

First, let $[A \rightarrow B]$ denote the set of partial functions from the set A to the set B. In each state S of the state machine, some particular mapping v_S from $[D_v \rightarrow R_v]$ will be associated with v, where D_v and R_v are specified by the V-function's *mapping description.*

This mapping, of course, varies with the state of the machine. In general, the mapping associated with v will not be total. As an example, in Figure 2 of Chapter 1, any mapping associated with *stack* is a member of [integer → integer].

The sets D_v and R_v also carry the following stipulation regarding the data abstraction defined by the machine. In general, they will be the cartesian product $G_1 \times ... \times G_n$ of a group of sets. But the G_i are restricted so that no element of the data abstraction defined by the machine may be an element of any of the G_i. This restriction only allows the definition of unary operations on the data abstraction specifed by the machine. For example, in the definition of the data abstraction *integer set*, it is not possible to define a function which computes the union of two sets. But, it is possible to define the unary operation, *has*, which determines if a set contains a given integer.

Now, since the state of the machine is characterized by a set of mappings associated with each non-derived and hidden V-function, we can view the state set $\mathbb{S}$ as a subset of

$$[D_{v_1} \rightarrow R_{v_1}] \times ... \times [D_{v_n} \rightarrow R_{v_n}] = \mathbb{D}$$

where $\{v_1,...,v_n\}$ is the set of non-derived and hidden V-functions of the machine. In most cases, $\mathbb{S}$ is a proper subset of $\mathbb{D}$. This occurs when an n-tuple of $\mathbb{D}$ contains, as an element, a function that can never be associated with a non-derived or hidden V-function. For example, in the *bounded stack* example of Chapter 1, a total function from the integers into the integers can never be associated with *stack*.

The *applicability condition* of a V-function governs when a call of that function by a user of the machine succeeds. This section specifies a partial function $\mathbb{A}_v$ from $\mathbb{D} \times D_v$ into the Booleans. Hence, the success of a call depends on the state of the machine. For any $x \in D_v$ and $S \in \mathbb{S}$, $\mathbb{A}_v(S,x)$ must evaluate to true for the V-function to return the value $v_S(x)$,

where v_S denotes the mapping associated with v in state S. When $\mathfrak{A}_v(S,x)$ equals **false**, v returns an error condition.

The *initial value section* of a non-derived or hidden V-function v defines the mapping associated with v in the initial state of the machine. This section specifies one member, denoted $\underline{init}_v$, of $[D_v \rightarrow R_v]$. In practice, for non-derived V-functions, $\underline{init}_v$ is usually a constant, total function.

2.1.1.2 Derived V-functions

A derived V-function v also has three sections in its definition: a mapping description, an applicability condition and a derivation section. The *mapping description* and *applicability condition* are defined in the same manner and have the same interpretation as the mapping description and applicability section of a non-derived or hidden V-function. The derivation section is unique to this type of function.

Figure 4. Derived V-function v

Mapping Description: D_v; R_v
Applicability Condition: $\mathfrak{A}_v$: $\mathfrak{D} \times D_v \rightarrow$ Boolean
Derivation: $\underline{der}$ v such that $(\underline{der}\ v_S) \in [D_v \rightarrow R_v]$ for states S

The *derivation section* specifies the mapping associated with v in terms of the mappings associated with the hidden and non-derived V-functions. This section defines a function schema, denoted $\underline{der}$ v, expressed as the composition of the non-derived and hidden

V-functions of the machine and other functions associated with the elements of D_v. The particular mapping associated with the schema, denoted $(\underline{der}\ v_S)$, depends on the state S of the machine, which contains an interpretation for the non-derived and hidden V-functions. As an example, consider the derivation section of *top* in Figure 2 of Chapter 1. In any state S, *top* returns the value *stack(depth)*. This value is, of course, dependent on the mappings associated with *stack* and *depth* in state S.

2.1.2 O-functions

O-functions too have three sections in their definition. They are a mapping description, an applicability condition and an effects section.

Figure 5. O-function o

Mapping Description: D_o
Applicability Condition: $\mathfrak{A}_o$: $\mathfrak{D} \times D_o \rightarrow$ Boolean
Effects Section: $\mathfrak{T}_o$: $\mathfrak{D} \times D_o \rightarrow \mathfrak{D}$

In a given state, each O-function o is a member of $[D_o \rightarrow \mathfrak{S}]$, where D_o is given by the *mapping description* and $\mathfrak{S}$ is the state set of the machine. As with V-functions, D_o will, in general, equal the cartesian product of a group of sets $G_1 \times \ldots \times G_n$ which are constrained so that no element of the data abstraction defined by the machine may be an element of any of the G_i. The range of the O-function is not specified by the mapping description since it is understood that the range of all O-functions is the state set.

The *applicability condition* of an O-function determines when the O-function changes the state of the machine. As for V-functions, this section defines a partial function $\mathfrak{A}_o$ from $\mathfrak{D} \times D_o$ into the Booleans. $\mathfrak{A}_o$ must evaluate to true for the function to change the state of the machine. Otherwise, an error condition is raised and the state remains unchanged. For example, the applicability condition of *pop* in Figure 2 of Chapter 1 prohibits its execution when the stack is empty.

The *effects section* of an O-function specifies how the function changes the state of the machine. This section defines a partial function $\mathfrak{T}_o$ from $\mathfrak{D} \times D_o$ into $\mathfrak{D}$.

2.2 The Semantics of a State Machine

2.2.1 The State Set of a State Machine

As was previously mentioned, a state of a state machine is modelled by mappings associated with each non-derived and hidden V-function of the machine. Hence, we view the state set, $\mathfrak{S}$, of a state machine in the following manner:

$$\mathfrak{S} \subset [D_{v_1} \rightarrow R_{v_1}] \times \ldots \times [D_{v_n} \rightarrow R_{v_n}] = \mathfrak{D}$$

where $\{v_1,\ldots,v_n\}$ is the set of non-derived and hidden V-functions of the machine.[1] Note that D_{v_i} and R_{v_i} are specified by v_i's mapping description.

Our purpose in this section is to define $\mathfrak{S}$. Here, a constructive approach will be used. Note that the initial state of a state machine is explicitly defined by the initial value sections of the non-derived and hidden V-functions. This initial state, Q, can generate the state set by means of the following construction:

---

1. Recall $[A \rightarrow B] = \{f \mid f \text{ is a partial function from } A \text{ to } B\}$

1) Q is an element of $\mathfrak{S}$.

2) If S is an element of $\mathfrak{S}$ and o is an O-function call,
then the state S^* obtained by applying o to S is an element of $\mathfrak{S}$.

3) These are the only members of $\mathfrak{S}$.

So, to define $\mathfrak{S}$, it suffices to define the initial state of the machine and then to describe the state changes caused by O-function calls or, in general, how an O-function call maps one member of $\mathfrak{D}$ into another.

The *initial state* Q is the tuple $(\underline{init}_{v_1},\ldots,\underline{init}_{v_n})$ containing the mappings derived from the initial value section of each of the non-derived and hidden V-functions $(v_1,\ldots,v_n)$.

Furthermore, the next state function has the following definition.

Definition
Let o be an O-function with mapping $\mathfrak{A}_o$ in its applicability condition and mapping $\mathfrak{T}_o$ in its effects section.
Let $a \in D_o$ and $R \in \mathfrak{D}$.
Then,

$$\text{NEXT}(R,o,a) = \begin{cases} \mathfrak{T}_o(R,a) & \text{if } \mathfrak{A}_o(R,a)\text{=true} \\ R & \text{if } \mathfrak{A}_o(R,a)\text{=false} \end{cases}$$

Thus, the state set is generated as follows.

1) $Q \in \mathfrak{S}$.

2) If $R \in \mathfrak{S}$ and o is an O-function, then if NEXT(R,o,a) is defined, $\text{NEXT}(R,o,a) \in \mathfrak{S}$ where $a \in D_o$.

3) These are the only elements of $\mathfrak{S}$.

In other words, the state set $\mathfrak{S}$ is the closure of Q under the state transition function associated with the O-functions. Note that in 2) above NEXT(R,o,a) may be undefined. This depends on the functions $\mathfrak{T}_o$ and $\mathfrak{A}_o$.

Recall that $\mathfrak{T}_o$ is a partial function. So, it is possible for some state S and $x \in D_o$ that $\mathfrak{T}_o(S,x)$ is undefined. Then, if $\mathfrak{A}_o(S,x)$=true, NEXT(S,o,x) would be undefined. This situation is undesirable since when $\mathfrak{A}_o(S,x)$=true, a state change should occur. Furthermore, $\mathfrak{A}_o$ is also a partial function. Here, it is possible for some state S' and $x' \in D_o$ that $\mathfrak{A}_o(S',x')$ is undefined, again making NEXT(S',o,x') undefined. These two considerations lead us to the notion of a well-defined state machine.

Definition
A state machine is *well-defined* if for any $S \in \mathfrak{S}$ and O-function o NEXT(S,o,a) is defined where $a \in D_o$.

This definition guarantees that in a well-defined state machine, for every O-function o, $\mathfrak{A}_o$ is a total function from $\mathfrak{S} \times D_o$ into the Booleans and $\mathfrak{T}_o$ is a total function from $\{(S,a) \in \mathfrak{S} \times D_o \mid \mathfrak{A}_o(S,a)\}$ into $\mathfrak{S}$. This can be seen by inspection of the definition of NEXT.

2.2.2 The Semantics of V-functions and O-functions

With this definition of the state set $\mathfrak{S}$ of a state machine specification, it is possible to formally define the meaning of the O-functions and V-functions. This will be done by defining mappings V-Eval for V-functions and O-Eval for O-functions such that

$$\text{V-Eval}: \mathcal{S} \times NV \rightarrow [A \rightarrow R]$$

and

$$\text{O-Eval}: \mathcal{S} \times NO \rightarrow [A \rightarrow \mathcal{S}]$$

where NV is the set of V-function names, A is the set of arguments, R is the set of results and NO is the set of O-function names.

First, it is necessary to deal with some notational details. Here, the notation "$\mathcal{B} \rightarrow x,y$" has the value x if $\mathcal{B}$ is true and the value y if $\mathcal{B}$ is false. This notation will be used to raise an error condition when a function's applicability condition is not satisfied.

O-Eval will be defined first. Now, given a state S and an O-function o, O-Eval returns a function from D_o into $\mathcal{S}$ U {error}. So, using lambda notation,

$$\text{O-Eval}(S,o) = \lambda a.[\mathcal{A}_o(S,a) \rightarrow \text{NEXT}(S,o,a), \underline{\text{error}}]$$

O-Eval(S,o) is not necessarily total since either $\mathcal{A}_o(S,a)$ or NEXT(S,o,a) can be undefined. However, O-Eval(S,o) is always a total function in a well-defined state machine.

For any V-function v and state S, V-Eval will return a function from D_v into R_v U {error}. First, for a non-derived or hidden V-function v and a state S, recall that v_S denotes the function associated with v in state S. Then for any non-derived or hidden V-function v with applicability condition $\mathcal{A}_v$,

$$\text{V-Eval}(S,v) = \lambda a.[\mathcal{A}_v(S,a) \rightarrow v_S(a), \underline{\text{error}}]$$

Finally, for a derived V-function v with applicability condition $\mathfrak{A}_v$ and derivation der v,

$$V\text{-Eval}(S,v) = \lambda a.[\mathfrak{A}_v(S,a) \rightarrow (\underline{der}\ v_S)(a), \underline{error}]$$

Note that V-Eval(S,v) is not necessarily defined over the entire set D_v since $\mathfrak{A}_v(S,a)$ can be undefined or, depending on the type of V-function, $v_S(a)$ or $(\underline{der}\ v_S)(a)$ can be undefined when $\mathfrak{A}_v(S,a)$=true. When this is not the case, we say the state machine is consistent.

Definition
A state machine is *consistent* if V-Eval(S,v) is a total function from D_v into $R_v \cup \{\underline{error}\}$ for every state $S \in \mathfrak{S}$ and V-function v.

In a consistent state machine, $\mathfrak{A}_v$ is always a total function from $\mathfrak{S} \times D_v$ into the Booleans and v_S or $(\underline{der}\ v_S)$ is always a total function from $\{x \in D_v \mid \mathfrak{A}_v(S,x)\}$ into R_v.

2.2.3 An Induction Principle

Since any state of a state machine is generated by zero or more O-function calls, the structural induction principle [Burstall 69] holds here. In structural induction, proofs proceed by course of values induction on the complexity of the structure,[2] which, for state machines, means that to prove the data abstraction defined by the machine has property P, one must prove that the initial state has property P, and that if all states produced by zero through n-1 O-function calls have P, the P is true after n O-function calls. This is one

2. The general schema of *course of values* induction on the natural numbers is:
$P(0), \forall j(\forall i((i<j \wedge P(i)) \rightarrow P(j)) \vdash \forall k P(k)$

advantage of the generative approach used in this model to define the state set.

2.2.4 Proving Properties of State Machines

Although it is not possible to establish formally that a state machine specification is correct with respect to our intuition, there are certain properties that a specification should satisfy to enhance our confidence in its correctness. Two important properties of a state machine are whether or not it is well-defined or consistent. In a well-defined machine, the O-functions behave properly, either changing the state or informing the user of an error. In a consistent machine, the same is true of the V-functions. They either return a value or raise an error condition.

A state machine is well-defined when NEXT is a total function. This occurs when, for every O-function o, $\mathfrak{A}_o$ is a total function from $\mathfrak{S} \times D_o$ into the Booleans and $\mathfrak{T}_o$ is a total function from $\{(S,a) \in \mathfrak{S} \times D_o \mid \mathfrak{A}_o(S,a)\}$ into $\mathfrak{S}$.

Since $\mathfrak{S}$ is defined generatively, a state machine can be proved to be well-defined by using structural induction. As outlined in Section 2.2.3, this involves first showing that NEXT(Q,o,a) is defined for all O-functions o and $a \in D_o$ and then assuming

$$NEXT(...NEXT(NEXT(Q,o_1,a_1),o_2,a_2),...,o_{n-1},a_{n-1})$$

is defined for all $a_i \in D_{o_i}$, $n \geq 2$ and then proving that

$$NEXT(...NEXT(NEXT(Q,o_1,a_1),o_2,a_2),...,o_n,a_n)$$

is defined for all $a_i \in D_{o_i}$. In practice, however, it may be necessary to strenghten the inductive hypothesis to simplify the proof.

A state machine is consistent when, for every V-function v, $\mathfrak{A}_v$ is a total function from $\mathfrak{S} \times D_v$ into the Booleans and, for every non-derived or hidden V-function v and state $S \in \mathfrak{S}$, v_S is a total function from $\{a \mid \in D_v$ and $\mathfrak{A}_v(S,a)\}$ into R_v and for every derived V-function v, (der v_S) is a total function from $\{a \mid a \in D_v$ and $\mathfrak{A}_v(S,a)\}$ into R_v. All these properties can be established by using structural induction in the manner outlined above.

In general, for most practical specifications, the task of proving that a state machine is well-defined or consistent is not extremely difficult but rather tedious due to the many cases that must be verified. The hardest step in a proof usually involves discovering an inductive hypothesis that allows the proof to follow readily. These comments are illustrated by the example in Section 3.3 of Chapter 3 where a specification of a queue is shown to be well-defined and consistent.

Note, however, that both the problems of determining whether or not an arbitrary state machine is well-defined and determining whether or not an arbitrary state machine is consistent are undecidable. This situation arises since both problems can be reduced to the halting problem for Turing machines [Hennie 77]. These two results are established for the specification language of Chapter 3 in Appendix 1. However, they are not language dependent.

The reductions for both problems are similar. Below, the reduction for the question of determining whether or not a state machine is well-defined is sketched. Here, we shall actually reduce this problem to the blank tape halting problem which is, in turn, reducible to the halting problem for Turing machines [Hennie 77]. So, consider a deterministic, one-tape, one-head Turing machine T. T's computation on blank tape can be simulated by the following state machine TUR.

TUR consists of the following functions:

tape(i)	a non-derived V-function to store the contents of T's work tape. Initially, the tape is entirely blank.
state	a non-derived V-function to record T's state. Initially, state returns the initial state of T.
head	a non-derived V-function to record the position of T's head on its work tape.
move	an O-function to carry out one step in T's computation. A step in the computation consists of writing a symbol on the tape, moving the head left or right and then going to a new state.

All of these functions's applicability conditions consist of a constant, total function that returns the Boolean value true. Now recall that a computation of a Turing machine halts when it reaches a state and input symbol for which its next state function is undefined. The function in move's effects section shall be undefined for every such pair. It will be defined for every pair which continues T's computation. TUR simulates T by having a state that corresponds to each step in T's computation.

If T's computation when started on blank tape halts, then T will eventually reach a state and input symbol for which its next state function is undefined. So, in TUR's simulation of T, a state S of TUR will be reached that corresponds to this situation. Then, by construction, NEXT(S,move) is undefined so TUR is not well-defined. On the other hand, if TUR is not well-defined, the function in move's effects section must be undefined for some state S since the function in move's applicability condition is total. By construction,

this corresponds to T halting. Thus, TUR is well-defined if and only if T does not halt when started on blank tape.

3. A Language for State Machine Specifications

This chapter presents the syntax and semantics of a specification language for state machines called ALMS (A Language for Machine Specifications). Section 3.1 describes the syntax of ALMS and Section 3.2 discusses its semantics.

The discussion here is concrete, dealing with a specific language and its semantics. This chapter is a complement to the abstract discussion in Chapter 2. It shows how an actual language can be used to specify state machines and how its semantics can be defined using the model in Chapter 2 as a guide. The chapter concludes with an example discussing a proof that a particular machine is well-defined and consistent.

ALMS is similar in spirit and approach to SPECIAL [Roubine 76], SRI's specification language based on Parnas' approach. However, there are significant differences between the two languages. ALMS was developed soley to illustrate how to use the model in Chapter 2 to define the semantics of a state machine specficiation language. It is a simple language and does not have the features nor the expressive power that would be found in a specification language intended for use in the development of software systems. For example, when using ALMS to specify a symbol table for a block structured language, one can not define a V-function that returns the attributes associated with an identifier in the most local scope in which it occurs. This happens since ALMS contains no iteration or recursion constructs. ALMS can be extended to have these features but this would be beyond the intent of this chapter.

SPECIAL, however, was designed explicitly for specifying software systems. It is intended to be used in conjunction with a methodology for the design, implementation and

proof of computer systems [Roubine 76]. It naturally contains more features than ALMS. In SPECIAL, there are more constructs for defining the effects section of O-functions and the derivation section of derived V-functions. Furthermore, SPECIAL permits the definition of greater than unary operations on the data abstraction defined by the machine.

3.1 The Syntax of ALMS

An example of a state machine specification, described using ALMS, is given below in Figure 6. Here, the data abstraction defined is a symbol table for use in a block structured language. It has the following operations. *Add* is a O-function that places an identifier and its attributes into the symbol table at the current scoping level. We assume here that an identifier and its attributes are character strings and denote this type by *string*. The current scoping level is given by the non-derived V-function *level*. It can be incremented and decremented by the O-functions *inc_level* and *dec_level*, respectively. *Retrieve* is a derived V-function that returns the attributes of an identifier in a given level of the table and *present?* is another derived V-function that indicates whether or not an identifier has already been placed into a given scoping level of the table. The functions P_1 and P_2 used in these two derived V-functions's derivations are projection functions that return the first and second components, respectively, of an ordered pair. They simply permit one hidden V-function instead of two. Finally, *table_storage* is a hidden V-function used for storage purposes.

This specification illustrates the three major components of a state machine described using ALMS: the defining abstractions, the interface description and the definitions of the V-functions and O-functions. The interface description provides a very

Figure 6. Symbol Table

```
symbol_table = state machine is add, inc_level, dec_level, retrieve, present?, level

        level = non-derived V-function( ) returns integer
                Appl. Cond.: true
                Initial Value: 0
                end level

        table_storage = hidden V-function(a:integer,i:string) returns string × Booleans
                Appl. Cond.: true
                Initial Value: (don't care,false)
                end table_storage

        retrieve = derived V-function(a:integer,i:string) returns string
                Appl. Cond.: P2(table_storage(a,i))
                Derivation: retrieve(a,i) = P1(table_storage(a,i))
                end retrieve

        present? = derived V-function(a:integer,i:string) returns Boolean
                Appl. Cond.: true
                Derivation: present?(a,i) = P2(table_storage(a,i))
                end present?

        add = O-function(i,j:string)
                Appl. Cond.: ~P2(table_storage(level,i))
                Effects: 'table_storage'(level,i) = (j,true)
                end add

        inc_level = O-function( )
                Appl. Cond.: true
                Effects: 'level' = level + 1
                end inc_level

        dec_level = O-function( )
                Appl. Cond.: level > 0
                Effects: 'level' = level - 1
                end dec_level

end symbol_table
```

brief description of the V-functions and O-functions that users of the machine may employ. These functions, along with the hidden V-functions, are fully defined in the body of the machine. In these definitions, the defining abstractions are used. Here, they compose the domain and range of the V-functions and O-functions and, further, through their associated functions, help specify the meaning of the V-functions and O-functions.

3.1.1 The Defining Abstractions

As was discussed in Chapter 1, a state machine uses data abstractions that are distinct from the data abstraction defined by the machine. These abstractions are called the *defining abstractions*. They are assumed to be defined elsewhere.

In the remainder of this thesis, we shall use the integers, character strings and Booleans as defining abstractions and associate the usual operations with them. Furthermore, the set $\{\lambda\}$, where λ is the empty string, will be used as the domain of nullary V-functions and O-functions.

ALMS can, of course, have other defining abstractions besides these three. We will, however, leave the actual collection of defining abstractions unspecified and only assume that it at least contains the integers, character strings and Booleans.

Note also that the collection of defining abstractions can be augmented dynamically in the sense that once a data abstraction is specified in ALMS, such as *bounded_stack* in Chapter 1, it can be used as a defining abstraction in other specifications. So, the specification of a symbol table for a block structured language could use *bounded_stack* in its specification. We however chose not to do this for the symbol table in Figure 6.

3.1.2 The Interface Description

In ALMS, the *interface description* of a state machine provides a very brief description of the interface that the machine presents to the outside environment. It consists of the name of the data abstraction defined by the machine and a list of the functions that users of the machine may employ:

symbol_table = state machine is add, inc_level, dec_level, retrieve, present?, level

The list of functions contains the name of every non-derived V-function, derived V-function and O-function in the machine. The names of hidden V-functions may not appear in the interface description as they are not available outside the machine.

3.1.3 V-functions

This section specifies the syntax for the three types of V-functions of a state machine, the non-derived, hidden and derived V-functions. In the next section, the syntax of O-functions is given. Recall that non-derived V-functions are primitive aspects of the data abstraction defined by the machine. Hidden V-functions are used to represent aspects of the state that are not immediately observable and are inaccessible to users of the machine. However, limited access to them is provided by the derived V-functions, which are defined in terms of the non-derived and hidden V-functions.

Throughout this section and the next, it will be necessary to use *expressions*. An expression is formed through the composition of the non-derived and hidden V-functions of the machine and the functions associated with the defining abstractions. It may also contain elements of the defining abstractions and formal arguments. The formal arguments

serve as place holders in the expression.

We now turn to the definition of an expression. Our definitions will be written as though all expressions were written f(...) although we will use expressions like x+y with infixes such as + in examples.

Given a particular state machine SM we define an expression recursively as follows:

1) An element of a defining abstraction or a formal argument is an expression.

2) If $e_1, ..., e_n$ are expressions and f is a non-derived V-function of SM, a hidden V-function of SM or a function associated with one of the defining abstractions and f requires n arguments, then $f(e_1,...,e_n)$ is an expression.

We shall also refer to expressions by the type of value they return upon evaluation. For example, a Boolean expression evaluates to either true or false. Note that this definition excludes derived V-functions from appearing in an expression. This restriction is made to simplify the semantic definition in Section 3.2. In principle, there is no difficulty in allowing derived V-functions to appear in expressions.

3.1.3.1 Non-derived V-functions

The general schema for defining non-derived V-functions is given below in Figure 7.

The *mapping description* of a non-derived V-function specifies its name, domain and range, plus the fact that it is a non-derived V-function. Its syntax is

name = non-derived V-function( ) returns t_r

Figure 7. Syntax of a Non-derived V-function

name = non-derived V-function(x_1:t_1;...;x_n:t_n) returns t_r
Appl. Cond.: *Boolean expression*
Initial Value: init
end *name*

where t_r and t_i are names of defining abstractions and init $\in t_r$ U {undefined}

---

for nullary V-functions such as *level* in Figure 6 and

name = non-derived V-function(x_1:t_1;...;x_n:t_n) returns t_r

for n-ary non-derived V-functions such as *has* in Figure 15 of Chapter 4.

Here, t_r, the name of one of the defining abstractions, is the equivalent of R_v of Section 2.1.1. It is sometimes referred to as the *type* of the V-function. The x_i are the *formal arguments* of the V-function. They must be distinct. Also, t_i, again the name of a defining abstraction, is called the *type* of the formal argument x_i.

For a nullary non-derived V-function, D_v is {λ}. For an n-ary non-derived V-function v, D_v is $t_1 \times ... \times t_n$.

For example, consider the mapping description of *level* in Figure 6.

level = non-derived V-function( ) returns integer

Here, D_{level} = {λ} and R_{level} = integer and, in any state, the function associated with *level* is a member of [{λ} → integer].

The *applicability condition* of a non-derived V-function contains a Boolean expression that determines the success of a call to the function. This expression must be *type correct*. This means that whenever an object is used in the expression, its type must be

compatible with the type expected at that location. Furthermore, this expression must only contain formal arguments that appear in the V-function's mapping description.

The *initial value section* of a non-derived V-function specifies one element of R_v or contains the special symbol undefined. This restricts the mapping associated with the V-function in the initial state of the machine to be either a constant, total function or a totally undefined function. The latter case is specified by undefined.

3.1.3.2 Hidden V-functions

Hidden V-functions are specified in an analogous manner to non-derived V-functions. The only difference occurs in the syntax of the mapping description which contains the special symbol *hidden* instead of non-derived.

Figure 8. Syntax of a Hidden V-function

name = hidden V-function(x_1:t_1;...;x_n:t_n) returns t_r
Appl. Cond.: *Boolean expression*
Initial Value: init
end *name*

where t_r and t_i are the names of defining abstractions and init$\in t_r$ U (undefined)

3.1.3.3 Derived V-functions

The three sections in the definition of a derived V-function are defined as follows. The *mapping description* only differs from the mapping description of a non-derived or hidden V-function by use of the special symbol ***non-derived***. The *applicability condition* exactly follows the syntax of the applicability condition of a non-derived or hidden V-function. The *derivation section* is unique for this type of function.

Figure 9. Syntax of a Derived V-function

name = derived V-function(x_1:t_1;...;x_n:t_n) returns t_r
Appl. Cond.: *Boolean expression*
Derivation: defining clause
end *name*

where t_r and t_i are names of defining abstractions.

The *derivation section* of a derived V-function v contains a clause that defines v in terms of the other non-derived and hidden V-functions in the machine. Its syntax is described as follows.

If a derived V-function v has formal arguments x_1, ..., x_n and type t_r, then the derivation section of v is of the form

Derivation: $v(x_1,...,x_n) = e_1$

or

Derivation: if b then $v(x_1,...,x_n) = e_1$ else $v(x_1,...,x_n) = e_2$

Here, b is a boolean expression and e_1 and e_2 are expressions of type t_r. Again,

these expressions must be type correct and only use formal arguments of v.

3.1.4 O-functions

The general method of specifying an O-function is shown below in Figure 10.

Figure 10. Syntax of an O-function

name = O-function(x_1:t_1;...;x_n:t_n)
Appl. Cond.: *Boolean expression*
Effects: $equation_1$
.
.
.
$equation_m$
end *name*

where t_i is the name of a defining abstraction.

The *mapping description* specifies the domain of the O-function and identifies the particular function as an O-function. Its syntax is

name = O-function( )

for nullary O-functions such as *pop* in Figure 2 of Chapter 1 and

name = O-function(x_1:t_1;...;x_n:t_n)

for n-ary O-functions such as *add* in Figure 6. Here, t_i is the name of a defining abstraction and the x_i are the *formal arguments* of the O-function. They must be distinct. Also, t_i is the *type* of the formal argument x_i.

For a nullary O-function, D_o is {λ}. For an n-ary O-function o, D_o is $t_1 \times ... \times t_n$.

The range of the O-function is not specified by the mapping description since it is understood that the range of any O-function is the state set of the state machine.

The *applicability condition* of an O-function contains a Boolean expression. Naturally, this expression must be type correct and only contain formal arguments from the O-function's mapping description.

The *effects section* of an O-function contains a group of equations that reflect how the mappings associated with the non-derived and hidden V-functions are changed by an O-function call. There are two types of equations; simple equations and conditional equations. A *simple equation* in a state machine SM is defined as follows.

1) Let v be a nullary non-derived or hidden V-function of SM having type t and let e be an expression of type t. Then,

$$'v' = e$$

is a *simple equation.*

2) Let v be a n-ary (n>0) non-derived V-function or hidden V-function of SM having type t with formal arguments x_i of type t_i. Now let e be an expression of type t and e_i be expressions of type t_i. Then,

$$'v'(e_1,...,e_n) = e$$

is a *simple equation.*

The quotes are used to represent the result returned by the V-function after completion of the O-function call. An unquoted V-function denotes the value returned before the O-function call.

A conditional equation employs simple equations in its definition. Let eq_1 and eq_2 be simple equations and let b be a Boolean expression. Then,

if b **then** eq_1

and

if b **then** eq_1 **else** eq_2

are *conditional equations.* Note that this definition prohibits nested conditional equations and blocks of equations following the *then* or *else.* These restrictions were made only to simplify the semantic definition in Section 3.2. No problems would arise if the restrictions were lifted.

Finally, the effects section of an O-function contains a listing of conditional and simple equations. Its syntax is

Effects: eq_1
.
.
.
eq_m

The ordering is immaterial. Of course, all expressions in the effects section must be type correct and contain only formal arguments of the O-function.

3.2 The Semantics of ALMS

3.2.1 The State Set

As was previously mentioned in Chapter 2, a state of a state machine is completely specified when the mapping associated with each non-derived and hidden V-function of the machine is given. Hence, we view the state set, $\mathcal{S}$, of a state machine in the following manner:

$$\mathbb{S} \subset [D_{v_1} \rightarrow R_{v_1}] \times \ldots \times [D_{v_n} \rightarrow R_{v_n}] = \mathfrak{D}$$

where $\{v_1,\ldots,v_n\}$ is the set of non-derived and hidden V-functions. Note that D_{v_i} and R_{v_i} are defined in Sections 3.1.3.1 and 3.1.3.2.

Our purpose in this section is to define $\mathbb{S}$. Here, we shall use the same approach outlined in Section 2.2.1, taking the transitive closure of the initial state Q under the state transition function. So, to define $\mathbb{S}$, it suffices to define the initial state of the machine and then to describe the state change caused by an O-function call.

The *initial state* Q is the n-tuple $(\underline{init}_{v_1},\ldots,\underline{init}_{v_n})$ where $\{v_1,\ldots,v_n\}$ is the set of non-derived and hidden V-functions of the machine and

$$\underline{init}_{v_i} = \begin{cases} \phi & \text{if } v_i\text{'s initial value section contains the word } \underline{\text{undefined}} \\ \{(a,b) \mid a \in D_{v_i}\} & \text{if } v_i\text{'s initial value contains } b \in R_{v_i} \end{cases}$$

Here, ϕ is the null set. Note that functions are represented as sets of ordered pairs.

To define the next state function of a state machine, it is necessary to define, in general, how an O-function call maps one member of $\mathfrak{D}$ into another. This mapping is done by the O-function's effects section and we now turn to describing the meaning of this section.

The basic components of an O-function's effects section are the expressions that are used to build the simple equations and the conditional equations. These expressions are formed by composing the functions associated with the defining abstractions and the

non-derived and hidden V-functions of the machine. So, the first step in defining the next state mapping is to specify the meaning of these expressions. This will be done by defining a function μ that evaluates an expression. Then, using μ, it will be possible to describe the effect of a single equation. This will be done in the definition of a function $\mathbf{E}$ that specifies how an equation changes the mapping associated with a V-function. Finally, the total effect of the effects section will be specified by a function $\mathbf{TE}$ which, using $\mathbf{E}$, combines the effect of each equation in the effects section.

The meaning of an expression is dependent on two items. First, it depends on the particular O-function or V-function call since, in general, the expressions will contain formal arguments from the function's definition. Actual values must be substituted for these formal arguments. Second, the meaning of the expressions depends on the member $R \in \mathfrak{D}$ since R gives an interpretation to the non-derived and hidden V-functions. Note that the functions associated with the defining abstractions have a constant fixed interpretation and are independent of members of $\mathfrak{D}$.

So, let $R \in \mathfrak{D}$ and let ω be an O-function or V-function with expression E appearing in ω's definition. Finally, let $(a_1,\ldots,a_n) \in D_\omega$. Then to find the meaning of expression E, we can proceed as follows.

1) First, substitute a_i for every occurrence of its corresponding formal argument in E, obtaining E^*. Note, if $D_\omega = \{\lambda\}$, this step is unnecessary since ω has no formal arguments.

2) Now, to evaluate E^*, we shall view R as an interpretation or environment that specifies, for each symbol A, the value A_R of A in R. If A is an element of a defining

abstraction or one of their associated functions, then A_R is simply A. If A is a non-derived or hidden V-function, then A_R is the function associated with A in R. The value of $E^* = A(E_1,...,E_k)$ in R, following [Pratt 77], will be denoted by $R \vdash E^*$ and is defined by

$$R \vdash A(E_1,...,E_k) = A_R(R \vdash E_1,...,R \vdash E_k)$$

Since the non-derived and hidden V-functions may not be associated with total functions in R, it is possible that $R \vdash E^*$ is undefined.

Thus, as outlined above, we can define a semantic function $\mu(R,E,\omega,a)$ for $R \in \mathfrak{D}$, expressions E in ω's definition and $a \in D_\omega$ such that

$$\mu(R,E,\omega,a) = R \vdash E^*$$

We include the O-function or V-function name ω as a parameter to μ since it describes how to substitute the actual arguments for the formal arguments.

Now, let $R \in \mathfrak{D}$ and consider the simple equation

$$v(\alpha) = \beta$$

appearing in an O-function o's effects section. Then any call o(a) of o, where $a \in D_o$, would change v_R to the function

$$v_R^*(x) = \begin{cases} \mu(R,\beta,o,a) & \text{if } x = \mu(R,\alpha,o,a) \\ v_R(x) & \text{if } x \neq \mu(R,\alpha,o,a) \end{cases}$$

Here, a new value is returned for the argument $\mu(R,\alpha,o,a)$ and, otherwise, the old value is returned.

To help indicate such a function, we shall use the notation "b→x,y" developed in Chapter 2. Recall this notation has the value x if b is true and value y if b is false. So, for v_R^* above, we have

$$v_R^* = \lambda x.[(x=\mu(R,\alpha,o,a)) \rightarrow \mu(R,\beta,o,a), v_R(x)]$$

Using this notation, we define in Figure 11 an effects function

$$\mathbf{E}(R,o,a,Eq)$$

that specifies the change caused by an equation Eq on a V-function. $\mathbf{E}$ returns the new mapping associated with the V-function. It shows the effect of a single equation and not the entire effects section. So, in general, $\mathbf{E}$ can not be observed outside the machine.

The definition of $\mathbf{E}$ characterizes the expressive power of the effects section. If one wished to increase the expressive power of ALMS by adding constructs such as a *while* or *for all* statement, the definition of $\mathbf{E}$ would have to be extended. In fact, this is the only definition that would require modification. Both μ and $\mathbf{TE}$ would remain unchanged. This new definition of $\mathbf{E}$ could use the definition in Figure 11 as its basis. The effect of the new constructs could be defined in terms of the effects of their simpler parts in much the same manner as the effect of the *if-then-else* statement in Figure 11 is given in terms of the first two clauses of the definition.

To define the next state function, we must combine the effect of all the equations in the effects section. This can be done by calculating $\mathbf{E}(R,o,a,Eq)$ for every equation in the effects section and then combining these mappings into a new state.

Figure 11. Effects Function

Definition

Given a state machine specification SM and $R \in \mathfrak{D}$,
let o be an O-function of SM with Eq appearing in o's effects section and $a \in D_o$.
Then $\mathbf{E}(R,o,a,Eq)$ is defined as follows;

i) If Eq is a simple equation of the form *'v' = e* where v is a parameter-less V-function,

$$\mathbf{E}(R,o,a,Eq) = \{(\lambda, \rho_s(R,e,o,a))\}$$

ii) If Eq is a simple equation of the form *'v'(w) = e*, then

$$\mathbf{E}(R,o,a,Eq) = \lambda x.[(x = \rho_s(R,w,o,a)) \rightarrow \rho_s(R,e,o,a), v_R(x)]$$

iii) If Eq is a conditional equation of the form *if c then s* where *s* is *'v' = e* or *'v'(w) = e*,

$$\mathbf{E}(R,o,a,Eq) = \begin{cases} \mathbf{E}(R,o,a,s) & \text{if } \rho_s(R,c,o,a) = \text{true} \\ v_R & \text{if } \rho_s(R,c,o,a) = \text{false} \end{cases}$$

iv) If Eq is a conditional equation of the form *if c then s_1 else s_2* then

$$\mathbf{E}(R,o,a,Eq) = \begin{cases} \mathbf{E}(R,o,a,s_1) & \text{if } \rho_s(R,c,o,a) = \text{true} \\ \mathbf{E}(R,o,a,s_2) & \text{if } \rho_s(R,c,o,a) = \text{false} \end{cases}$$

First, define the function

$$U_n((a_1,\ldots,a_n),i,c) = \begin{cases} (a_1,\ldots,a_{i-1},c,a_{i+1},\ldots,a_n) & \text{if } 1 \le i \le n \\ (a_1,\ldots,a_n) & \text{if } i \le 0 \text{ or } i > n \end{cases}$$

where i is an integer and $(a_1,\ldots,a_n)$ is an n-tuple. This function changes the *i*th component

of the n-tuple to c.

Now, let o be an O-function with equations $Eq_1,...,Eq_m$ in its effects section and let $a \in D_o$. Furthermore, assume

$$R \in \mathfrak{D} = [D_{v_1} \rightarrow R_{v_1}] \times ... \times [D_{v_n} \rightarrow R_{v_n}].$$

Finally, let $f_i = \mathbb{E}(R,o,a,Eq_i)$ and let

$$j_i = \begin{cases} k & \text{if } \mathbb{E}(R,o,a,Eq_i) \text{ changes V-function } v_k\text{'s mapping} \\ n+1 & \text{if } \mathbb{E}(R,o,a,Eq_i) \text{ doesn't change any V-function's mapping} \end{cases}$$

Then the total effect of the effects section is given by

$$T\mathbb{E}(R,o,a,Eq_1,...,Eq_m) = U_n(...U_n(U_n(R,j_1,f_1),j_2,f_2)...,j_m,f_m)$$

Note that a V-function not effected by an equation Eq_i retains the previous mapping asociated with it. $T\mathbb{E}(R,o,a,Eq_1,...,Eq_m)$ corresponds to $\mathbb{E}_o(R,a)$ in Chapter 2. So, we can define the next state function as follows.

Definition
Let o be an O-function with Boolean expression $\hat{b}$ in its applicability condition and equations $Eq_1,...,Eq_m$ in its effects section.
Let $a \in D_o$ and $R \in \mathfrak{D}$.
Then,

$$NEXT(R,o,a) = \begin{cases} T\mathbb{E}(R,o,a,Eq_1,...,Eq_m) & \text{if } p_{\hat{b}}(R,\hat{b},o,a)\text{=true} \\ R & \text{if } p_{\hat{b}}(R,\hat{b},o,a)\text{=false} \end{cases}$$

So, the state set can be generated as in Chapter 2.

1) $Q \in \mathcal{S}$.

2) If $R \in \mathcal{S}$ and σ is an O-function, then if $NEXT(R,\sigma,a)$ is defined, $NEXT(R,\sigma,a) \in \mathcal{S}$ where $a \in D_\sigma$.

3) These are the only elements of $\mathcal{S}$.

Again, we must consider the question of whether or not NEXT is well-defined. This is dependent on μ and $\mathcal{TE}$. Recall that μ is not necessarily a total function. So, it is possible for some state S and $x \in D_\sigma$ that $\mu(S,R,\sigma,x)$ is undefined. Also, $\mathcal{TE}$ is not necessarily total so we can encounter a similar situation. These two cases correspond to the problem, discussed in Chapter 2, when $\mathcal{N}_\sigma(S,x)$ and $\mathcal{Z}_\sigma(S,x)$ are undefined.

Besides the totality of $\mathcal{TE}$, there is a further problem that we must consider. We have defined the ordering of the equations $Eq_1,...,Eq_m$ as immaterial. However, it is possible for some state S and $a \in D_\sigma$ that $\mathcal{TE}(S,\sigma,a,Eq_1,...,Eq_m) \neq \mathcal{TE}(S,\sigma,a,Eq_{\pi(1)},...,Eq_{\pi(m)})$ where π is a permutation from $(1,...,m)$ onto $(1,...,m)$. In this case, $\mathcal{TE}$ would be nondeterministic or not uniquely defined in the sense that its value depends on the choice of the order of the equations $Eq_1,...,Eq_m$.

To handle these situations, we must introduce the notion of a well-defined state machine. Due to the last case, the definition differs slightly from that of Chapter 2 since we must explicitly guarantee that $\mathcal{TE}$ is uniquely defined whereas in Chapter 2 this was unnecessary since by definition $\mathcal{T}_\sigma$ was a function.

Definition
A state machine SM is *well-defined* if for any $S \in \mathbb{S}$, O-function o and $a \in D_o$
both 1) NEXT(S,o,a) is defined
and 2) $\mathbb{TE}(S,o,a,Eq_1,...,Eq_m) = \mathbb{TE}(S,o,a,Eq_{\pi(1)},...,Eq_{\pi(m)})$
where equations $Eq_1,...,Eq_m$ appear in o's effects section
and π is any permutation from (1,...,m) onto (1,..,m).

3.2.2 The Semantics of V-functions and O-functions

With this definition of the state set $\mathbb{S}$ of a state machine specification, it is now possible to formally define the meaning of the O-functions and V-functions. As in Chapter 2, this will be done by defining mappings V-Eval for V-functions and O-Eval for O-functions.

O-Eval will be defined first. Now, given a state S and an O-function o with Boolean expression $\mathbb{b}$ in its applicability condition, O-Eval returns a function from D_o into $\mathbb{S} \cup \{\underline{error}\}$. So, using lambda notation,

$$O\text{-}Eval(S,o) = \lambda a.[\mathbb{ps}(S,\mathbb{b},o,a) \rightarrow NEXT(S,o,a),\underline{error}]$$

Again O-Eval(S,o) is not necessarily total but in a well-defined state machine this is always the case.

For any V-function v and state S, V-Eval will return a function from D_v into $R_v \cup \{\underline{error}\}$. First, for a non-derived or hidden V-function v and a state S, recall that v_S denotes the function associated with v in state S. Then for any non-derived or hidden V-function v with expression $\mathbb{b}$ in its applicability condition,

$$V\text{-}Eval(S,v) = \lambda a.[\mathbb{ps}(S,\mathbb{b},v,a) \rightarrow v_S(a),\underline{error}]$$

Finally, for a derived V-function v with expression $\hat{b}$ in its applicability condition, there are two cases.

i) If v's derivation section contains $v(x_1,...,x_n) = e$, then

$$V\text{-}Eval(S,v) = \lambda a.[\rho_B(S,\hat{b},v,a) \rightarrow \rho_B(S,e,v,a), \underline{error}]$$

ii) If v's derivation section contains *if c then* $v(x_1,...,x_n) = e_1$ *else* $v(x_1,...,x_n) = e_2$, then

$$V\text{-}Eval(S,v) = \lambda a.[\rho_B(S,\hat{b},v,a) \rightarrow [\rho_B(S,c,v,a) \rightarrow \rho_B(S,e_1,v,a), \rho_B(S,e_2,v,a)], \underline{error}]$$

As was mentioned in Chapter 2, V-Eval(S,v) is not necessarily a total function from D_V into $R_v \cup \{\underline{error}\}$. When this is not the case, we say the state machine is consistent. The definition is the same as in Chapter 2.

3.3 An Example

In this section, we outline a proof that a particular state machine is well-defined and consistent. The full details of the proof are contained in Appendix 2. Our example specification is illustrated in Figure 12. This data abstraction is a queue with three operations; *insert* which adds an integer to the rear of the queue; *delete* which removes the integer at the front of the queue and *first_element* which returns the integer at the front of the queue. The hidden V-function *storage* is used to store the elements of the queue. *Front* and *back* point, respectively, to the beginning and end of the queue. Note that this queue can hold an arbitrary number of integers.

Figure 12. Queue

```
queue = state machine is insert, delete, first_element

    first_element = derived V-function( ) returns integer
                    Appl. Cond.: ~(front = back - 1)
                    Derivation: first_element = storage(front)
                    end first_element

    front = hidden V-function( ) returns integer
                    Appl. Cond.: true
                    Initial Value: -1
                    end front

    back = hidden V-function( ) returns integer
                    Appl. Cond.: true
                    Initial Value: 0
                    end back

    storage = hidden V-function(i:integer) returns integer
                    Appl. Cond.: front ≤ i ≤ back
                    Initial Value: undefined
                    end storage

    insert = O-function(i:integer)
                    Appl. Cond.: true
                    Effects: 'storage'(back - 1) = i
                    'back' = back - 1
                    end insert

    delete = O-function( )
                    Appl. Cond.: ~(front = back - 1)
                    Effects: 'front' = front - 1
                    end delete

end queue
```

We shall first show that the specification is well-defined. This will be done by initially proving a lemma that captures the key properties necessary to insure that the machine is well-defined. Informally, the lemma states that *front* and *back* always return an integer value. With the aid of this lemma, we will directly establish that the specification is well-defined.

Lemma For any $S \in \mathbb{S}$, $back_S \in [\{\lambda\} \rightarrow integer]$ and $front_S \in [\{\lambda\} \rightarrow integer]$ and $back_S \neq \phi \neq front_S$ where ϕ is the null set.

This lemma can be established using the inductive method outlined in Section 2.2.4. The basis of the induction trivially follows since *first* and *last* are defined to return -1 and 0, respectively, in the initial state of the machine. The inductive step is also readily apparent. For any state, *insert* decrements *back* by 1 and leaves *front* unchanged. Furthermore, *delete* leaves *back* unchanged and only decrements *front* by 1 if its applicability condition is satisfied.

We can now prove that the machine is well-defined using the above lemma. This lemma is helpful because both *front* and *back* must be evaluated in *insert's* and *delete's* applicability condition and effects section. The lemma guarantees that this evaluation can be done.

To prove that the machine is well-defined, three properties must be established i) the applicability conditions of the O-functions *insert* and *delete* are defined; ii) the next state function is defined for both *insert* and *delete*; and, finally, iii) the ordering of the equations in both *insert's* and *delete's* effects sections is immaterial. Note that iii) is trivially established since *delete* has only one equation in its effects section and the two equations in

insert's effects section modify different V-functions. Thus, it is only necessary to deal with i) and ii). We now complete the proof.

Since *insert's* applicability condition is a constant and *delete's* applicability condition only involves *front* and *back*, which were shown by the lemma always to return an integer value, i) is established. The second part of the proof is also established by appealing to the lemma. Since *insert's* and *delete's* effects sections only evaluate *front* and *back*, it is clear that the next state function is defined for both these O-functions.

We will now show that the specification is consistent. This involves proving that the four V-functions are total. First note that the lemma guarantees that *front* and *back* are total. *Storage* and *first_element*, however, require more attention. Again, we must introduce a lemma and then prove the desired results directly from the lemma. The lemma shows that *storage's* applicability condition accurately describes its domain.

Lemma

For any $S \in \mathfrak{S}$, if $front_S \geq k \geq back_S$, then $storage_S(k)$ is defined.

This lemma can also be established by the inductive approach outlined in Section 2.2.4. The basis is vacuously true since, in the initial state, *back* is greater than *front*. Now assume the lemma is true for any state S. We must consider the effect of *insert* and *delete* on S. Since *delete* decreases *front* by 1, the result immediately follows from the inductive hypothesis. Now let $S^* = NEXT(S,insert,x)$. There are two cases. Either $front_{S^*} = back_{S^*}$, in which case $storage_{S^*}(front_{S^*})$ evaluates to x, or $front_{S^*} \neq back_{S^*}$. In the latter case, $front_{S^*} > back_{S^*}$ and $front_{S^*} = front_S$ and $back_{S^*} = back_S - 1$. So for $front_{S^*} \geq k \geq back_{S^*} + 1$, $storage_{S^*}(k)$ is defined by the inductive hypothesis. Also,

$storage_S(back_S)$ evaluates to x.

This lemma immediately establishes that V-Eval(S,storage) is total in any state S. To see that V-Eval(S,first_element) is total in any state S, note that there are two cases. First, *first_element's* applicability condition can evaluate to false in which case the special symbol error is returned. Otherwise, its applicability condition evaluates to true and $front_S \geq back_S$. So, by the lemma, $storage_S(front_S)$ is defined and the proof is complete.

4. An Implementation Language for State Machines

Chapters 2 and 3 have focused on formalizing the semantics of state machine specifications. The work accomplished in these two chapters allows one to write precise and unambiguous specifications of data abstractions using state machines. But these abstractions are only mathematical objects. They can not be used directly as objects in any programming language. They must first be implemented. Thus, an important aspect in any formalization is to be able to describe formally when a data abstraction, specified by a state machine, is properly implemented in some programming language. Developing this definition involves the following. First, a programming language for implementing state machines must be described. This topic is discussed in this chapter. Then, a method of proving the correctness of an implementation must be fixed. This topic is treated in the next chapter.

In this chapter, the general properties of any programming language for implementing state machine specifications are described; in particular, the basic data abstractions to represent the specified objects and the control constructs to implement the V-functions and O-functions. This approach of illustrating program correctness is valid since any programming language for implementing state machines must include these features. The actual implementation of these data abstractions and control constructs is unimportant here. Accordingly, this detail is totally suppressed in this chapter. Rather control constructs to be used with state machine specifications are introduced. So, implementations of state machine specifications will be written in terms of other, simpler state machine specifications. For instance, a specification of a stack could be implemented using state machine specifications of variables and arrays to represent elements of the data

abstraction and the control constructs to realize the V-functions and O-functions.

To develop a definition of program correctness, it is only necessary to define the relation between the objects of the specification and the objects of the implementation. Hence, since this chapter contains a general discussion of the objects of an implementation, it is possible in Chapter 5 to give a general definition of program correctness. To prove that this definition holds requires involvement with the semantics of the programming language and identifying correspondences between objects of the language and terms used in the definition. But these issues are not a major concern for only stating the definition of a correct program that involves state machines.

4.1 An Example

An example state machine specification and its corresponding implementation are given in Figures 13 and 14, respectively. The data abstraction specified in Figure 13 is a finite integer set. *Insert* and *remove* are O-functions that insert and remove, respectively, integers from the set. *Cardinality* is a V-function that returns the number of integers in the set. *Has* is another V-function that determines whether or not a given integer is in the set.

Figure 14 contains an implementation of finite_integer_set. The set is stored as an ordered sequence of integers in the array A. *INSERT*, *REMOVE*, *CARDINALITY* and *HAS* are the corresponding implementations of *insert*, *remove*, *cardinality* and *has*.[1] Each of these operations uses *SEARCH*, which performs a binary search on the array A. *SEARCH*

---

1. Throughout this thesis, lower case letters will be used in the names of V-functions and O-functions of a state machine specification. Capital letters will represent their corresponding implementation.

Figure 13. Specification of Finite Integer Set

```
finite_integer_set = state machine is cardinality, has, remove, insert

        cardinality = non-derived V-function( ) returns integer
                Appl. Cond.: true
                Initial Value: 0
                end cardinality

        has = non-derived V-function(i:integer) returns Boolean
                Appl. Cond.: true
                Initial Value: false
                end has

        insert = O-function(i:integer)
                Appl. Cond.: cardinality<100
                Effects: 'has'(i) = true
                if ~has(i) then 'cardinality' = cardinality + 1
                end insert

        remove = O-function(i:integer)
                Appl. Cond.: true
                Effects: 'has'(i) = false
                if has(i) then 'cardinality' = cardinality - 1
                end remove

end finite_integer_set
```

Figure 14. Implementation of finite integer set

```
FINITE_INTEGER_SET = implementation is INSERT, REMOVE, HAS, CARDINALITY


A: array of integers initially undefined
COUNT: integer variable initially 0


SEARCH = procedure(a,f,k:integer) returns integer
        if f=k
                then return k
                else if a≤A.read(⌊(f+k)/2⌋)
                        then return SEARCH(a,f,⌊(f+k)/2⌋)
                        else return SEARCH(a,⌊(f+k)/2⌋+1,k)
        end SEARCH


INSERT = procedure(i:integer)
        if COUNT.read=0
            then begin
            A.change(0,i);
            COUNT.change(1)
            end
            else if COUNT.read<100
                 then
                 if COUNT.read = SEARCH(i,0,COUNT.read)
                      then begin
                      A.change(SEARCH(i,0,COUNT.read),i);
                      COUNT.change(COUNT.read+1)
                      end
                      else if A.read(SEARCH(i,0,COUNT.read))=i
                           then return
                           else begin
                           for j:=COUNT.read step -1 until SEARCH(i,0,COUNT.read) do
                                        if j>1 then A.change(j,A.read(j-1));
                            A.change(SEARCH(i,0,COUNT.read),i);
                            COUNT.change(COUNT.read+1);
                            end
                 else signal error
        end INSERT
```

```
REMOVE = procedure(i:integer)
        if COUNT.read=0
                then return
                else
                if A.read(SEARCH(i,0,COUNT.read))=i
                        then begin
                        for j := SEARCH(i,0,COUNT.read) until COUNT.read-2 do
                                A.change(j,A.read(j+1));
                        COUNT.change(COUNT.read-1)
                        end
                        else return
        end REMOVE

CARDINALITY = procedure( ) returns integer
        return COUNT.read
        end CARDINALITY

HAS  = procedure(i:integer) returns Boolean
        if COUNT.read=0
                then return false
                else if A.read(SEARCH(i,0,COUNT.read))=i
                        then return true
                        else return false
        end HAS

end FINITE_INTEGER_SET
```

---

returns the index where the binary search stops.

An implementation consists of three parts: an interface description, an object description and operation definitions.

The *interface description* of an implementation provides a very brief description of the interface that the implementation presents to the outside environment. It consists of the name of the data abstraction being implemented and a list of the operations that users of the implementation may employ.

FINITE_INTEGER_SET - implementation is INSERT, REMOVE, HAS, CARDINALITY

Operations such as SEARCH whose names do not appear in the interface description should not be accessible by users of the implementation.

The *object description* consists of the declaration of data abstractions that are used to represent the object being implemented. These data abstractions are used as the concrete representation of the specified abstraction. Here, each of these data abstractions will be specified as a state machine and ALMS will be used for this purpose although any specification language could be used.

In the example, the object description consists of

A: array of integers initially undefined
COUNT: integer variable initially 0

These phrases are syntactic sugar for the state machine specifications of a variable and an array given in Figures 15 and 16, respectively. COUNT and the array A are used to represent the data abstraction. COUNT reflects the number of integers in the set. These integers are stored as an ordered sequence in the array A from A[1] to A[COUNT-1].

The body of the implementation consists of operation definitions. These definitions provide implementations of the permissible operations on the data abstraction; the O-functions and the non-derived and derived V-functions. It is not necessary to implement the hidden V-functions since they are unknown to users. An operation definition should be given for every operation that appears in the interface description.

In our examples, operation definitions will be written using V-functions and O-functions grouped together by the usual control constructs that would be found in, say, ALGOL 60 or PASCAL. These V-function and O-function calls should be interpreted as

Figure 15. Variable

X: type_t variable initially a is equivalent to

```
X = state machine is X.read, X.change

    X.read = non-derived V-function( ) returns type_t
             Appl. Cond.: true
             Initial Value: a
             end X.read

    X.change = O-function(I:type_t)
             Appl. Cond.: true
             Effects: 'X.read' = I
             end X.change

    end X
```

where type_t is the name of a defining abstraction of ALMS
and a is an element of type_t or undefined.

---

follows. Assume that the implementation maintains a record of the current state of the machines in the object description. For example, if no O-functions have been called, the implementation would view each state machine as being in its initial state. Now, each V-function call v(a) should be interpreted as

$$V\text{-}Eval(S,v)(a)$$

where S, remembered by the implementation, is the current state of the V-function's machine. An O-function call o(a) is interpreted as

$$O\text{-}Eval(S,o)(a) = S^*$$

Figure 16. Array

X: type_t array initially a is equivalent to

```
X = state machine is X.read, X.change

    X.read = non-derived V-function(i:integer) returns type_t
             Appl. Cond.: true
             Initial Value: a
             end X.read

    X.change = O-function(j:integer,i:type_t)
             Appl. Cond.: true
             Effects: 'X.read'(j) = i
             end X.change

    end X
```

where type_t is the name of a defining abstraction of ALMS
and a is an element of type_t or undefined.

---

where S is as before. Furthermore, the implementation now updates S to S^* and maintains S^* as the current state of o's machine until another O-function of that machine is called.

5. Proving an Implementation Correct

To formally establish the correctness of a program, one must prove that the program is equivalent to a specification of its intended behavior by formal, analytic means. This chapter is concerned with this process, discussing how to prove the correctness of programs that implement data abstractions specified by state machines.

Here, the homomorphism property will be used in the proofs. In general, this involves showing the following [Hoare 72a]. Assume there is a class of abstract objects $\mathbb{A}$ with abstract operations. Furthermore, suppose that x^* is the concrete object representing an abstract object belonging to $\mathbb{A}$. Let $\mathbb{C}$ be the collection of all such x^*. Finally, suppose that ω_c is a concrete operation that purports to be an implementation of an abstract operation ω_a. Then, the homomorphism property involves defining an abstraction function, A, mapping from $\mathbb{C}$ onto $\mathbb{A}$ and showing for every operation that

$$\omega_a(A(x^*)) = A(\omega_c(x^*)).$$

Before attempting such a proof, three steps must be performed. First, the concrete objects used to represent the elements of a data abstraction must be characterized. This is discussed in Section 5.1. Then the class of abstract objects $\mathbb{A}$ must be identified. This is done in Section 5.2. Finally, the abstraction function must be described. Section 5.3 is concerned with this issue and the problem of adapting the homomorphism property to the particular needs of state machine specifications.

5.1 The Concrete Representation

A concrete implementation of a data abstraction specified by a state machine will usually consist of a collection of objects to represent the state set and a group of operations that purport to implement the various functions of the machine. Some of these operations will implement O-functions and others will implement derived and non-derived V-functions. Note that it is unnecessary to implement hidden V-functions since they are inaccessible and not an intrinsic part of the data abstraction.

All of these operations will access or modify the concrete objects that are used to represent the state set of the state machine. We shall denote the set of these concrete objects by $\mathbf{C}$. If a concrete operation implements an O-function ω, then we view the operation, denoted ω_C, as a mapping from $\mathbf{C} \times D_\omega$ into $\mathbf{C}$. If it implements a V-function ω, then it is a mapping from $\mathbf{C} \times D_\omega$ into R_ω. By adopting this view, we are making $\mathbf{C}$ an explicit parameter of each operation. This may differ from the actual syntax of the implementation language but clarifies the operation of procedures that operate through side effects or by accessing global variables. For example, in the *finite integer set* implementation in Chapter 5, *INSERT* would be viewed as mapping from $\mathbf{C}$ × integer into $\mathbf{C}$. Here, $\mathbf{C}$ corresponds to the states of A and COUNT remembered by the implementation.

We shall now describe $\mathbf{C}$ in more detail. In general, $\mathbf{C}$ is a subset of a large collection of objects. For example, in the *finite integer set* example of Chapter 4,

$$\mathbf{C} \subset \mathbb{S}_A \times \mathbb{S}_{COUNT}$$

A set such as $\mathbb{S}_A \times \mathbb{S}_{COUNT}$ is too large to use as the domain of the abstraction function since it usually contains elements that do not correspond to any element of the data

abstraction being implemented. So, it is necessary to describe $\mathbf{C}$ explicitly.

The standard way to do this is to use a *concrete invariant*, I_c. This is a predicate defining some relationship between the concrete variables and thus placing a constraint on the possible combinations of values that they may take. Then,

$$\mathbf{C} = \{x \mid I_c(x)\}.$$

For the *finite integer set* implementation, I_c is

$$0 \leq \text{COUNT.read} \leq 100 \ \wedge \ (\forall i\ j)[0 \leq i < j < \text{COUNT.read} \rightarrow \text{A.read}(i) < \text{A.read}(j)]$$

This predicate states that the implementation of *finite_integer_set* contains at most 100 integers and that the elements between 0 and COUNT in the array A are all distinct and ordered. This latter condition is necessary to insure the correctness of *SEARCH*. The ordered pair

$$(\phi,((\lambda,0))) \in \mathbf{S}_A \times \mathbf{S}_{\text{COUNT}}$$

satisfies I_c above. This ordered pair corresponds to both machines A and COUNT being in their initial states.

5.2 The Abstract Objects

The elements of the concrete representation $\mathbf{C}$ should implement or represent the entire state set of a state machine specification. However, a concrete object need not represent a single state but rather a set of states. This occurs because certain states may have no observable differences. When this happens, we say the states are *equivalent*. So, a concrete object actually implements the equivalence class of a state and we identify the class of abstract objects $\mathbf{A}$ with the set of equivalence classes of the state set.

For example, consider the specification of *bounded_stack* in Chapter 1. Its state set is

a subset of $[D_{stack} \rightarrow R_{stack}] \times [D_{depth} \rightarrow R_{depth}]$. Now consider the two states

$$S_1 = (\phi,((\lambda,0)))$$

and

$$S_2 = (((1,1)),((\lambda,0)))$$

Here, ϕ is the null set. The first state S_1 corresponds to Q_0, the initial state of *bounded_stack*, so *stack* is totally undefined and *depth* returns 0. (Recall our previous convention that the domain of nullary functions is $\{\lambda\}$.) The second state S_2 corresponds to stack(1) returning the value 1 and for $x \neq 1$, stack(x) is undefined. Also in S_2, *depth* returns the value 0. Thus S_2 = NEXT(NEXT(Q_0,push,1),pop,λ) where Q_0 is the initial state of *bounded_stack*. These two states, S_1 and S_2, are equivalent as far as a user of the machine is concerned since *stack* is hidden from a user and *depth* returns 0 in either state.

Equivalent states are defined below. Intuitively, two states are equivalent when it is impossible for a user of the specification to determine any difference between them.

Definition
Two states S_1 and S_2 of a state machine specification SM are *equivalent* if for any

$$S_1^* = \text{O-Eval}(\ldots\text{O-Eval}(\text{O-Eval}(S_1, o_1)(a_1)),o_2)(a_2)),\ldots,)o_n(a_n))$$

$$S_2^* = \text{O-Eval}(\ldots\text{O-Eval}(\text{O-Eval}(S_2, o_1)(a_1)),o_2)(a_2)),\ldots,)o_n(a_n))$$

where o_i is an O-function of SM, $a_i \in D_{o_i}$ and $n \geq 0$

either

$$S_1^* = S_2^* = \underline{\text{error}}$$

or

both S_1^* and S_2^* are undefined

or

$$\text{V-Eval}(S_1^*,v) = \text{V-Eval}(S_2^*,v)$$

for any non-derived or derived V-function v of SM.

This definition guarantees that if a series of O-functions are applied to two equivalent states, then two new states are obtained where the non-derived and derived V-functions

behave identically. Furthermore by applying a series of O-functions to the two states, we make certain that all delayed effects become apparent. We shall denote the equivalence class of a state S by [S].

For example, for a state S of *finite_integer_set* in Chapter 4, [S] simply contains the state S. For the initial state Q of *bounded_stack*,

$$[Q] = \{(F,\{(\lambda,0)\}) \in \mathfrak{S}_{\text{bounded\_stack}}\}$$

where F is a mapping associated with the hidden V-function *stack*. In other words, [Q] is the set of all states where *depth* returns the value 0. Note that [Q] is infinite. This occurs since the data abstraction *integers* used in *bounded_stack* contains infinitely many elements. If *bounded_stack* used a data abstraction for the integers that had a bound on the number of elements (such as the integers used in programming languages), then [Q] and all the other equivalence classes of *bounded_stack* would be finite. Furthermore, *bounded_stack's* state set would be finite.

The equivalence classes of the state set can be enumerated by using a *normal form generation* of a state as the representative of each equivalence class. A normal form generation of a state is either Q , the initial state, or generated from Q by only using information adding O-functions. Recall that an information removing O-function deletes information that was previously added by an information adding O-function. The same effect can be achieved by initially not adding this information. Thus, this representation is valid since every state either equals a normal form generation or is equivalent to a normal form generation. For example, in *finite_integer_set*,

$$\text{NEXT}(Q,\text{insert},1) = S$$

is a normal form generation but

NEXT(NEXT(NEXT(Q,insert,1),remove,1),insert,1)

is not. However, it is equivalent to S.

5.3 The Homomorphism Property

It is now possible to state what the correctness of an implementation means. Informally, the implementation of a data abstraction is correct when the operations of the implementation and of its corresponding specification behave identically and there is at least one object of the implementation corresponding to every object of the data abstraction. This is the usual meaning of a homomorphism in mathematics [Fraleigh 67].

Formally, to prove the correctness of an implementation, one must first define an abstraction function A from $\mathbf{C}$ onto the equivalence classes of $\mathbf{S}$, the state set of the state machine being implemented. A simple and natural way to do this is to first define a function f from $\mathbf{C}$ into $\mathbf{S}$ (e.g. into the normal forms in $\mathbf{S}$) and then define the abstraction function as $A(x) = [\![f(x)]\!]$. A must map *onto* the abstract objects $\mathbf{R}$, the equivalence classes of $\mathbf{S}$. This property guarantees that every state is implemented. However, A need not be a one-to-one mapping from $\mathbf{C}$ onto the equivalence classes of $\mathbf{S}$. So, many concrete objects can represent one abstract object.

Now, after defining A, one must show for every $C \in \mathbf{C}$ and O-function ω,

$$[\![O\text{-}Eval(f(C),\omega)(x)]\!] = [\![f(\omega_c(C,x))]\!]^1$$

where $x \in D_\omega$ and for every non-derived or derived V-function ω

$$V\text{-}Eval(f(C),\omega)(x) = \omega_c(C,x)$$

---

1. This is a slight abuse of notation. We assume [[error]] = error.

where $x \in D_\omega$.

The above definition assumes that an implementation of a V-function does not modify the concrete representation. If it does, we must add the condition

$$[\![f(C)]\!] = [\![\omega_c(C,x)]\!].$$

This could occur in an implementation of *finite_integer_set* where *HAS* reorders the elements of A.

We shall illustrate these ideas using the example in Chapter 5. Again we assume

$$\mathbb{C} \subset \mathbb{S}_A \times \mathbb{S}_{COUNT}$$

and

$$\mathbb{S}_{integer\_set} \subset [D_{has} \to R_{has}] \times [D_{cardinality} \to R_{cardinality}].$$

First, let $(C_1,C_2) \in \mathbb{C}$. So C_1 is a state of the array A and C_2 is a state of the variable COUNT. It is helpful to define the predicate $IN(C_1,C_2,i)$, which is true if the integer i is a member of the concrete set, as follows:

$$IN(C_1,C_2,i) \equiv (\exists j)[V\text{-}Eval(C_1,A.read)(j) = i \wedge 0 \le j < V\text{-}Eval(C_2,COUNT.read)].$$

Informally, this predicate is true if there exists an integer j such that in the state C_1 of A, A.read(j) returns i and j is greater than or equal to zero and less than the value returned by COUNT.read in state C_2 of COUNT. Then

$$f(<C_1,C_2>) =$$

$$(\{(i,true) \mid IN(C_1,C_2,i)\} \cup \{(i,false) \mid \sim IN(C_1,C_2,i)\} \ , \{(\lambda,V\text{-}Eval(C_2,COUNT.read))\})$$

Now to establish the correctness of the implementation it is necessary to show that A is onto $\mathbb{A}$. This can be established by showing for every $S \in \mathbb{S}$ that there exists a $C \in \mathbb{C}$ such that $[\![f(C)]\!] = [\![S]\!]$. Then one must prove that

1. $[\![O\text{-}Eval(f(C),insert)(x)]\!] = [\![f(INSERT(C,x)]\!]$

2. [O-Eval(f(C),remove)] = [f(REMOVE(C)]

3. V-Eval(f(C),has)(x) = HAS(C,x)

4. V-Eval(f(C),cardinality) = CARDINALITY(C,x)

Consider proving 3. Here, it is necessary to show that if an integer x is or is not in the set, then *HAS*, respectively, returns true or false. This property could be shown by first proving a lemma stating that SEARCH(x,0,COUNT,real) always returns the index where x should appear in the array A. This lemma would also be useful in establishing 1. and 2. Furthermore, in proving 1. and 2., it would be necessary to show that both preserve the concrete invariant since *A*'s domain is $\mathbb{C}$.

6. An Extended Model for State Machines

The model of a state machine developed in Chapter 2 does not allow the specification of V-functions or O-functions that operate on two or more elements of the data abstraction defined by the machine. In this chapter, this restriction is lifted and the model is extended to allow the specification of these greater than unary operations.

To specify greater than unary operations, the definitions of the O-functions and V-functions must first be extended. O-functions and derived V-functions will now be allowed to have more than one argument of the data abstraction specified by the machine. For example, this allows the definition of an O-function, *union*, which computes the union of two sets, or the definition of a derived V-function, *common_element?*, which returns **true** or **false** if two sets have or do not have, respectively, any common elements.

O-functions will still retain their interpretation of changing the state of the machine but now this state change can be dependent on more than one state. Derived V-functions will also have their previous interpretation expanded. Now, instead of allowing the user limited access to only one state, they will permit simultaneous access to more than one state.

Non-derived V-functions and hidden V-functions will, however, still be restricted to their previous interpretation. So, they can only specify unary operations on the data abstraction specified by the machine. This conforms to their interpretation as fully characterizing a single state of the machine.

An example of a state machine specification with greater than unary operations is given in Figure 17. This is the specification of an *integer set* that can contain an arbitrary number of integers. The specification defines the usual operations *insert*, *remove* and *has* as

Figure 17. Specification of Integer Set

integer_set = state machine is has, remove, insert, union, common_element_?

has = non-derived V-function(s:state,i:integer) returns Boolean
Appl. Cond.: true
Initial Value: false
end has

common_element_? = derived V-function(s_1,s_2:state) returns Boolean
Appl. Cond.: true
Derivation: common_element_? = (∃i)['has'(s_1,i) ∧ 'has'(s_2,i)]
end common_element_?

insert = O-function(s:state,i:integer)
Appl. Cond.: true
Effects: 'has'(s,i) = true
end insert

remove = O-function(s:state,i:integer)
Appl. Cond.: true
Effects: 'has'(s,i) = false
end remove

union = O-function(s_1,s_2:state)
Appl. Cond.: true
Effects: (∀i)['has'(s,i) = has(s_1,i) ∨ has(s_2,i)]
end union

end integer_set

---

well as the operations *union* and *common_element_?* described above. *Union's* effects section defines the mapping of each non-derived V-function in the new state that it creates. Note that a *for all* statement has been added for this purpose. Any greater than unary O-function must define the mapping associated with every non-derived or hidden V-function in the new state that it creates so that this new state is fully characterized.

We shall now formalize this type of specification by making a few extensions to the model in Chapter 2. As in that chapter, each machine is modelled by a set of states, where each state is modelled by a set of functions corresponding to the hidden and non-derived V-functions; O-functions define transitions between states.

6.1 Extensions to the Basic Components

6.1.1 V-functions

6.1.1.1 Non-derived and Hidden V-functions

Non-derived and hidden V-functions are specified as in Figure 18. Note that $\mathfrak{D}$ is now included in the mapping description to indicate that the V-function is a unary operation on the data abstraction defined by the machine. The remiander of the definition is defined in the same manner and retains the same interpretation as in Section 2.1.1.1 of Chapter 2.

Figure 18. Non-derived or hidden V-function v

Mapping Description: $\mathfrak{D}$; D_v; R_v
Applicability Condition: $\mathfrak{A}_v$: $\mathfrak{D} \times D_v \rightarrow$ Boolean
Initial Value: $\underline{init}_v \epsilon [D_v \rightarrow R_v]$

So, the sets D_v and R_v in the V-function's mapping description may not contain any element of the data abstraction defined by the machine. And as before, since the state of the machine is characterized by a set of mappings associated with each non-derived and

hidden V-function, we view the state set $\mathcal{S}$ as a subset of

$$[D_{v_1} \rightarrow R_{v_1}] \times \ldots \times [D_{v_n} \rightarrow R_{v_n}] = \mathcal{D}$$

where $\{v_1,\ldots,v_n\}$ is the set of non-derived and hidden V-functions of the machine.

6.1.1.2 Derived V-functions

A derived V-function v also retains the three sections in its definition. However, these sections' definitions and meanings are not the same as in Section 2.1.1.2 of Chapter 2.

Figure 19. Derived V-function v

Mapping Description: $\mathcal{D}^n$; D_v; R_v
Applicability Condition: $W_v: \mathcal{D}^n \times D_v \rightarrow$ Boolean
Derivation: der v such that $(\underline{\text{der}}\ v_{S^n}) \in [D_v \rightarrow R_v]$ for states S^n

---

As before, the *derivation section* defines a function schema, denoted der v, expressed as the composition of the non-derived and hidden V-functions of the machine and other functions associated with the elements of D_v. But, if v is a greater than unary operation, der v also specifies the state in which each non-derived or hidden V-function should be interpreted. For any states S^n, the mapping associated with the schema is denoted by $(\underline{\text{der}}\ v_{S^n})$.

As an example, consider the derivation section of *common_element_?* in Figure 17. For any two states S_1 and S_2, *common_element_?* returns the value

$$(\exists i)[\text{has}(s_1,i) \wedge \text{has}(s_2,i)].$$

This value is, of course, dependent on the mappings associated with *has* in state S_1 and *has* in state S_2.

Now, for any states S^n, the mapping associated with der v is a member of $[D_v \rightarrow R_v]$ where D_v and R_v are specified by v's *mapping description* . These sets D_v and R_v can not contain any elements of the data abstraction defined by the machine.

Finally, the *applicability condition* specifies a partial function $\mathfrak{A}_o$ from $\mathfrak{D}^n \times D_v$ into the Booleans.

6.1.2 O-functions

O-functions too have the meaning and interpretation of the three sections in their definition changed.

Figure 20. O-function o

Mapping Description: $\mathfrak{D}^n$; D_o
Applicability Condition: $\mathfrak{A}_o$: $\mathfrak{D}^n \times D_o \rightarrow$ Boolean
Effects Section: $\mathfrak{T}_o$: $\mathfrak{D}^n \times D_o \rightarrow \mathfrak{D}$

As with derived V-functions, the *mapping description* now contains $\mathfrak{D}^n$ and D_o to reflect the O-functions' extended capability. Furthermore, D_o is constrained so that it contains no elements of the data abstraction defined by the machine. The applicability condition and effects section are also extended to reflect the O-functions' new interpretation. The *applicability condition* of an O-function now defines a partial function $\mathfrak{A}_o$ from

$\mathfrak{D}^n \times D_o$ into the Booleans. Similarly, the *effects section* of an O-function now defines a partial function $\mathfrak{T}_o$ from $\mathfrak{D}^n \times D_o$ into $\mathfrak{D}$.

6.2 The Semantics of a State Machine

6.2.1 The State Set of a State Machine

Our purpose in this section is to define $\mathfrak{S}$. Here, we shall use the same approach outlined in Section 2.2.1, taking the transitive closure of the initial state Q under the state transition function. The *initial state* Q is the tuple $(\underline{init}_{v_1},...,\underline{init}_{v_n})$ containing the mappings derived from the initial value section of each of the non-derived and hidden V-functions $\{v_1,...,v_n\}$. Furthermore, the next state function has the following definition.

Definition
Let o be an O-function with mapping description $\mathfrak{D}^n; D_o$,
mapping $\mathfrak{A}_o$ in its applicability condition and mapping $\mathfrak{T}_o$ in its effects section.
Let $a \in D_o$ and $R \in \mathfrak{D}^n$.
Then,

$$NEXT(R,o,a) = \begin{cases} \mathfrak{T}_o(R,a) & \text{if } \mathfrak{A}_o(R,a)\text{=true} \\ R & \text{if } \mathfrak{A}_o(R,a)\text{=false} \end{cases}$$

Thus, the state set is generated as follows.

1) $Q \in \mathbb{S}$.

2) If o is an O-function with mapping description $\mathbb{D}^n$; D_o and $S^n \in \mathbb{S}^n$, then if NEXT(S^n,o,a) is defined, NEXT(S^n,o,a)$\in \mathbb{S}$ where $a \in D_o$.

3) These are the only elements of $\mathbb{S}$.

Note that in 2) above NEXT(S^n,o,a) may be undefined. As was explained in Chapter 2, this depends on the partial functions $\mathbb{T}_o$ and $\mathbb{H}_o$. To guarantee that NEXT is always defined, we introduce the notion of a well-defined state machine.

Definition
A state machine is *well-defined* if for any O-function o with mapping description $\mathbb{D}^n$; D_o and for any $S^n \in \mathbb{S}^n$, NEXT(S^n,o,a) is defined where $a \in D_o$.

This definition guarantees that in a well-defined state machine, for every O-function o with mapping description $\mathbb{D}^n$; D_o, $\mathbb{H}_o$ is a total function from $\mathbb{S}^n \times D_o$ into the Booleans and $\mathbb{T}_o$ is a total function from $\{(S^n,a) \in \mathbb{S}^n \times D_o \mid \mathbb{H}_o(S^n,a)\}$ into $\mathbb{S}$.

6.2.2 The Semantics of V-functions and O-functions

With this definition of the state set $\mathbb{S}$ of a state machine specification, it is possible to formally define the meaning of the O-functions and V-functions. This will be done by defining mappings V-Eval for V-functions and O-Eval for O-functions such that

$$\text{V-Eval}: \mathbb{S}^n \times NV \rightarrow [A \rightarrow R]$$

and

$$\text{O-Eval}: \mathbb{S}^n \times NO \rightarrow [A \rightarrow \mathbb{S}]$$

where NV is the set of V-function names, A is the set of arguments, R is the set of results and NO is the set of O-function names. Note that the domains of V-Eval and O-Eval

have been changed to reflect the extensions made to the V-functions and O-functions.

O-Eval will be defined first. Now, given an O-function o with mapping description $\mathfrak{D}^n$; D_o and applicability condition $\mathfrak{A}_o$, for any $S^n \in \mathfrak{S}^n$, O-Eval returns a function from D_o into $\mathfrak{S}$ U {error}. So, using lambda notation,

$$\text{O-Eval}(S^n,o) = \lambda a.[\mathfrak{A}_o(S^n,a) \rightarrow \text{NEXT}(S^n,o,a), \underline{\text{error}}]$$

O-Eval(S^n,o) is not necessarily total since either $\mathfrak{A}_o(S^n,a)$ or NEXT(S^n,o,a) can be undefined. However, O-Eval(S^n,o) is always a total function in a well-defined state machine.

For any non-derived or hidden V-function v and state S, V-Eval will return a function from D_v into R_v U {error}. So for any non-derived or hidden V-function v with applicability condition $\mathfrak{A}_v$,

$$\text{V-Eval}(S,v) = \lambda a.[\mathfrak{A}_v(S,a) \rightarrow v_S(a), \underline{\text{error}}]$$

Finally, for a derived V-function v with mapping description $\mathfrak{D}^n$; D_o, applicability condition $\mathfrak{A}_v$ and derivation $\underline{\text{der}}$ v,

$$\text{V-Eval}(S^n,v) = \lambda a.[\mathfrak{A}_v(S^n,a) \rightarrow (\underline{\text{der}}\ v_{S^n})(a), \underline{\text{error}}]$$

where $S^n \in \mathfrak{S}^n$.

Note that the function that V-Eval evaluates to is not necessarily defined over the entire set D_v since the applicability condition can be undefined or, depending on the type of V-function, $v_S(a)$ or $(\underline{\text{der}}\ v_{S^n})(a)$ can be undefined when the applicability condition evaluates to true. When this is not the case, we say the state machine is consistent.

Definition

A state machine is *consistent* if,

for every state $S \in \mathfrak{S}$ and non-derived or hidden V-function v,

V-Eval(S,v) is a total function from D_v into $R_v \cup$ {error}

and if,

for every derived V-function v with mapping description $\mathfrak{D}^n$; D_v and $S^n \in \mathfrak{S}^n$,

V-Eval(S^n,v) is a total function from D_v into $R_v \cup$ {error}.

In a consistent state machine, for non-derived and hidden V-functions, v_S is always a total function from $\{x \in D_v \mid \mathfrak{A}_v(S,x)\}$ into R_v and, for derived V-functions, (der v_{S^n}) is always a total function from $\{x \in D_v \mid \mathfrak{A}_v(S^n,x)\}$ into R_v.

7. Conclusions

The aim of this thesis has been the development of a formal specification technique for data abstractions based on Parnas' ideas. First, a general model for the semantics of a state machine specification was developed. This model gave an effective construction for the state set of a state machine and then used the state set to formalize the semantics of the V-functions and the O-functions. Also, the notions of a well-defined and of a consistent state machine were introduced. Next this abstract model was used to formalize the semantics of a concrete specification language for state machines. This language was used to specify a number of data abstractions and also to illustrate how to prove a particular state machine is well-defined and consistent. Then a proof methodology to use with state machine specifications was discussed and illustrated. This methodology employed the homomorphism property to establish the correctness of an implementation of a state machine specification. Finally the model for the semantics of a state machine specfication was extended. This new model allowed the specification of a greater class of data abstractions than the previous one.

In this final chapter, the usefulness of the state machine specification technique is evaluated and reviewed. This evaluation is then followed by some suggestions for further research on state machine specifications.

7.1 Evaluation

The state machine specification technique is best suited for the specification of data abstractions. Its conceptual basis of a group of functions operating on a state set matches quite well the notion of a data abstraction where a group of functions operate on a collection

of objects. To construct a state machine specification of a data abstraction, one must model the objects of the data abstraction using the V-functions of the machine. In a sense, this corresponds to modelling the objects of the data abstraction by using infinite arrays. So the state machine technique is a variant of the *abstract model approach* [Berzins 78], [Yonezawa 77] where one is restricted to modelling objects of the data abstraction by using infinite arrays.

In the abstract model approach, the objects of a data abstraction are represented in terms of other data abstractions with known properties established by formal specifications given in advance. Then the operations of the data abstraction being defined can be specified in terms of the operations of the known abstractions selected as the representation. So, a model for the data abstraction is developed. This differs from *axiomatic specifications* [Zilles 74], [Goguen 75], [Guttag 75] where the behavior of a data abstraction is given by axioms relating its operations. Currently research is being done on both these techniques. Since any comparison made between abstract model and axiomatic specifications will apply to state machine specifications, we shall limit the following discussion to a comparison of the abstract model and state machine techniques.

In using the abstract model approach one is free to choose the data abstractions used to represent the specified objects. Thus it appears that abstract model specifications would be easier to construct than state machine specifications. In fact, one can encounter difficulty in using the state machine technique to specify an abstraction whose objects can not be modelled well by arrays such as *lists* or *trees*.

Another issue in constructing state machine specifications is that one usually wishes to write a specification that is well-defined and consistent. So it will be necessary to prove

that these two properties hold. Studying the proofs in Appendix 2, it appears, at first glance, that the proofs of these two properties are rather complex. However, while the proofs in Appendix 2 may indeed be somewhat cumbersome, they are basically quite simple and straight forward. They primarily rely on the definitions in Chapter 3. The most "creative" step in the proofs was the introduction of the lemmas. Even here, however, the creativity involved was minimal. For example, when I started work on showing that the specification of *queue* was well-defined, I did not begin with the first lemma. It was only when I was forced to show that both *front* and *back* could be evaluated in any state that I realized that I had to prove this lemma. So, in carrying out the method outlined in Section 2.2.4, I found the extra condition I needed to simplify the proof. This experience was repeated when I attempted to show that V-Eval(S,first_element) was total.

I feel that in most cases it will be necessary to prove simple lemmas to help in carrying out proofs of properties of state machine specifications. However, it appears that these lemmas are usually quite easy to discover and that the actual steps in the proof will involve one in time consuming, but not difficult, work.

However, it appears that proving the correctness of an implementation will be more difficult. Here not only is it necessary to show that the homomorphism property holds but one must also show that the abstraction function is an onto mapping. This latter task is not simple. One must first characterize every equivalence class of the state set and then show that there exists an element of the concrete objects that maps into this element.

To prove that some property holds for an abstract model specification of a data abstraction, one must show that the property holds in the data abstraction's model. The difficulty of this proof depends on how well chosen the model is. Thus proving that a

property holds in a state machine specification can be easier or harder than proving that the same property holds in an abstract model specification according to the aptness of the latter specificiation's model. However proving the correctness of a data abstraction specified by either technique appears to involve equal difficulty.

7.2 Topics for Further Research

One area for further research is to determine the usefulness of the state machine specification technique. Specifically, can state machine specifications be used successfully in the design and development of large scale software systems? Research that should help answer this question is currently being done at SRI. They have used state machine type specifications in the design of a provably secure operating system [Neumann 77] and are developing a methodology for the development of software that uses state machine type specifications. Their preliminary results in this area have been encouraging.

Another research area is the extension of the state machine specification technique's error handling capabilities. At present, when the applicability condition of an O-function or V-function evaluates to **false**, the function returns the special symbol error. Clearly, this does not give the user any clue as to what has caused the error. More information should be given. Furthermore, the meaning of returning an error message has not been discussed.

The specifications could be extended to allow one to define more descriptive error messages. For example, *cardinality* in the finite set specification of Chapter 5 could return an error message such as "too many elements" when one attempts to add more than 100 integers to the set. Parnas has noted that more information is needed to describe how his specifications handle errors [Parnas 72, 75].

Another extension to state machine specifications can be made in the class of data abstractions specified. Throughout this thesis, we have assumed that the data abstractions specified by a state machine are *immutable*. In an immutable abstraction, the objects of the abstraction are constants, i.e., their properties do not vary over time. This corresponds to the behavior of the states in a state machine specification. An O-function o, when given a state S and $x \in D_o$, does not modify S, but instead returns a new state S^*. Furthermore, if the O-function o is again given S and x later in a computation, it will still return S^*. Similar behavior is also exhibited by a V-function. However in a mutable data abstraction the behavior of the objects may change. An operation of a mutable data abstraction when passed a mutable object may return a different result at different instances in the computation history. Thus, an obvious topic for further research is to modify the state machine technique to allow the specification of mutable data abstractions. Liberato [83] is studying how abstract model specifications can be used to specify mutable data abstractions.

Appendix I - Undecidable Properties of State Machines

In this appendix, we shall show that it is impossible to decide algorithmically whether or not a state machine, specifed in ALMS, is well-defined or consistent. This is established by reducing both problems to the *blank tape halting problem* for Turing machines. The blank tape halting problem is the problem of determining, given a particular Turing machine T, whether or not T halts when started on blank tape. This problem is undecidable [Hennie 77].

The definition of a Turing machine used here is given by [Hennie 77]. A Turing machine consists of an infinitely long tape coupled to a finite control unit. The tape, which acts as the machine's memory unit, is ruled off into squares. Each square may be inscribed with a single symbol from a finite alphabet Σ, or it may be blank. The special symbol $\#$ is used to represent a blank. The control unit can shift the tape back and forth and is able to examine one square at any time.

The control unit is capable of assuming any one of a fixed, finite number of states. We shall only consider deterministic Turing machines. So, at any given time, the state of the control unit, together with the currently scanned tape symbol, uniquely determines the behavior of the Turing machine. The Turing machine has two actions: it may either *halt* or carry out a *move*. Each move consists of writing a symbol on the currently scanned tape square, shifting the tape one square to the left or right, and causing the control unit to enter a new state.[1] The action of the Turing machine is characterized by the successive moves that

---

1. The symbol that the Turing machine writes need not differ form the symbol that is already there and the new state need not differ from the current state.

occur when, initially, the control unit assumes some predesignated starting state and some finite number of the tape squares are inscribed with symbols and the remainder are left blank.

A Turing machine is represented by a group of quintuples of the following form

$$q_i\ s_j\ s_k\ d_l\ q_h$$

where q_i is the current state

s_j is the symbol under the tape head

s_k is the symbol to be printed on the tape

$d_l \in \{\underline{\text{right}}, \underline{\text{left}}\}$ is the direction of the head's movement

q_h is the next state

Each quintuple must have a distinct prefix $q_i\ s_j$. The Turing machine halts when the control unit is in a state q and is scanning a symbol s such that q s is not the prefix of any quintuple.

So, assume we are given a Turing machine $\mathfrak{M}$ with quintuples

$$\begin{array}{l} q_{i_1}\ s_{j_1}\ s_{k_1}\ d_{l_1}\ q_{h_1} \\ \quad\vdots \\ q_{i_n}\ s_{j_n}\ s_{k_n}\ d_{l_n}\ q_{h_n} \end{array}$$

and initial state q_{i_1}.

Now, consider the state machine given in Figure 21.

For the notation $\delta(d)$,

$$\delta(d) = \begin{cases} 1 & \text{if } d = \underline{\text{right}} \\ -1 & \text{if } d = \underline{\text{left}} \end{cases}$$

Figure 21. Turing_machine_$\mathfrak{M}$_1

Turing_machine_$\mathfrak{M}$_1 = state machine is tape, state, head_pos, move

state = non-derived V-function( ) returns character string
Appl. Cond.: true
Initial Value: q_{i_1}
end state

head_pos = non-derived V-function( ) returns integer
Appl. Cond.: true
Initial Value: 0
end head_pos

tape = non-derived V-function(i:integer) returns character string
Appl. Cond.: true
Initial Value: Λ
end tape

well_defined_? = hidden V-function( ) returns integer
Appl. Cond.: true
Initial Value: undefined
end well_defined_?

move = O-function( )
Appl. Cond.: true
Effects:
if state = q_{i_1} $\wedge$ tape(head_pos) = s_{j_1} then 'state' = q_{h_1}
if state = q_{i_1} $\wedge$ tape(head_pos) = s_{j_1} then 'head_pos' = head_pos + $\delta(d_{l_1})$
if state = q_{i_1} $\wedge$ tape(head_pos) = s_{j_1} then 'tape'(head_pos) = s_{k_1}
.
.
.
if state = q_{i_n} $\wedge$ tape(head_pos) = s_{j_n} then 'state' = q_{h_n}
if state = q_{i_n} $\wedge$ tape(head_pos) = s_{j_n} then 'head_pos' = head_pos + $\delta(d_{l_n})$
if state = q_{i_n} $\wedge$ tape(head_pos) = s_{j_n} then 'tape'(head_pos) = s_{k_n}
if $\sim$((state = q_{i_1} $\wedge$ tape(head_pos) = s_{j_1}) $\vee$...$\vee$ (state = q_{i_n} $\wedge$ tape(head_pos) = s_{j_n}))
then tape'(head_pos) = well_defined_?
end move

end Turing_machine_$\mathfrak{M}$_1

Turing_machine_$\mathfrak{M}$_1 simulates $\mathfrak{M}$ by having a state in its state set for every step in $\mathfrak{M}$'s computation.

Now, Turing_machine_$\mathfrak{M}$_1 is not well-defined if and only if $\mathfrak{M}$ halts when started on blank tape. Assume $\mathfrak{M}$ halts when started on blank tape. Then there is a state S of Turing_machine_$\mathfrak{M}$_1 corresponding to the final step in $\mathfrak{M}$'s computation. In S,

$$\sim((\text{state} = q_{i_1} \wedge \text{tape(head\_pos)} = s_{j_1}) \vee \ldots \vee (\text{state} = q_{i_n} \wedge \text{tape(head\_pos)} = s_{j_n}))$$

evaluates to true. But, the equation

'tape'(head_pos) = well_defined_?

is undefined since the V-function *well_defined_?* is never assigned a value. Hence, Turing_machine_$\mathfrak{M}$_1 is not well-defined.

Going the other way, assume Turing_machine_$\mathfrak{M}$_1 is not well-defined. This can only be caused by

'tape'(head_pos) = well_defined_?

since the other equations only use total V-functions. Thus,

$$\sim((\text{state} = q_{i_1} \wedge \text{tape(head\_pos)} = s_{j_1}) \vee \ldots \vee (\text{state} = q_{i_n} \wedge \text{tape(head\_pos)} = s_{j_n}))$$

is satisfied so $\mathfrak{M}$ must halt.

Now, consider the state machine specification in Figure 22. This state machine is not consistent if and only if $\mathfrak{M}$ halts when started on blank tape.

First, assume $\mathfrak{M}$ halts on blank tape. Then there is a state S for which

$$\sim((\text{state} = q_{i_1} \wedge \text{tape(head\_pos)} = s_{j_1}) \vee \ldots \vee (\text{state} = q_{i_n} \wedge \text{tape(head\_pos)} = s_{j_n}))$$

evaluates to true. Consider O-Eval(S,move)(λ) = S'. In S', *switch* evaluates to true so the V-function *consistent_?* is not total. By reversing this argument, it is apparent that if Turing_machine_$\mathfrak{M}$_2 is not consistent, $\mathfrak{M}$ halts on blank tape.

Figure 22. Turing_machine_$\mathfrak{M}$_2

Turing_machine_$\mathfrak{M}$_2 = state machine is tape, state, consistent_?, head_pos, move

state = non-derived V-function( ) returns character string
Appl. Cond.: true
Initial Value: q_{i_1}
end state

head_pos = non-derived V-function( ) returns integer
Appl. Cond.: true
Initial Value: 0
end head_pos

tape = non-derived V-function(i:integer) returns character string
Appl. Cond.: true
Initial Value: Λ
end tape

consistent_? = non-derived V-function( ) returns integer
Appl. Cond.: switch
Initial Value: undefined
end consistent_?

switch = hidden V-function( ) returns Boolean
Appl. Cond.: true
Initial Value: false
end switch

move - O-function( )
Appl. Cond.: true
Effects:
if state = q_{i_1} ∧ tape(head_pos) = s_{j_1} then 'state' = q_{h_1}
if state = q_{i_1} ∧ tape(head_pos) = s_{j_1} then 'head_pos' = head_pos + $\delta(d_{j_1})$
if state = q_{i_1} ∧ tape(head_pos) = s_{j_1} then 'tape'(head_pos) = s_{k_1}

.
.
.

if state = q_{i_n} ∧ tape(head_pos) = s_{j_n} then 'state' = q_{h_n}
if state = q_{i_n} ∧ tape(head_pos) = s_{j_n} then 'head_pos' = head_pos + $\delta(d_{j_n})$
if state = q_{i_n} ∧ tape(head_pos) = s_{j_n} then 'tape'(head_pos) = s_{k_n}
if ~((state = q_{i_1} ∧ tape(head_pos) = s_{j_1}) ∨ ... ∨ (state = q_{i_n} ∧ tape(head_pos) = s_{j_n}))
then 'switch' = true
end move

end Turing_machine_M_2

Appendix II - Proofs

This appendix contains the proofs of the lemmas and the theorems in Section 3.3 of Chapter 3. We shall first show that the specification is well-defined. This will be done by initially proving a lemma that captures the key properties necessary to insure that the machine is well-defined. Then, with the aid of this lemma, we will directly establish that the specification is well-defined.

First, some notational details must be handled. We shall denote the initial state of *queue* by Q, its state set by $\mathcal{S}$ and assume

$$\mathcal{S} \subset [D_{storage} \rightarrow R_{storage}] \times [D_{back} \rightarrow R_{back}] \times [D_{front} \rightarrow R_{front}].$$

Lemma For any $S \in \mathcal{S}$, $back_S \in [\{\lambda\} \rightarrow integer]$ and $front_S \in [\{\lambda\} \rightarrow integer]$ and $back_S \neq \phi \neq front_S$ where ϕ is the null set.

Proof by induction:

Basis: By definition,

$$back_Q = \{(\lambda,0)\} \text{ and } front_Q = \{(\lambda,-1)\}.$$

Inductive step: Assume for all

$$S = NEXT(...NEXT(NEXT(Q,o_1,a_1),o_2,a_2),...o_{n-1},a_{n-1}) \in \mathcal{S}$$

where $a_i \in D_{o_i}$, $n \geq 2$ and $o_i \in \{insert,delete\}$ that $back_S \in [\{\lambda\} \rightarrow integer]$ and $front_S \in [\{\lambda\} \rightarrow integer]$ and $back_S \neq \phi \neq front_S$. We must show for all

$$S' = NEXT(S,\omega,x) \in \mathcal{S}$$

where $x \in D_\omega$ and $\omega \in \{insert,delete\}$ that $back_{S'} \in [\{\lambda\} \rightarrow integer]$ and $front_{S'} \in [\{\lambda\} \rightarrow integer]$ and $back_{S'} \neq \phi \neq front_{S'}$

Case 1: $back_{S^\bullet}$

Case 1a: ω = insert

Then $S^\bullet$ = NEXT(S , insert , x).

Since p_I(S , true , insert , x) = true,

$S^\bullet = TE(S , insert , x , 'storage'(back - 1) = i , 'back' = back - 1)$

$= E(E(S , insert , x , 'storage'(back - 1) = i) , insert , x , 'back' = back - 1)$

$= S_1$

Since $S^\bullet \in$ S, S_1 is defined and hence

$S_2 = E(S , insert , x , 'storage'(back - 1) = i)$ is defined.

Then $S^\bullet = E(S_2 , insert , x , 'back' = back - 1)$

$= (storage_{S_2} , \{(\lambda , p_I(S_2 , back - 1 , insert , x))\} , front_{S_2})$

$= (storage_{S_2} , \{(\lambda , S_2 [back - 1])\} , front_{S_2})$

$= (storage_{S_2} , \{(\lambda , back_{S_2} - 1)\} , front_{S_2})$

Now $back_{S_2} = back_S$ and so by the inductive hypothesis,

it is a member of [{λ} → integer] - {ϕ}.

Thus, $back_{S^\bullet} \in$ [{λ} → integer] - {ϕ}.

Case 1b: ω = delete

Since $S^\bullet$ = NEXT(S , delete , λ) is by assumption defined, there are two cases relating to p_I(S , ~(front = back - 1) , delete , λ)

Case 1b1: p_I(S, ~(front = back - 1) , delete , λ) = false.

Then $S^\bullet$ = S and by the inductive hypothesis,

$back_S \in$ [{λ} → integer] - {ϕ}.

Case 1b2: p_I(S , ~(front = back - 1) , delete , λ) = true.

Then $S^\bullet = TE(S , delete , \lambda , 'front' = front - 1)$.

So $back_S = back_{S^\bullet}$ which, by the inductive hypothesis,

is a member of [{λ} → integer] - {ϕ}.

Case 2: $front_{S^\bullet}$

Case 2a: ω = insert

Here, $S^\bullet = TE(S , insert , x , 'storage'(back - 1) = i , 'back' = back - 1)$.

So, $front_{S^\bullet} = front_S \in$ [{λ} → integer] - {ϕ} by the inductive hypothesis.

Case 2b: ω = delete

Case 2b1: p(S , ~(front = back - 1) , delete , λ) = false.

Then S* = S and by the inductive hypothesis,

$front_S \in [[\lambda] \rightarrow integer] - \{\phi\}$.

Case 2b2: p(S , ~(front = back - 1) , delete , λ) = true.

Then S* = $\mathbb{TE}$(S , delete , λ , 'front' = front - 1)

= $\mathbb{E}$(S , delete , λ , 'front' = front - 1)

= (storage$_S$, back$_S$, {(λ , p(S , front - 1 , delete , λ))})

= (storage$_S$, back$_S$, {(λ , S $\models$ front - 1)})

= (storage$_S$, back$_S$, {(λ , front$_S$ - 1)})

By the inductive hypothesis, $front_S \in [[\lambda] \rightarrow integer] - \{\phi\}$ so

$front_{S^*} \in [[\lambda] \rightarrow integer] - \{\phi\}$. ∎

We can now prove that the machine is well-defined using the above lemma. Three properties must be established: i) the applicability conditions of the O-functions *insert* and *delete* are defined; ii) the next state function is defined for both *insert* and *delete*; and, finally, iii) the ordering of the equations in both *insert's* and *delete's* effects sections is immaterial. Note that iii) is trivially established since *delete* has only one equation in its effects section and the two equations in *insert's* effects section modify different V-functions. Thus, it is only necessary to deal with i) and ii). We now complete the proof.

Case 1: The Applicability Condition

Case 1a: Insert's Applicability Condition

p(S , true , insert , x) = S $\models$ true

= true

Case 1b: Delete's Applicability Condition

p(S , ~(front = back - 1) , delete , λ) = S $\models$ ~(front = back - 1)

= ~(front$_S$ = back$_S$ - 1)

By the lemma, both $front_S$ and $back_S$ are members of $[(\lambda) \rightarrow integer] - (\phi)$

so $\sim(front_S = back_S - 1)$ is defined.

Case 2: The Next State Function

Case 2a: NEXT(S , insert , x)

By Case 1a, p_s(S , true , insert , x) = true

So,

NEXT(S , insert , x) = $\mathbb{TE}$(S , insert , x , 'storage'(back - 1) = i , 'back' = back - 1)

= $\mathbb{E}(\mathbb{E}$(S , insert , x , 'storage'(back - 1) = i) , insert , x , 'back' = back - 1)

Now,

$\mathbb{E}$(S , insert , x , 'storage'(back - 1) = i)

= $(\lambda x.[(x = p_s(S,back-1,insert,x)) \rightarrow p_s(S,i,insert,x), storage_S(x)], back_S, front_S)$

= $(\lambda x.[(x = back_S - 1) \rightarrow x , storage_S(x)] , back_S , front_S)$

= S^*

Note that $back_S$ is defined by the lemma so S^* is defined.

Now,

$\mathbb{E}(S^*$, insert , x , 'back' = back - 1)

= $(storage_{S^*} , \{(\lambda , p_s(S^* , back - 1 , insert , x))\} , front_S)$

= $(storage_{S^*} , \{(\lambda , back_{S^*} - 1)\} , front_S)$

= $(storage_{S^*} , \{(\lambda , back_S - 1)\} , front_S)$

By the lemma, $back_S$ is defined and hence NEXT(S , insert , x) is defined.

Case 2b: NEXT(S , delete , λ)

Case 2b1: p_s(S , $\sim$(front = back - 1) , delete , λ) = false

Then NEXT(S , delete , λ) = S.

Case 2b2: p_s(S , $\sim$(front = back - 1) , delete , λ) = true

Then NEXT(S , delete , λ) = $\mathbb{TE}$(S , delete , λ , 'front' = front - 1)

= $\mathbb{E}$(S , delete , λ , 'front' = front - 1)

= $(storage_S , back_S , \{(\lambda , p_s(S , front - 1 , delete , \lambda))\})$

= $(storage_S , back_S , \{(\lambda , S \vdash front - 1)\})$

= $(storage_S , back_S , \{(\lambda , front_S - 1)\})$

So by the lemma, NEXT(S , delete , λ) is defined. ∎

We will now show that the specification is consistent. This involves proving that the four V-functions are total. For *front* and *back* note that

1) back

$$\begin{aligned} \text{V-Eval}(S, \text{back}) &= \lambda a.[\mu(S, \text{true}, \text{back}, \lambda) \rightarrow \text{back}_S, \underline{\text{error}}] \\ &= \lambda a.[\text{true} \rightarrow \text{back}_S, \underline{\text{error}}] \\ &= \lambda a.\text{back}_S \end{aligned}$$

2) front

$$\begin{aligned} \text{V-Eval}(S, \text{front}) &= \lambda a.[\mu(S, \text{true}, \text{front}, \lambda) \rightarrow \text{front}_S, \underline{\text{error}}] \\ &= \lambda a.[\text{true} \rightarrow \text{front}_S, \underline{\text{error}}] \\ &= \lambda a.\text{front}_S \end{aligned}$$

By the lemma, both back_S and front_S are defined.

To see that both V-Eval(S , storage) and V-Eval(S , first_element) are total, we must again introduce a lemma and then prove the desired results directly from the lemma. The lemma describes the domain of *storage*.

Lemma

For any $S \in \mathfrak{S}$, if $\text{front}_S \geq k \geq \text{back}_S$, then $\text{storage}_S(k)$ is defined.

Basis: Since $\text{front}_Q = ((\lambda,-1))$ and $\text{back}_Q = ((\lambda,0))$, the lemma trivially follows.

Inductive step: Assume for all

$$S = \text{NEXT}(\ldots\text{NEXT}(\text{NEXT}(Q, o_1,a_1),o_2,a_2),\ldots,o_{n-1},a_{n-1}) \in \mathfrak{S}$$

where $a_i \in D_{o_i}$, $n \geq 2$ and $o_i \in$ {insert,delete} that if $\text{front}_S \geq k \geq \text{back}_S$, then $\text{storage}_S(k)$ is defined. We must show for all

$$S' = \text{NEXT}(S,\omega,x) \in \mathfrak{S}$$

where $x \in D_\omega$ and $\omega \in \{insert, delete\}$ that if $front_{S\bullet} \geq k \geq back_{S\bullet}$, then $storage_{S\bullet}(k)$ is defined.

Case 1: ω = insert

Note $x \in D_{insert}$.

Case 1a: $front_{S\bullet} = back_{S\bullet}$

Then $storage_{S\bullet}(front_{S\bullet})$ evaluates to x

due to the equations 'storage'(back - 1) = i and 'back' = back - 1.

Case 1b: $front_{S\bullet} \neq back_{S\bullet}$

Then $front_{S\bullet} > back_{S\bullet}$

and $front_{S\bullet} = front_S$ and $back_{S\bullet} = back_S - 1$.

So for $front_{S\bullet} \geq k \geq back_{S\bullet} + 1$, $storage_{S\bullet}(k)$

is defined by the inductive hypothesis.

Also, $storage_{S\bullet}(back_{S\bullet})$ evaluates to x

due to the equations 'storage'(back - 1) = i and 'back' = back - 1.

Case 2: ω = delete

Since $front_{S\bullet} = front_S - 1$ and $back_{S\bullet} = back_S$,

for $front_{S\bullet} \geq k \geq back_{S\bullet}$, $storage_S(k)$

is defined by the inductive hypothesis. ∎

To see that V-Eval(S , storage) is total, note that

$$V\text{-}Eval(S, storage) = \lambda a.[\mu i(S, front \geq i \geq back, storage, a) \rightarrow storage_S(a), \underline{error}]$$
$$= \lambda a.[front_S \geq a \geq back_S \rightarrow storage_S(a), \underline{error}]$$

The desired result immediately follows from the lemma. To see that V-Eval(S , first_element) is total, note that for any $S \in$

V-Eval(S , first_element)

$= \lambda a.[\mu i(S, \sim(front = back - 1), first\_element, \lambda) \rightarrow \mu i(S, storage(front), first\_element, \lambda), \underline{error}]$

$= \lambda a.[\sim(front_S = back_S - 1) \rightarrow storage_S(front_S), \underline{error}]$

If $\sim(front_S = back_S - 1)$ is false, then

$$V\text{-}Eval(S, first\_element) = \lambda a.\underline{error}.$$

Otherwise,

$$V\text{-}Eval(S, first\_element) = \lambda a.storage_S(front_S)$$

Now $front_S \geq back_S$ so, by the lemma, $storage_S(front_S)$ is defined. Thus, we conclude V-Eval(S , first_element) is total.

References

[Berzins 78] V. Berzins
Abstract Model Specifications for Data Abstractions
forthcoming Ph. D. Thesis M.I.T. (1978)

[Burstall 69] R. M. Burstall
"Proving Properties of Programs by Structural Induction"
Computer Journal Vol.12 (1969) pp.41-48

[Curry 50] H. B. Curry
"The Logic of Program Composition"
Applications Scientifique de la Logique Mathematique
Actes du 2^e Colloque International de Logique Mathematique 1952
Paris: Gauthier-Villars (1954) pp.97-102 [Paper written in March, 1950]

[Knuth 76] D. E. Knuth, L. T. Pardo
The Early Development of Programming Languages
Stanford University, STAN-CS-76-562 (August 1976)

[Fraleigh 67] J. Fraleigh
A First Course in Abstract Algebra
Addison-Wesley Reading Mass. (1967)

[Goguen 75] J. A. Goguen, J. W. Thatcher, E. G. Wagner, J. B. Wright
"Abstract Data Types as Initial Algebras and the Correctness of Data Representations"
Proc. of the Conference on Computer Graphics, Pattern Recognition and Data Structures
(May 1975) pp.89-93

[Guttag 75] J. V. Guttag
The Specification and Application to Programming of Abstract Data Types
Ph. D. Thesis, University of Toronto CSRG-59 (1975)

[Hennie 77] F. Hennie
Introduction to Computability
Addison-Wesley Reading Mass (1977)

[Hoare 72a] C. A. R. Hoare
"Proofs of Correctness of Data Representations"
Acta Informatica Vol.1 No.4 (1972) pp.271-281

[Hoare 72b] C. A. R. Hoare
"Notes on Data Structuring"
Structured Programming pp.83-174
A.P.I.C. Studies in Data Processing No.8
Academic Press London-New York (1972)

[Kapur 78] D. Kapur
Towards a Theory of Data Abstractions
forthcoming Ph. D. Thesis M.I.T. (1978)

[Liskov and Zilles 74] B. H. Liskov, S. Zilles
"Programming with Abstract Data Types"
Proc. ACM SIGPLAN Conference on Very High Level Languages,
SIGPLAN Notices Vol.9 No.4 (Apr. 1974) pp.50-59

[Liskov and Zilles 75] B. H. Liskov, S. Zilles
"Specification Techniques for Data Abstractions"
IEEE Transcations on Software Engineering SE-1 Vol.1 (1975) pp.7-19

[Liskov and Berzins 77] B. H. Liskov, V. Berzins
"An Apppraisal of Program Specifications"
Computation Structures Group Memo 141-1, M.I.T. (April 1977)

[Milne and Strachey 76]
A Theory of Programming Language Semantics
Halsted Press New York (1976)

[Neumann 74] P. G. Neumann
Towards a Methodology for Designing Large Systems and Verifying their Properties
Gesellschaft fur Informatif, Berlin (1974)

[Neumann 77] P. G. Neumann, R. S. Boyer, R. J. Feiertag, K. N. Levitt, L. Robinson
A Provably Secure Operating System: The System, its Applications, and Proofs
SRI Fianl Report, Project 4332 (February 1977)

[Parnas 72] D. L. Parnas
"A Technique for the Specficiation of Software Modules, with Examples"
Comm. ACM Vol.15 No.5 (May 1972) pp.330-336

[Parnas 75] D. L. Parnas
"More on Specfication Techniques for Software Modules"
Research Group on Operating Systems I, T. H. Darmstadt, Fed. Rep. of Germany

[Pratt 76] V. R. Pratt
"Semantical Considerations on Floyd-Hoare Logic"
17th IEEE Symposium on the Foundations of Computer Science (Oct. 1976) pp.109-121

[Price 73] W. L. Price
Implications of a Virtual Memory Mechanism for Implementing Protection in a Family of Operating Systems
Ph. D. Thesis, Carnegie-Mellon University (June 1973)

[Robinson 75] L. Robinson, K. Levitt, P. Neumann, A. Saxena
"On Attaining Reliable Software for a Secure Operating System"
Proc. International Conf. on Reliable Software (1975) pp.267-284

[Robinson 77] L. Robinson, K. Levitt
"Proofs Techniques for Hierarchically Structured Programs"
Comm. ACM Vol.20 No.4 (April 1977) pp.271-283

[Roubine 76] O. Roubine, L. Robinson
SPECIAL Reference Manual
Stanford Research Institute, Technical Report CSG-45 (August 1976)

[Spitzen 76] J. Spitzen, K. Levitt, L. Robinson
An Example of Hierarchical Design and Proof
Stanford Research Institute (January 1976)

[Yonezawa 77] A. Yonezawa
Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics
Ph. D. Thesis, M.I.T. Laboratory for Computer Science TR-191 (December 1977)

[Zilles 74] S. Zilles
"Algebraic Specfication of Data Types"
Project MAC Progress Report, (1974) pp.52-58 M.I.T.