LABORATORY FOR
COMPUTER SCIENCE
(formerly Project MAC)

MASSACHUSETTS
INSTITUTE OF
TECHNOLOGY

MIT/LCS/TR-191

SPECIFICATION AND VERIFICATION TECHNIQUES
FOR
PARALLEL PROGRAMS BASED ON MESSAGE PASSING SEMANTICS

Akinori Yonezawa

This research was supported by the
Office of Naval Research under
Contract No. N00014-75-C-0522

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

MIT/LCS/TR-191

Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics

by

Akinori Yonezawa

December 1977

This research was supported by the Office of Naval Research under contract number N00014-75-C-0522.

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge Massachusetts 02139

Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics

by

Akinori Yonezawa

Submitted to the Department of Electrical Engineering and Computer Science
on December 30, 1977 in partial fulfillment of the requirements
for the Degree of Doctor of Philosophy

Abstract

This thesis presents formal specification and verification techniques for both serial and parallel programs written in SIMULA-like object oriented languages.

These techniques are based on the notion of states of individual objects which are defined uniformly in serial and parallel computations. They can specify and verify the behavior of data and procedural objects in multi-process environments, thus overcoming some of the difficulties in dealing with parallelism which characterized previous work on formal specifications for abstract data types. Among others, the specifications and verifications of a bounded buffer and air line reservation systems are given.

Using a model of a simple post office, we illustrate our specification and verification techniques for systems, such as operating systems and multi-user data base systems, which are characterized by complex internal concurrent activities. It is demonstrated that the specifications of the overall functions of the system which we call task specifications can be derived from specifications of the individual behavior and mutual interaction of the subsystems.

A method of defining states of individual objects as mathematical functions is suggested.

Thesis Supervisor: Carl Hewitt
Title: Associate Professor of Electrical Engineering and Computer Science

Acknowledgements

I would like to express my deep gratitude

to Professor Carl Hewitt whose patience and insightful guidance constantly refired my enthusiasm for this research. His numerous suggestions and advice have been invaluable throughout my graduate work at MIT.

I would like to express my special appreciation

to Professor Barbara H. Liskov whose detailed and illuminating comments on an early version of the draft considerably strengthened this thesis, and

to Professor Joseph C. R. Licklider whose warmth and encouragement have always been valuable sources of support.

My first year graduate work was supported by the Japan Society for the Promotion of Science (Nihon Gakujutsu-shinko-kai) and the Japan Foundation for the Education and Training of Information Processing (Nihon Jyohoshori-Kyoiku-Zaidan).

This research was conducted at the Laboratory for Computer Science (formerly Project MAC) and the Artificial Intelligence Laboratory, Massachusetts Institute of Technology, under the sponsorship of the Office of Naval Research, contract number N00014-75-C-0522.

CONTENTS

1. Introduction

1.1 Formal Specifications and Verifications

A program specification is a description of the desired program behavior. It is necessary to specify what task the program is supposed to perform and what effects (side-effects) are caused by carrying out the intended task.

Program specifications can be written in languages of varying degrees of formality. Although informal languages, such as natural languages, diagrams, and combinations of these, help people to convey intuitive ideas about program behavior, their inherent ambiguity is a drawback. In order to rule out the possibility of ambiguous interpretations, program specifications should be written in formal languages. When formal specifications might be difficult to understand, they may be accompanied by informal descriptions of program behavior.

Formal specifications play an important role in the construction of reliable software. They also provide designers and programmers with an exact communication medium for discussing the properties of program modules in various phases of software construction, such as initial design and coding. Furthermore, they can be used as documentation during the maintenance phase. A formal specification can be viewed as a contract which describes the agreements between the implementors of a program module and its users. The users of a module rely only on the properties derived from its formal specifications, while the implementors need only satisfy the requirements stated in the specifications.

Program verification is the process of proving that a given program (implementation) meets its formal specifications. When a program module M is built on a collection of submodules, their formal specifications can be used in the verification of M. Actual programs (implementations) of the submodules need not be used.

1.2 A Model of Parallel Computation

This thesis is concerned with the techniques for formal specification and verification of both serial and parallel computations.

In order to discuss specification and verification techniques, we must clearly define the computation model on which the execution of programs is based. The computation model used in this thesis is the actor model of computation[Greif-Hewitt75, Hewitt-Baker77], which can be roughly characterized as one obtained by generalizing the computation model used in SIMULA-like object-oriented languages[1] to include *parallelism*.

The fundamental objects in our model of computation are actors, which unify procedures and data structures. An actor is a potentially active object which becomes active when it receives a message. No actor treats other actors as objects to operate on; instead it sends messages (which are also actors) to other actors. Actors behave like data or data structures as well as functions or procedures. For example, a push-down-stack actor pops up and returns its top element when it receives a (*pop:*) message (if it is not empty), and when it receives a (*push:* ●) message, it stores ● as its new top element. A factorial actor returns 6 when it receives 3.

The only activity possible in the model is message passing among actors. More than one transmission of messages may take place concurrently, which models parallel computations. Since processors and processes can be viewed as actors, multi-processor information systems and computer networks are modelled by actor systems. In particular, distributed systems[2] and communicating parallel processes can be easily modelled by actors or systems of actors[Yonezawa-Hewitt77, Hewitt-Baker77].

The concept of an *event* is fundamental in describing the model of computation precisely. An event is the receipt of a message by an actor. A computation is expressed as a partially ordered set of events, where the order relation represents the temporal "precedes" relation. Unordered events can take place concurrently. Thus the partial order of events naturally generalizes serial computations (which are totally ordered sets of events) to parallel

---

[1] Besides SIMULA-67[Dahl-et-al70], CLU[Schaffert-et-al75], ALPHARD[Wulf-et-al75] and SMALL-TALK[Learning-Research-Group76] are examples of such programming languages.

[2] Distributed systems are multi-processor information processing systems which do not rely on the central shared memory for communication.

computations.

1.3 Local State Approach

In this thesis, we propose an approach, called the *local state approach*, for specifying the behavior of actors (objects). In general, the behavior of an actor in response to a message depends upon the past history of messages received by the actor. By defining the *state* of an actor A as *equivalence classes* on the past message histories of A, we can specify the behavior of A in response to a message M in terms of:

(1) the state of A before A receives M,

(2) a set of mutually concurrent events caused by the event where A receives M and

(3) the state of A after A receives M.

Since we assume, in the model of computation, that the order of message arrivals at the same actor is always total, the state of an actor is always well-defined in both serial and parallel computations. Consequently, the behavior of an actor in both serial and parallel computations can be specified in a uniform manner.

We use the term "*local*" to emphasize that our approach does not rely on the notions of the global clock and the global state of a system.[1] The use of global states is often motivated by the use of non-deterministic serial computations as the underlying semantic model for parallel computations. This leads to counter-intuitive serialization of unrelated concurrent events and a large number of possible cases in analyzing properties of

---

[1] The global clock is the unique time reference available within the entire system. The global state of the system at a given time t (by the global clock) is a vector of the states of system components determined at the same time t.

the system.

In our approach, the behavior of a system is specified in terms of the *individual* behavior of system components and their mutual interaction. Such behavior and interaction are described by the states of the system components determined at their *local* times.

1.4 Contributions of the Thesis

Based on the notion of local states, the work presented in this thesis has made several contributions to the area of program specification and verification.

(1) Formal specifications of *Abstract Data Types* with *Parallelism* and *Side-effects*

The importance of abstract data types[Liskov-Zilles74] in the construction of reliable software has been recognized and two approaches to the formal specification technique for abstract data types, i.e. algebraic axiomatic[Zilles74, Spitzen-Wegbreit75, Guttag75] and abstract model[Hoare72, Liskov-Berzins77] approaches, have been proposed. Yet none of the techniques of these approaches are able to deal with parallelism and side-effects. These techniques are only applicable to data objects <u>without</u> side-effects and they fail to specify the behavior of data objects which are used in parallel computations (multi-process environments). Our specification techniques have overcome these limitations. Formal specifications for an air line reservation system and bounded buffers will be given as illustrations of our techniques.

(2) Conceptual Representations

We have developed notational devices called *conceptual representations* to describe the states of individual actors (objects, and data structures) at various levels of abstraction.

The use of conceptual representations reinforces the notion of data and procedural objects as abstract entities whose internal structures are hidden. By separating the states of an object from its identity, conceptual representations can express sharing among objects in an intuitive, yet rigorous manner. Thus our specification language with its use of conceptual representations has flexible and powerful expressiveness.

(3) Symbolic Evaluation of Programs written in Object-Oriented Languages

Symbolic evaluation is a process which abstractly executes programs on abstract data. As the major tool for program verification, we have developed a method for symbolic evaluation of programs written in SIMULA-like object-oriented languages. Our formalism based on conceptual representations enables us to deal with the difficulties due to object sharing which often arise in verification of programs written in object-oriented languages.

(4) Specifications of Systems with High Internal Concurrency and Task Specifications

Little work has been done on specifying and verifying the behavior of a system characterized by complex concurrent activities of its subsystems. Operating systems and multi-user data base systems fall into this category. In order to illustrate our techniques for dealing with such systems, we give a model of a simple post office where a number of customers and mail-collectors are represented as internal concurrent activities. We show that the specifications of the over-all functions of such a system, which we call *task specifications*, are derived from the specifications of the individual behavior and mutual interaction of its subsystems.

1.5 Outline of the Thesis

Chapter 2 introduces conceptual representations, which are extensively used in the work presented in this thesis. The precise syntax of conceptual representations and their uses in writing formal specifications of abstract data types without parallelism and side-effects are exemplified. Further, algebraic axiomatic and abstract model approaches to the specification of abstract data types are discussed in the light of our approach. (This chapter does not use the actor model of computation.)

Chapter 3 gives a precise account of the actor computation model on which the discussion in the subsequent chapters is based. It also describes certain characteristics of the behavior of actors which must be considered in the development of specification techniques.

Chapter 4 presents our specification techniques for serial computation. The separation of the identities of objects from their states is explained and how this is incorporated into our formalism is illustrated before our specification language is introduced with examples of formal specifications. Several other approaches to program specification are reviewed.

Chapter 5 describes our method of symbolic evaluation and illustrates our verification techniques for serial computations based on the symbolic evaluation method. The application of symbolic evaluation to other domains is also discussed.

Chapter 6 extends the specification language introduced in Chapter 4 to cover parallel computations and illustrates our techniques for writing formal specifications of abstract data types with *parallelism* and *side-effects*. The notion of local states of actors (objects) is discussed in detail in the beginning of the chapter.

Chapter 7 presents our verification techniques for parallel computations. The verifications of air line reservation systems and bounded buffers are illustrated.

Chapter 8 contains an actor model of a simple post office, which is an intuitive example of a system with high internal concurrency. We show that the internal activities of the post office meet its task specifications.

Chapter 9 makes the concluding remarks and suggests future research.

2. Conceptual Representations

Conceptual representations occupy the central role in the formal specification and verification techniques presented in this thesis. In this chapter, we will explain the basic idea of conceptual representations by illustrating how specifications of conventional data structures are written using conceptual representations. However, as will be seen in the later chapters, conceptual representations are used to describe states of actors of a wide variety. In the later part of this chapter, existing specification techniques for data structures (data types), such as algebraic axiomatic ones, and an abstract model approach, will be discussed in relation to the techniques based on conceptual representations.

2.1 Introduction

We will use conceptual representations to specify a wide range of data structures at various levels of abstraction. The motivation in developing conceptual representations is to provide a specification language which serves as a good interface between programmers and the computer and also between users and implementors. A "good" interface language should allow programmers to easily express and understand their intuitive concept of a data structure and how it behaves for various operations. For example, the "language" of diagrams using boxes and arrows is a very good language in which people can exchange their intuitive ideas about the sharing relationships among objects. However, such a language is not rigorous enough for the computer to understand without many hidden assumptions. The specification language based on conceptual representations introduced in this chapter is rigorous and yet able to express graphical intuitions about data structures.

Different degrees of awareness about the implementation of a data structure are required in the different activities of implementing a system such as the initial design, coding, and the subsequent evolution. Conceptual representations are flexible enough to express only the details which are important in each activity. As mentioned above, conceptual representations are not confined to specifying data structures. They are used to describe states of both procedural and data objects and also used to express views and summaries of behaviors of such objects. Examples of such conceptual representations will be found in the later chapters [e.g., Chapter 6 and Chapter 8].

2.2 Conceptualization of Data Structures

In this section, we explain syntactic constructs of conceptual representations using simple examples. The BNF syntax of conceptual representations is given in Figure 2.1 at the end of this section.

2.2.1 Keywords and C-packages

Let us consider a simple data structure, a cell, which contains information that can be retrieved and updated. In order to express a cell which has its contents, say 10, we use the following notation

(*CELL* (*contents:* 10)).

This is a conceptual representation of the cell. When this cell is updated with new contents, 12, its conceptual representation becomes

(*CELL* (*contents:* 12))

A word "*CELL*" in the above conceptual representations is an example of the *keywords* which express the conceptual types of data structures. The keywords are always spelled in italic capital letters.

In addition to keywords, another syntactic construct, *conceptual packages* (abbreviated as *c-packages*)[1] is used to express more detailed information about data structures. A notation "(*contents:*...)" in the conceptual representations for cells is an example of c-packages. C-packages are useful to distinguish conceptually different kinds of

---

[1] The syntax of c-packages are borrowed from that of packages in PLASMA [Hewitt-Smith75, Hewitt77]

components of a data structure. For example, a node in list structures of LISP has two conceptually different kinds of components, the car-part and the cdr-part. The following conceptual representation

(*NODE* (*car:* 10) (*cdr:* 12))

expresses a node whose car-part and cdr-part are 10 and 12, respectively. (*car:* 10) and (*cdr:* 12) are c-packages. Selectors of packages (e.g. *car* and *cdr*) are always spelled in the lower case italic letters followed by a colon.

When the details or specification of some components of a data structure are unimportant, but their existence in the data structure needs attention, question marks may be placed in conceptual representations. For example, when we want to express a node whose car-part is 13, but cdr-part may be anything,

(*NODE* (*car:* 13) (*cdr:* ?))

may be used. We call the question marks used in this way *dummy element notations*.

2.2.2 C-sequences

There are many data structures which are naturally viewed as a linear sequence of components at some levels of abstraction. Queues, stacks, arrays, tables and etc. are examples of such data structures. To express such conceptual sequences of components in data structures, we use a syntactic construct, *conceptual sequences* (abbreviated as *c-sequences*).[1]

---

[1] Specifications of *forms* in ALPHARD[Wulf-et-al76] are stated in terms of mathematical objects such as sequences and sets.

Let us consider queues to see how c-sequences are used. Programmers envisage a queue as a linear sequence of elements which are enqueued at one end and dequeued from the other end. Suppose that we have a queue consisting of three elements, 1, 2, and 3, where 1 is its front element and 3 is its rear element. Using a c-sequence [1 2 3], this queue is expressed by the following conceptual representation.

(*QUEUE* [1 2 3])

When a new element 4 is enqueued at the rear end of this queue, this queue is expressed as:

(*QUEUE* [1 2 3 4]).

2.2.3 Unpack Operations and Dot Notions

In order to express a queue which has an indefinite number (including zero) of elements, we use a c-sequence variable, say x, in conceptual representations as follows:

(*QUEUE* [!x])

!x is an abbreviation of the "unpack" operation on x.

In general, !<expression> is equivalent to writing out all of the elements of the c-sequence denoted by <expression> individually. For example, suppose that x denotes a c-sequence [2 3 4]. Then

[1 !x] = [1 ![2 3 4]] = [1 2 3 4]

whereas

[1 x] = [1 [2 3 4]] ≠ [1 2 3 4].

When y denotes an empty c-sequence [],

[1 !y] = [1 ![]] = [1].

Thus (*QUEUE* [!y]) is equivalent to (*QUEUE* []) which is the conceptual representation of an empty queue.

Let us look at more elaborate examples of conceptual representations of queues which use unpack operations and c-sequence variables. The two conceptual representations:

(*QUEUE* [6 !z]) and (*QUEUE* [!z 9])

express a queue whose front element is 6 followed by the elements of z and a queue whose last element is 9, respectively. Furthermore

(*QUEUE* [!x 8 !y])

expresses a queue which has 8 as one of its elements. When the elements before and after 8 (i.e. !x and !y) in the queue are uninteresting, the following conceptual representation may be used.

(*QUEUE* [... 8 ...])

"..." inside the c-sequence is called a *dot notation*. In general, dot notations are used to indicate only the existence of an indefinite number (including zero) of elements whose specification is not important in a c-sequence or c-collection. (Cf. 2.2.4) Dummy element notations may be used as elements of c-sequences. For example, a conceptual representation:

(*QUEUE* [? 3 4 5])

describes a queue whose front element is unknown (or unimportant), and the rest of whose elements are 3, 4 and 5, in this order.

2.2.4 C-collections

Another syntactic construct of conceptual representations is *conceptual collections* (abbreviated as *c-collections*) which are used to represent a conceptual group of components in data structures. C-collections are different from c-sequences in that the order of elements in c-collections is unimportant. For example, a c-collection {2 3 7} is equivalent to both {2 7 3} and {7 3 2}. C-collections are also different from mathematical sets in that multiple occurrences of the same elements in c-collections are important. For example, a c-collection {2 2 7} is not equivalent to {2 7}.

A simple example of conceptual representations using c-collections is

(*SET* {3 4 5})

which expresses a data structure of the type "set" whose elements are 3, 4, and 5. An indefinite number of elements of a c-collections can be expressed by the unpack operations and c-collection variables. Thus a general form of the conceptual representation for the data structure "set" may be expressed as

(*SET* {!x}).

C-collections may be described by using dummy element notations "?" and dot notations "..." in the same way as c-sequences.

2.2.5 Pattern Matching

Unpack operations are extremely useful in pattern matching[1] of c-sequences and c-collections. Below we will give basic examples of pattern matching, instead of presenting the matching algorithm.

Suppose that a c-sequence of four numbers [1 9 8 4] matches against the following patterns, where u, v, and w are pattern (or free) variables on c-sequences.

(1) [1 !u], u *must be* [9 8 4].
(2) [!v 8 4], v *must be* [1 9].
(3) [!w], w *must be* [1 9 8 4].
(4) [!u 8 !v], u *and* v *must be* [1 9] *and* [4], *respectively.*
(5) [1 9 8 4 !u], u *must be* [].

Suppose that the same c-sequence matches against the following patterns, where M and N are pattern (or free) variables on numbers.

(6) [M !u], M *and* u *must be* 1 *and* [9 8 4], *respectively.*
(7) [!u N], u *and* N *must be* [1 9 8] *and* 4, *respectively.*

But [1 9 8 4] does not match against the following pattern:

(8) [M N].

Some patterns may have more than one matching case. For example, when [1 9 8 4] matches against

---

[1] The use of pattern matching in our specification and verification techniques will be exemplified in the process of symbolic evaluation in Chapter 5.

(9) **[1 !u M !v],** ***there are three matching cases:***

Case-1: **u = [], M = 9, v = [8 4].**
Case-2: **u = [9], M = 8, v = [4].**
Case-3: **u = [9 8], M = 4, v = [].**

Fig. 2.1. Syntax of Conceptual Representations in BNF

<conceptual-representation> ::= (<keyword>) | (<keyword> <conceptual-constituents>)

<keyword> ::= *% a word in the upper case italic font %*

<conceptual-constituents> ::= <an-entity> | <c-sequence> | <c-collection> | <c-package-sequence>

<an-entity> ::= *% a single conceptual entity, which is often an actor %*

<c-sequence> ::= [<juxtaposition>]

<c-collection> ::= { <juxtaposition> }

<c-package-sequence> ::= <c-package> | <c-package> <c-package-sequence>

<c-package> ::= (<selector> <conceptual-constituents>)

<juxtaposition> ::= <element> | <element> <juxtaposition>

<selector> ::= *% an identifier in the lower case italic font followed by a colon. %*

<element> ::= <empty> | <an-entity> | <c-sequence> | <c-collection> |
        <c-package> | <unpacked-c-sequence> | <dot-notation> | <dummy-element-notation>

<empty> ::= *% an empty string %*

<unpacked-c-sequence> ::= !<c-sequence> | !<c-sequence-variable>

<dot-notation> ::= ...

<dummy-element-notation> ::= ?

<c-sequence-variable> ::= *% an identifier in the lower case roman font %*

2.3 Specifications of Data Structures

In this section, we exemplify how conceptual representations are used in specifications of data structures. An abstract data type [Liskov-Zilles74] or a data structure is specified by the functionality (domains and ranges) of the applicable operations and the effects of these operations. If the data structure may be created by users, how it is created must be also specified. In specifying functionalities, a notation "*error*" is used to denote a set of error messages which warn users of operations that an error has occurred. We assume that data structures are not changed by operations which cause error messages.

2.3.1 Queues

As suggested in the previous section, we use conceptual representations of the following form to express a queue.

(*QUEUE* [...])

A complete specification of queues is given in Figure 2.2.

Fig. 2.2. A Specification of Queues

FUNCTIONALITY:

i) CREATE-QUEUE: ---> *queue*

;creates an empty queue.

ii) ENQUEUE: *queue* x *item* ---> *queue*

;enqueues a new item at the rear end of the queue.

iii) DEQUEUE: *queue* ---> *item* x *queue* or *error*

;tries to dequeue the front element of the queue.

;if the queue is empty, an error message is produced.

iv) IS-EMPTY: *queue* ---> *boolean*

;checks whether or not the queue is empty.

EFFECTS:

(1) CREATE-QUEUE() ----> (*QUEUE* [])

(2) ENQUEUE((*QUEUE* [!x]), A) ----> (*QUEUE* [!x A])

(3) DEQUEUE((*QUEUE* [])) ----> *ERROR*

(4) DEQUEUE((*QUEUE* [A !x])) ----> <A , (*QUEUE* [!x])>

(5) IS-EMPTY((*QUEUE* [])) ----> *TRUE*

(6) IS-EMPTY((*QUEUE* [A !x])) ----> *FALSE*

2.3.2 Sets

A typical use of conceptual collections in conceptual representations is the data type "set". The following four operations are associated with the set type.

FUNCTIONALITY:

i) **CREATE-SET:** ---> *set*

;creates an empty set.

ii) **INSERT:** *element* x *set* ---> *set*

;tries to insert an element,
;if the element is already in the set, no effect.

iii) **DELETE:** *element* x *set* ---> *set* or *error*

;tries to delete an element from a set.
;if the element is not in the set, error.

iv) **IN?:** *element* x *set* ---> *boolean*

;checks whether or not an element is a member of a set.

The effects of these operations are formally described in Figure 2.3. Note that the membership of an element in a set is expressed succinctly by dot notations in c-collections.

Fig. 2.3. A Specification of Sets

EFFECTS:

```
(1) CREATE-SET()  ---->  (SET {})

(2) INSERT(E, (SET {!x}))

        if  x = { ... E ... }      ---->  (SET {!x})
        if  x ≠ { ... E ... }      ---->  (SET {!x E})

(3) DELETE(E, (SET {!x}))

        if x = {!y E !z}      ---->    (SET {!y !z})
        if x ≠ { ... E ... }  ---->    ERROR

(4) IN?(E, (SET {!x}))

        if  x = { ... E ... }      ---->  TRUE
        if  x ≠ { ... E ... }      ---->   FALSE
```

2.3.3 Arrays

The following five operations are associated with the array type.

FUNCTIONALITY:

i) CREATE-ARRAY: *integer* x *integer* ---> *array* or *error*

;tries to create an empty array with the specified lower and upper bounds.

;the first integer should not be greater than the second integer.

ii) STORE: *array* x *integer* x *item* ---> *array* or *error*

;tries to store an item with the specified index

;the index should be within the bounds.

iii) FETCH: *array* x *integer* ---> *item* or *error*

;tries to fetch an item with the specified index

;the index should be within the bounds.

iv) BOTTOM: *array* ---> *integer*

;returns the lower bound.

v) TOP: *array* ---> *integer*

;returns the upper bound.

To express arrays, we use conceptual representations of the following form:

(*ARRAY* (*low:* l) (*high:* h) (*elements:* {...[i A]...}))

where l and h are the lower and upper bounds, respectively, and an item A with the index i is expressed as a c-sequence [i A] in the c-collection of the (*elements:*) c-package. The effects of the operations applicable to an array is given in Figure 2.4.

Multi-dimensional arrays can be expressed easily by modifying c-sequences in

Fig. 2.4. A Specification of Arrays

```
EFFECTS:

(1)   CREATE-ARRAY(l, h)

   if  l ≤ h,   --->  (ARRAY (low: l) (high: h) (elements: {}))
   if  l > h,   --->  ERROR                                          ;bound error.

(2)  STORE((ARRAY (low: l) (high: h) (elements: {!x})), i, A)

   if  i > h or  i < l,   --->  ERROR                                ;bound error.
   if  l ≤ i ≤ h  and  x = {!e1 [i ?] !e2}        ;when the i-th element already exists.
                         --->    (ARRAY (low: l) (high: h) (elements: {!e1 [i A] !e2}))
   if  l ≤ i ≤ h  and   x ≠ { ... [i ?] ... }      ;when the i-th element does not exist.
                         --->    (ARRAY (low: l) (high: h) (elements: {!x [i A]}))

(3)  FETCH((ARRAY (low: l) (high: h) (elements: {!x})), i)

   if  i > h or  i < l,   --->  ERROR                                ;bound error.
   if  l ≤ i ≤ h  and  x = { ... [i B] ... }   --->   B
   if  l ≤ i ≤ h  and   x ≠ { ... [i ?] ... }   --->   ERROR   ;when the i-th element is not found.

(4)  BOTTOM((ARRAY (low: l) (high: h) (elements: {...})))   --->  l

(5)  TOP((ARRAY (low: l) (high: h) (elements: {...})))   --->   h
```

the (*elements:*) c-package to include more than one index. For example, a two-dimensional array may be expressed by a conceptual representation of the following form

(*ARRAY* (*low:* l) (*high:* h) (*elements:* { ... [i j A] ... }))

2.3.4 Symbol Tables

As an example of specifications for more complicated data structures, we give a specification of symbol tables [Guttag75, London-et-al76]. Symbol tables are often used in writing compilers for programming languages which have ALGOL-like block structures. A symbol table records pairs of an identifier and its attribute. The same identifier may have different attributes depending upon where the identifier is used in the block structure. We assume the following six operations are applicable to a symbol table. No operations except ENTER-BLOCK are allowed before the most global block is entered. The creation of a symbol table does not imply the entering of the most global block.

FUNCTIONALITY:

i) CREATE-SYMBOL-TABLE: ----> *symbol-table*

;creates an empty symbol table.

;no block has been entered yet.

ii) ENTER-BLOCK: *symbol-table* ----> *symbol-table*

;set up a new local naming scope.

iii) LEAVE-BLOCK: *symbol-table* ----> *symbol-table* or *error*

;tries to leave the current block.

;if the current block is outside the most global one, then error.

;otherwise discard the current block and reactivate the most previous scope.

iv) ADD: *symbol-table* x *id* x *attribute* ----> *symbol-table* or *error*

;tries to add a pair of an identifier and its attribute.
;if the current scope is outside the most global block, then error.
;if the identifier is already declared in the current block, then error.

v) RETRIEVE: *symbol-table* x *id* ----> *attribute* or *error*

;tries to retrieve the attribute of an identifier in the most recent
;block in which the identifier is declared.
;if it is not found, then error.

As a conceptual representation for the symbol table, we use the following notation:

(*SYMBOL-TABLE* [§x]).

x is a c-sequence whose elements are empty or c-packages of the form

(*block:* [§y])

which conceptually represents a block. The order of c-packages in x corresponds to the order of blocks. That is, the last c-package in x corresponds to the most recently entered block. y is a c-sequence whose elements are pairs of an identifier and its attribute. Such pairs are expressed by a c-sequence. For example, suppose that in some block identifiers A and B are declared to be real and integer, respectively. Then the conceptual representation for this symbol table looks like:

(*SYMBOL-TABLE* [...(*block:* [... [A real] ... [B integer] ...]) ...]).

Using conceptual representations of this form, a specification of symbol tables is written as depicted in Figure 2.5.

Fig. 2.5. A Specification of Symbol Tables

EFFECTS:

(1) CREATE-SYMBOL-TABLE( ) ---> (*SYMBOL-TABLE* [])

(2) ENTER-BLOCK((*SYMBOL-TABLE* [!u])) ---> (*SYMBOL-TABLE* [!u (*block:* [])])

(3) LEAVE-BLOCK((*SYMBOL-TABLE* [])) ---> *ERROR*

;leaving the most global block (without entering).

(4) LEAVE-BLOCK((*SYMBOL-TABLE* [!w (*block:* [...])])) ---> (*SYMBOL-TABLE* [!w])

(5) ADD((*SYMBOL-TABLE* []), ID, ATT) ---> *ERROR*

;adding an id-attribute pair without entering the most global block

(6) ADD((*SYMBOL-TABLE* [!r (*block:* [!pairs])]), ID, ATT)

if pairs = [... [ID ?] ...] ----> *ERROR* ;ID is already declared in the current block.

if pairs ≠ [... [ID ?] ...]

---> (*SYMBOL-TABLE* [!r (*block:* [!pairs [ID ATT]])])

(7) RETRIEVE((*SYMBOL-TABLE* [!t]), ID)

if t ≠ [...(*block:* [...[ID ?]...])...] ---> *ERROR*

;the identifier is not found in any blocks.

if t = [...(*block:* [...[ID ATT] !x]) !y] *and* y ≠ [...(*block:* [...[ID ?]...])...] ---> ATT

2.4 Relationship to Other Work

In this section, we discuss the relationship of our specification techniques for data structures presented in this chapter[1] to some other work in the same area. We have chosen to consider an algebraic axiomatic approach and an abstract model approach because these two approaches are in clear contrast to ours and also well studied. An excellent survey of specification techniques for abstract data types is found in [Liskov-Zilles75]. Other approaches such as Parnas's "state machine model" [Parnas72] are also reviewed in [Liskov-Zilles75].

2.4.1 Algebraic Axiomatic Approach

Algebraic axiomatic techniques have been studied by a number of researchers [Zilles74, Spitzen-Wegbreit75, Guttag75, Wegbreit-Spitzen76]. In this approach, the effects of operations on an object of the data type being specified are expressed in terms of equations of the operations. To compare their approach with ours, we present two algebraic axiomatic specifications, one for queues (which is a modified version of [Spitzen-Wegbreit75]) in Figure 2.6 and the other for symbol tables (which a slightly simplified version of [Guttag75]) in Figure 2.7.

All the axioms given in their specifications in Figure 2.6 and Figure 2.7 are easily derived from our specifications of queues in Figure 2.2 and symbol tables in Figure 2.5. [For the derivation of the axiom (5) in Figure 2.6, see Appendix I.] We believe that specifications using conceptual representations are often easier for programmers to both

---

[1] In this chapter, we assume that data structures or data types are always used in serial computations. Our techniques for data structures (or abstract data types) with *parallelism* and *side-effects* will be presented in the later chapters.

Fig. 2.6. An Algebraic Axiomatic Specification of Queues

FUNCTIONALITY: omitted.

AXIOMS:

(1) IS-EMPTY(CREATE-QUEUE()) = *TRUE*

(2) IS-EMPTY(ENQUEUE(Q, A)) = *FALSE*

(3) DEQUEUE(CREATE-QUEUE()) = *ERROR* *;attempts to dequeue an empty queue.*

(4) *if* IS-EMPTY(Q) *then* DEQUEUE(ENQUEUE(Q, A)) = <A, Q>

(5) *if* ¬IS-EMPTY(Q) ∧ DEQUEUE(Q) = <B, Q'>
then DEQUEUE(ENQUEUE(Q, A)) = <B, ENQUEUE(Q', A)>

---

construct and understand than algebraic axiomatic specifications, because in the conceptual representation approach we describe directly and explicitly what effects take place in data structures (at the conceptual level) when the operations are applied, whereas the algebraic axiomatic specifications describe the effects of the operations indirectly and implicitly in terms of relations (or equations) among the operations. In particular, the axiom (6) for symbol tables in Figure 2.7 is expressed in terms of a recursion of RETRIEVE. Such indirect specifications are often difficult to grasp. Thus the author and reader of an algebraic axiomatic specification of a data type may be less confident as to whether or not the specification completely describes the desired properties of the data type.

Recently a serious problem in the algebraic approach has been pointed out[Majster77]. The problem is that there are some classes of abstract data types which cannot be specified by a finite set of axioms for the operations (equations of the

Fig. 2.7. An Algebraic Axiomatic Specification of Symbol Tables

FUNCTIONALITY: omitted.

AXIOMS:

(1) LEAVE-BLOCK(CREATE-SYMBOL-TABLE()) = ERROR

(2) LEAVE-BLOCK(ENTER-BLOCK(symtab)) = symtab

(3) LEAVE-BLOCK(ADD(symtab, id, attrs)) = LEAVE-BLOCK(symtab)

(4) RETRIEVE(CREATE-SYMBOL-TABLE(), id) = ERROR

(5) RETRIEVE(ENTER-BLOCK(symtab), id) = RETRIEVE(symtab, id)

(6) RETRIEVE(ADD(symtab, id, attrs), id1)
    if id = id1,
        then attrs
        else RETRIEVE(symtab, id1)

---

operations). To avoid this problem, they must use *axiom schemata* instead of infinitely many axioms. This violates the finiteness of the axiom set which is an important assumption of the underlying theory for algebraic specification techniques. Our conceptual representation approach does not have such a problem, because, as mentioned above, our techniques describe explicitly what effects the operations cause to data structures. (In appendix II, a data type which cannot be expressed by a finite set of algebraic axioms of operations is specified by using conceptual representations.)

Furthermore, the current algebraic and axiomatic techniques do not capture an important difference between data structures without side-effects and data structures with side-effects. (This difference will be explained in Chapter 3.) As will be seen in Chapter 4, the specification technique using conceptual representations can easily express this difference. For further discussions on the algebraic approach, see Section 4.5.5, Chapter 4.

2.4.2 Abstract Model Approach

B. Liskov and V. Berzins [Liskov-Berzins77] have been developing an abstract model approach in the area of specification of abstract data types. The construction of its mathematical foundation is underway [Berzins76]. In this approach, first a certain set of well established data types and mathematical objects (e.g., sets, sequences, tuples and etc.) is chosen. Then new abstract data types are specified in terms of such chosen data types or already defined abstract data types.

As an example, we give an abstract model specification of arrays cited from [Liskov-Berzins77] in Figure 2.8. Objects of the type *array[t]* are represented by the following tuple:

```
tuple[low: integer,
      high: integer,
      elements: sequence[tuple[index: integer, value: t]]]
```

Comparing the specification in Figure 2.8 with the one given in Figure 2.4 which is based on the conceptual representations, one is struck by the similarity. In fact, in representing objects of a new data type, the roles of sequence, sets and tuples in their approach correspond to those of c-sequences, c-collections and c-packages in our approach. However, in the abstract model approach, the operations applicable to objects of a new data type are

Fig. 2.8. An Abstract Model Specification of Arrays

FUNCTIONALITY: omitted.

OPERATIONS:

```
alloc(i1, i2) =         if i1 ≤ i2 then {low: i1, high: i2, elements: <>}
                                else error("bad array size")
                                        ;<> denotes an empty sequence and {...} denotes a tuple.

bottom(a) =                     a.low

top(a) =                        a.high

store(x, i, a) =        if a.low ≤ i ≤ a.high
                                then { low: a.low
                                        high: a.high
                                        elements: addfirst({index: i, value: x}, a.elements) }

fetch(i, a) =                   if a.low ≤ i ≤ a.high then getval(a.elements, i)
                                     else error("index out of bounds")

getval(elements, i) =           if length(elements) > 0 then
                                             if $elements_1$.index = i then $elements_1$.value
                                             else getval(butfirst(elements), i)
                                else UNDEFINED
                             ;$elements_1$ means the first item of the sequence denoted by "elements"
```

specificed in terms of *procedures* defined on pre-defined data types. Getval, addfirst, and butfirst in Figure 2.8 are examples of such procedures. In the conceptual representation approach, we do not use such procedures in specifying the effects of the operations. Instead, we rely on *pattern mechanisms* of keywords, c-sequences, c-collections and c-packages, which have been exemplified by a number of specifications.

As was pointed out in the previous subsection, our approach is extended easily to cover data structures with side-effects. The extendability of the abstract model approach remains to be seen.

3. Behaviors of Actors (A Model of Computation)

In this chapter, we introduce the model of *deterministic* computation on which the discussion in the rest of this thesis is based. The first section mainly contains definitions and intuitive accounts of various concepts and notations employed in the model of computation. The second section describes the characteristics which must be considered in trying to construct formal specifications of computations in the model. This section contains the classification of interactions among actors.

3.1 The Actor Model of Computation

The fundamental objects in our model of computation are *actors*. Computations are carried out through message passing between actors. An actor is a potentially active object (procedure) which becomes active when it receives a message. Each actor decides itself how to respond to messages sent to it. No actor can treat other actors as objects to operate on: it can only send a message to other actors.[1]

Messages are also actors. An actor may be created in the course of computation or may exist in the beginning of a computation. More than one transmission of a message at a time in an actor system may take place.

A collection of actors which communicate and cooperate with each other in a goal oriented fashion can be implemented as a single actor. A system of actors can model various kinds of information processing schemata from ordinary sequential arithmetic or symbolic computations to highly distributed parallel computations including computer networks of varying scales. Furthermore, it can model problem solving activities by a society of experts[Hewitt77].

A number of concepts in programming systems can be captured by simple concepts in the actor model of computation. For example, traditionally different kinds of entities such as data, data structures, files, and procedures are unified as a single kind of object, the actor. Control structures such as recursion, iteration, and coroutines can be viewed as particular patterns of message passing [Hewitt77]. Furthermore, calling a procedure, returning a value, retrieving and updating data structures, and synchronization and communication of cooperative parallel processes are achieved by message passing.

---

[1] For example, to get the i-th element of an array A, an *(i-th:)* message is sent to A instead of doing a fetch operation A[i].

An implementation of the actor model of computation has been realized as a programming language PLASMA[Hewitt-Smith75, Hewitt77]. The syntax of PLASMA is so designed that its underlying semantics is transparent.

The above intuitive account of the model of computation will be made more precise below.

3.1.1 Actors

An actor consists of two parts, *script (action)* and *acquaintances*. Its script is a description of how it should respond to messages sent to it. Each actor has a fixed set of messages by which it can be activated. When a message that does not belong to this set is sent to an actor, the response of the actor is undefined. The acquaintances of an actor are a finite collection of actors that it directly *knows about*. An actor A can send a message directly to an actor B only when B belongs to the acquaintances of A. The script of an actor can be realized by a PLASMA program for the actor. The acquaintances of a newly created actor C are the set of actors which are denoted by free identifiers in the PLASMA program for C at the time of creation.

3.1.2 Events

An event E is defined to be the *receipt* of a *message*[1] actor M by a *target* actor T. The event E is expressed by a notation of the form

---

[1] We use the terms "receipt" and "arrival" of a message interchangeably throughout the thesis.

⟦ T <-- M ⟧.

A message contains a *request* of what the target actor is asked to do and it may also contains a *continuation* actor which is the destination where the reply to the request is supposed to be sent. Messages are often expressed by notations of the form

[*request:* <request> *reply-to:* <continuation>].

The request usually consists of a tag which indicates a task to do and the data necessary to accomplish the task. PLASMA packages are often used as requests. For example, to request a queue-actor to enqueue some actor A at the end of the queue, the package (***enqueue:*** A) is used. To request a queue-actor to send back its front element, the package (***dequeue:***) is used. The continuation actor may be omitted in the message when it is unnecessary. For example, when the purpose of a message is to return the result of a task, or the reply to a request, the message need not contain a **<continuation>**. In such cases, messages are expressed by notations of the form

[*reply:* <result>]

When a continuation C in a message is unimportant or obvious from the context of discussion, we make only the request part explicit in expressing an event. So the following abbreviated form is used

⟦T <= <request>⟧ for ⟦T <== [*request:* <request> *reply-to:* C]⟧.

Furthermore, when it is obvious from the context that a message contains only a replying result, we use the following abbreviated form.

[T <= <result>] for [T <== [*reply:* <result>]].

Note that the above abbreviated forms use single shafted arrows "<=" instead of double shafted arrows "<==". In the subsequent presentation of this thesis, the terms "request" and "message" will be used interchangeably when we are not interested in the continuation in a message.

A *primitive event* is an event which activates exactly one immediate reply without causing any intermediate events. From this definition, we can define primitive actors. A *primitive actor* is an actor which always causes a primitive event when it is sent a message.

As we have noted above, different control structures in programming languages are viewed as different patterns of message passing in the actor model of computation. In fact, such different patterns of message passing correspond to different patterns of continuation in messages. The patterns of continuation for recursion, and iteration are found in [Hewitt77] and for coroutines in [Greif-Hewitt75]. The fact that continuations are sent together with requests allows the unification of control flow and data flow into a universal flow of information, message transmission. Consequently, this unification allows us to describe computations solely in terms of events.

3.1.3 Computations

A computation is defined as a *partially* ordered set of events, where the ordering is strict and transitive. A physical intuition for the ordering is that an event E *precedes* another event E'. We call this ordering the *precedes order* and denote it by "-->". Then a computation is a pair <Ev, "-->"> where Ev is a set of events. The strictness of the ordering imposes the constraint that any event E does not precedes itself:

$$\forall E, \neg(E \rightarrow E).$$

The partialness of the ordering allows that some events E_i and E_j do not precede each other, which means that E_i and E_j may take place concurrently. We assume that each such ordered set of events always has the maximal events in it. This means that every computation has a set of initial events.

Our assumption to model <u>physically</u> <u>realizable</u> <u>computations</u> requires two kinds of *finiteness* properties. First, for any two events E_i and E_j which are ordered by "-->", only finite numbers of event can take place between E_i and E_j. I.e., the set $\{E | E_i \rightarrow E \rightarrow E_j\}$ is finite. Second, each event E has finitely many immediate successor events. These two finite properties do not rule out non-terminating computations: they only exclude infinitely fast computations.[1]

The precedes ordering has two suborderings which reflect more detailed physical properties of computations. Suppose that E is an event in which a target actor T receives a message actor M. Then the event E triggers a response (or action). This response is a finite set C of events. We can view that the event E activates the events in C. Thus we call this type of ordering the *activation ordering* and denote it by "-act->". So $\forall$ EE in C, E -act-> EE. The activation ordering is intended to describe the notion of causality in computations.

Suppose that more than one message is sent to a single actor A in a computation. In our computation model we assume that one message arrives before another. I.e., no two messages arrive at the same actor simultaneously. Since each arrival of a message at A is an event by definition, if we fix a target actor, we can always introduce an ordering among

---

[1] Hewitt and Baker gave an proof for the impossibility of such infinitely fast computations in [Hewitt-Baker77].

events which have A as a target actor by *arrival time*. We call this ordering the *arrival ordering with respect to* A and denote it by "-arr->_A".

The important property of the arrival ordering is that it is a *total* order: each event in a computation can have at most one immediate successor event in terms of the arrival ordering, whereas it may have more than one immediate successor event in terms of the activation ordering.

A *nested activity* is a computation starting with a request event RQ of the form

[T <== [*request:* <request> *reply-to:* <continuation>]**]**

and ending with the corresponding reply event RP

[<continuation> <== [*reply:* <the-result>]**]**

The set of events consisting of the nested activity is the set:

$$\{E \mid E = RQ \vee E = RP \vee (RP \dashrightarrow E \wedge E \dashrightarrow RQ)\}$$

When a continuation is not contained in the message, the nested activity is undefined.

There are many activities in operating systems and distributed computing systems that are not nested. It should be pointed out that one may find many non-nested activities in the real world. The model of a post office given in Chapter 8 is an example of such non-nested activities.

3.1.4 Level of Detail

The behavior of an actor system can be described at varying levels of detail. Computing the factorial of 3 can be viewed as a process to input 3 and to output 6 at some level of detail. At this level of detail, an iterative way and a recursive way of computing the factorial of 3 are viewed as the same computation. Some difference between two implementations of the iterative factorial may be detected at some finer level of details. There may be many implementations of an actor which satisfy a given specification. Such implementations are viewed as the same implementation at one level and different ones at another level. At a finer level, some computations which may be viewed as a serial computation at a less fine level may be revealed to be parallel computations.

In order to describe the behavior of an actor system we need to choose a level of detail according to the purpose of description. The description of the behavior of an actor system at the lowest level of detail is given as a computation <Ev_0, "-->"> where Ev_0 is a set of all events which take place in the actor system. A level of detail is decided by criteria with which a subset Ev is chosen from Ev_0. Since any events E_i and E_j in Ev are also in Ev_0, if E_i and E_j are ordered by "-->" in Ev_0, the same order relation holds in Ev. Thus a partially ordered set of events Ev is a "sub"-computation of Ev_0. Choosing a subset from Ev_0 is done with various criteria which are decided by the purpose of description. For example, first we select actors from the set of all actors in the system, and then all events where the selected actors are involved as targets or messages are chosen from Ev_0. Another example of the criteria is to select events which meet some patterns such as the beginning and ending events of nested activities.

The notion of primitiveness defined in the previous subsection is relative to the level of detail chosen. The event where the factorial actor receives 3 is primitive at the level of detail where no events taking place before the arrival of 6 at the continuation are counted. An event where a data base receives a query can be viewed as a primitive event at a very high level of detail. Thus a data base can be considered as a primitive actor at that level.

3.2 Time Variant Behaviors of Actors

In this section we discuss the characteristics of individual actors which must be taken into account in formally specifying the behavior of an actor system.

3.2.1 Pure Actors and Impure Actors

All actors are classified into two categories depending upon their behavior. Actors which belong to one category never change their behavior. They always give the same reply to the same request. They are called *pure actors*. Actors which belong to the other category are called *impure actors* and their behavior may change with time. They do not always give the same reply to the same request. The following more precise definitions are given in terms of nested activities.[1]

> An actor T is *pure* if, for the same message M, the event [T <== M] always causes (precedes) the same reply event.

---

[1] The definitions can be viewed as behavioral definitions of immutable and mutable objects.

An actor T is *impure* (not pure) if there is a message M such that the event ⟦T <== M⟧ does not always cause (precede) the same reply event.

The "sameness" in the above definitions is used in the following sense: two actors are the "same" if they are *behaviorly equivalent*.[1] Two events are the same if they have the same target actor and the same message actor.

From this definition, it can be said that a pure actor behaves like a *mathematical* function. An actor which generates random numbers is impure because it returns a random number in response to the same message (***next-random-number:***). A *cell-actor* is another example of a simple impure actor. A cell-actor accepts a message (***update:*** **<new-content>**) which updates its contents and a message (***contents:***) which retrieves its contents. A cell-actor may change its behavior because it can give different answers to the (***contents?***) message, depending upon what it contains at the moment. An actor which behaves like a function + is a pure actor. The plus-actor always returns the same number, which is the sum of two numbers sent to it. Another example of a pure actor is a sequence-actor. One can retrieve elements of a sequence-actor, but one cannot change its elements; instead a completely new sequence-actor must be created. So a sequence-actor is pure.

3.2.2 Pure Queues and Impure Queues

To illustrate the difference between pure actors and impure actors, let us consider a pure actor and an impure actor, both of which behave like a queue. Both pure queue-actors and impure queue-actors accept the same two kinds of messages: one is (***nq:*** x)

---

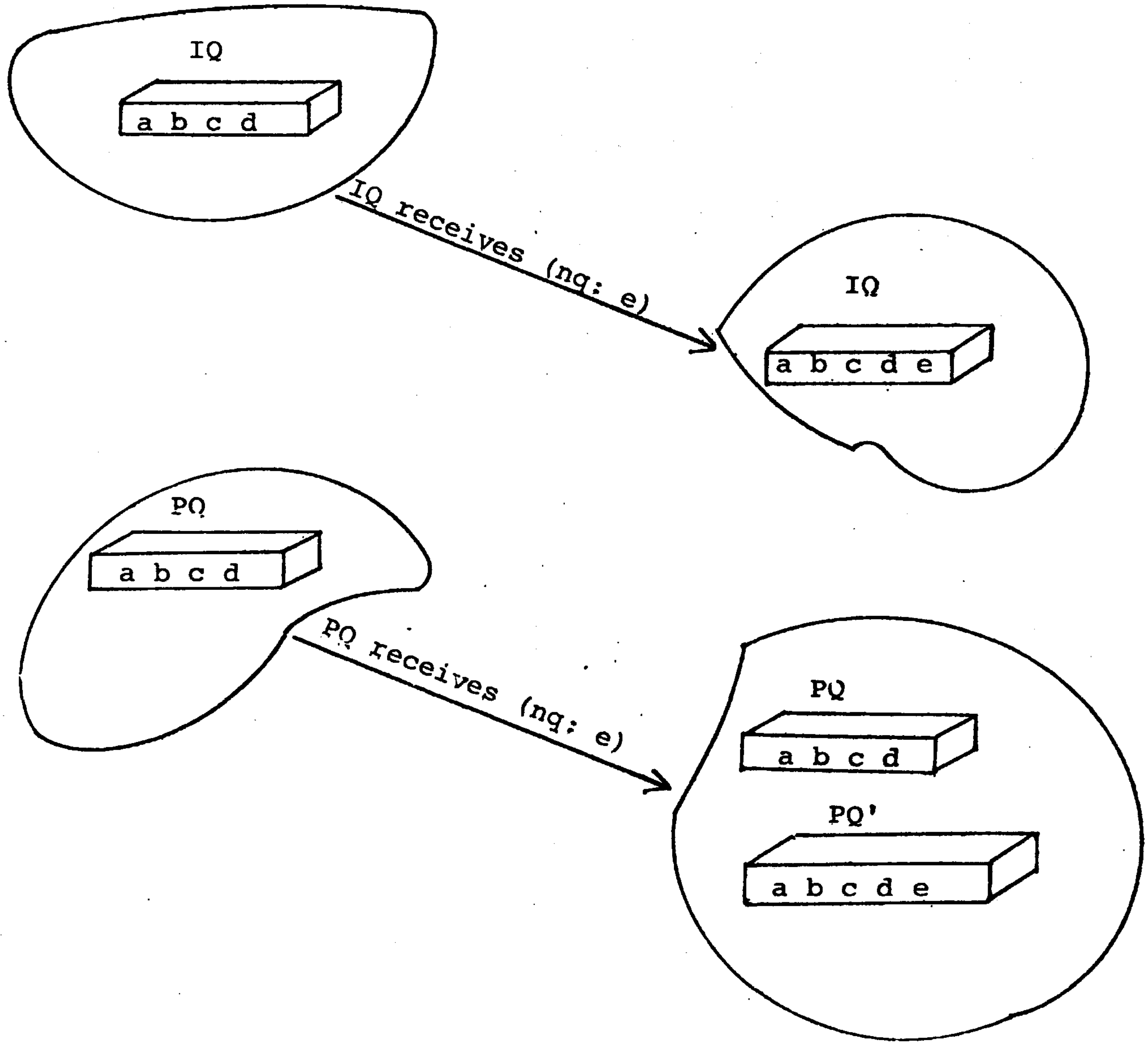
[1] For example, number actors which behave like 1 are behaviorly equivalent each other, but their identity may be distinct. The LISP functions, EQ and EQUAL, are impure and pure, respectively.

which is a request to enqueue a new elements x, and the other is (*dq:*) which is a request to take out the front element of the queue and return it with the remaining queue. However if the queue is empty, it returns a complaint message (*exhausted:*). The important difference between a pure queue-actor and an impure queue-actor is whether or not a new queue-actor is created when (*nq:* ...) and (*dq:*) are sent. When (*nq:* x) is sent to a pure queue-actor PQ, a new pure queue-actor PQ' which has x as the last element of the queue is created and returned. The original queue-actor PQ still has the same elements as before. When (*nq:* x) is sent to an impure queue-actor IQ, x is absorbed inside IQ and enqueued at rear of the previous elements. So IQ itself is extended and returned. No new queue-actors are created. (See Figure 3.1.)

When (*dq:*) is sent to a pure queue-actor PQ (which is not empty), a new pure queue-actor PQ' whose elements are all elements of PQ except the front element of PQ is created and the front element of PQ and the new pure queue-actor PQ' are returned. Again the original PQ is intact and has the same elements as before. When (*dq:*) is sent to IQ (which is not empty), then the front element of IQ and IQ itself which now has the rest of the original elements are returned. No queue-actors are created.

It might be helpful to see a LISP analogy in understanding this difference between pure queues and impure queues. Suppose that a queue is implemented as a list L. Then sending (*nq:* x) to a *pure* queue-actor corresponds to (**append** L (**list** x)), whose result is a totally new list constructed from a copy of L and x. Sending (*nq:* x) to an *impure* queue-actor corresponds to (**nconc** L (**list** x)) whose result does not consist of a copy of L.

Fig. 3.1. Behaviors of pure queues and impure queues



3.2.3 Sources of Impurity and Uses of Impurity

The change of behavior of an actor A is caused by the change of information used in computing the reply for a request to A. The change of such information is caused by the computation taking place before the reply event occurs.

Roughly speaking, the sources which may change the behavior of an actor A can be divided into two kinds. One is the activation of A initiated by messages which have been sent to A. The previous activations of A change the information stored inside A. For example, a random number generator usually keeps some internal values used to generate a random number. For the generation of the next random number, such internal values are changed during the generation of the previous random number. In the case of impure queue-actors, the history of the previous enqueuing and dequeuing operations determines the reply for the current dequeuing request.

The other kind of source is the computation initiated by messages which are *not* sent to A, but to some other actor B. When the computation initiated by a message sent to B changes information *shared* by both A and B, the subsequent behavior of A may change. Sharing of information sometimes happens inadvertently. When an actor A is created, some internal constituents of A might become known to other actors outside. For example, suppose that a new array-actor A is created by extending the upper bound of an existing array-actor B. If B receives a request to change one of its elements, the computation initiated by the request will change the subsequent behavior of A, because all elements of B are shared by A. There is another way in which internal constituents of an actor A become accessible. After an activation of A, the some internal constituents might be released outside as a result of the activation. Such released constituents become directly accessible from outside and information stored in them could be changed without sending legitimate requests to A.

Uses of an impure queue-actor are "destructive" in the sense that each enqueuing or dequeuing messages sent the actor changes the current status of the queue. If one wants to update the queue and still keep the previous status of the queue, the behavior of pure queue-actors is desirable even if it is costly in terms of both space and time. Sometimes the impurity of actors are necessary. For example, in order for concurrently running processes to communicate with each other, they need some actor which behaves as information storage through which they may exchange information. Such information storage may be contained inside each process or external common storage to which concurrent processes have access. This kind of impurity of actors is indispensable for communicating parallel processes.

3.2.4 Four Types of Interactions between Actors

Suppose that an actor M is sent as the request part of a message to a target actor T. This event initiates a computation where M and T are involved [i.e. an *interaction* between M and T]. After this computation, there will be no changes in the behavior of M or T if both M and T are pure actors. If M or T, however, are impure actors, the subsequent behavior of M or T may be different. Interactions between two actors M and T are classified into four types, depending upon the presence or absence of change in their future behavior.

No-Change-Type: Neither M nor T change their behavior.

The interactions initiated by the following events:

[factorial <= 3]
[create-array <= 4]

[**merge** <= [ARRAY-1 ARRAY-2]]

are examples of this type. The objective of this type of interaction is creation of new actors. Neither the factorial actor nor the number-actor 3 change their behavior, but the result of the computation, a number-actor 6, is created and returned. The create-array actor always creates an array of the size specified by the request message. The merge actor creates a new sorted arrays whose elements are those of the two sorted arrays ARRAY-1 and ARRAY-2. In this case, neither ARRAY-1 nor ARRAY-2 do not change.

Target-Change-Type: T changes its behavior, but M does not.

This type of interaction often takes places to modify information stored in actors which behave like data structures. For example,

[CELL <= (*update:* A)]
[IMPURE-QUEUE <= (*enqueue:* B)]

are of this type. The behavior of A or B do not change after the interactions.

Message-Change-Type: M changes its behavior, but T does not.

Examples of this type of interaction are initiated by events such as:

[**sort** <= ARRAY]
[**empty** <= IMPURE-QUEUE].

When an array-actor ARRAY is sent to the sort actor, the same array-actor ARRAY whose elements are sorted is returned. In a similar way, IMPURE-QUEUE is emptied but the empty actor does not change its behavior.

Target-Message-Change-Type: Both M and T change their behavior.

Examples of this type of interaction are often found in situations where some information is removed from one actor and transfered to another. In Chapter 8, we will model the activities in a simple post office in terms of actors. The interaction among customer actors, collector actors, and a mail box actor in the model is of this type.

4. Specifying Serial Computations

In this chapter, our specification techniques for serial computations are presented. Since our model is so constructed that serial computation is naturally extended to parallel computation, most of the concepts, notations, conventions and techniques introduced in this chapter are not only valid but also necessary for the specification and verification of parallel computations. In the first section, we introduce basic tools for describing the time variant behavior of actors. In the second section, we briefly discuss the role of conceptual representations in our model of computation. In the third section, our specification language for serial computations is explained and some issues of specifications related to "side-effects" are discussed. In the fourth section, examples of specifications written in our language are given.

4.1 Capturing Time Variant Behavior of Actors

In order for a formal specification language to be effective for our model of computation, it must be able to describe the time variant behavior of actors. The ability of our specification language to deal with this aspect of actor behavior is based on concepts introduced in this section.

4.1.1 History of Messages and States of Actors

As we have seen in the previous chapter, one source of the time variant behavior of an actor is the history of computations initiated by messages sent to the actor. If the whole past history of messages sent to an actor A is known, the subsequent behavior of A in response to a given message should be predictable. Thus, it is desirable to know the history of messages to specify the behavior of A. However, it is not practical to enumerate all possible histories of messages. Two actors with different past histories (sequences) of incoming messages sometimes show the same subsequent behavior. Thus we can partition the set of histories (sequences) of messages sent to A into equivalence classes according to the subsequent behavior of A. By such equivalence classes, we can define the notion of *states* of an actor. That is, the state of an actor A at a given point in time is defined as equivalence classes on the past histories (sequences) of messages sent to A. If A is in the same state at a different time, the subsequent behavior of A will be always the same.

The state of an actor which behaves as an information storer is often defined by the contents of the stored information. For example, the state of a cell-actor C at a time is defined by the contents B of the cell. This definition of states is a special case of our definition by equivalence classes on past message histories. For the contents of the cell can be viewed as the most recent update message (***update:*** B). The (***update:*** B) message represents the class of histories (sequences) of messages sent to C which have an (*update:* B) as the most recent update message.

Some kinds of states are not naturally expressed by the contents of stored information. For example, states of a data base which is being accessed by a number of concurrent processes are not expressed naturally by some stored information in the data base. The states where processes are updating or retrieving information in the data base may be expressed as certain monitoring mechanisms attached to the data base, but such mechanisms are dependent on the implementation of the data base. When the states of a data base are defined externally [i.e. independently of implementation], our definition of states is quite useful. The state of an air line reservation system discussed in Chapter 6 and that of a post office in Chapter 8 are examples of states of actors which are accessed by concurrent processes.

Equivalence relations which determine states (i.e. equivalence classes) are chosen according to the purposes and level of the detail of the specification. States which are different at some levels of the detail of the specification may be the same at other levels.

In Section 6.4, Chapter 6, we will discuss an alternative way of defining states of actors by continuous functions.

4.1.2 Situations

To incorporate the notion of states into the formalism for specification and verification, we need a notion of *situations*. A situation is the <u>local</u> state of an actor system at an instance of the local time.[1] A notion of situations which assumes the global state and global time reference has been proposed in the area of Artificial

---

[1] We will discuss the local time in detail in Section 6.1.2, Chapter 6.

Intelligence[McCarthy-Hayes69, Hewitt75]. Our model of computation allows parallelism which is realized by concurrent message passing. Since instances of concurrent message passing (i.e. events) may take place totally independently, it is quite unnatural to assume the global time reference and global states of the system. [Local computations carried out by a PDP-10 at CMU are irrelevant to computations carried out by a PDP-10 at Stanford even if two computers are connected through the ARPA network.]

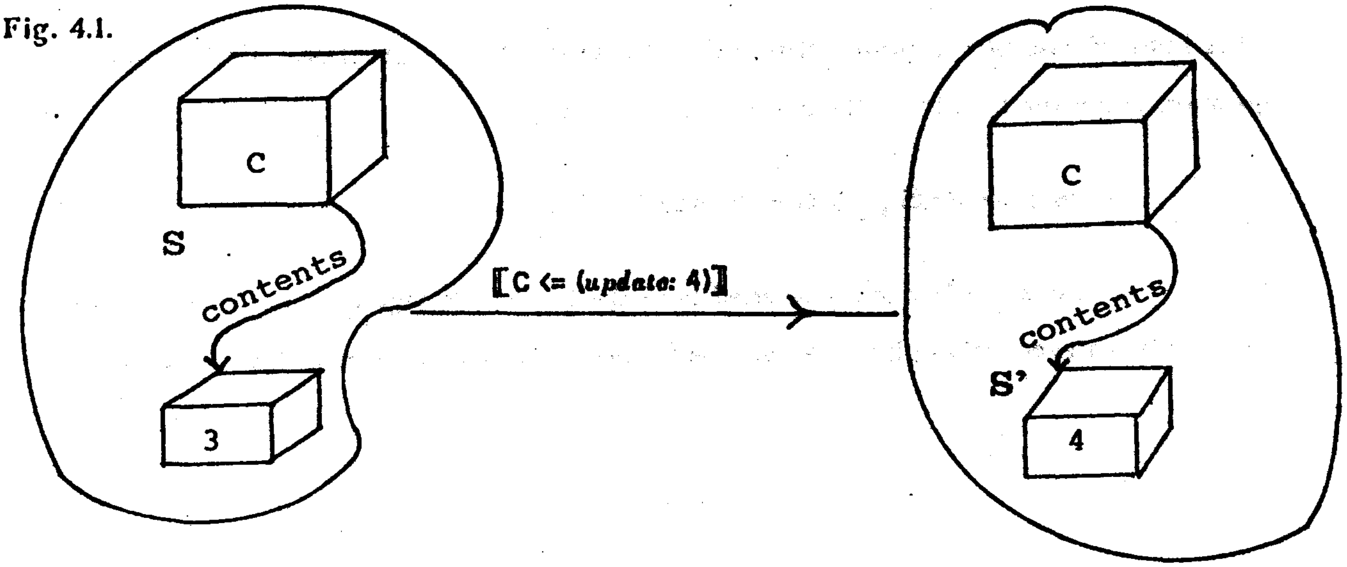
In our formalism, states of actors and actor systems are always used with reference to situations. From this viewpoint, situations can be considered as references of the local time. For example, the contents of a cell-actor C changes from time to time according to the update messages which have been most recently sent to C. Suppose that the contents of C is 3 in a situation **S** where C receives (*update:* 4) message. Then in the next situation **S'** where C receives the message (*contents:*), the contents of C is 4. (See Figure 4.1)

By using a symbol **S** to denote a situation, we express the contents of C in the situation in the following manner

$$(\textit{contents}\ C) = 3 \quad \textit{in}\ \mathbf{S}$$

We call a symbol such as **S**, which is used to refer to a situation, a *situational tag*.

Fig. 4.1.



Uses of situational tags considerably increase the expressive power of our formalism. For example, suppose that we have two impure queue-actors, queue-1 and queue-2, and that some event takes place in a situation $\mathbf{S}_{pre}$. Let $\mathbf{S}_{post}$ denote the situation after that event. Then the question and assertion of whether or not the length of queue-1 is equal to that of queue-2 in $\mathbf{S}_{post}$ is stated as follows:

(*length* queue-1) = (*length* queue-2) *in* $\mathbf{S}_{post}$.

By distributing the situational tag $\mathbf{S}_{post}$, the same statement can be made in the following two different ways:

((*length* queue-1) *in* $\mathbf{S}_{post}$) = ((*length* queue-2) *in* $\mathbf{S}_{post}$) or

(*length* (queue-1 *in* $\mathbf{S}_{post}$)) = (*length* (queue-2 *in* $\mathbf{S}_{post}$))

Since situational tags allow us to relativize facts, relations between facts holding in different situations can be easily stated in our formalism. For example, an assertion that the length of queue-1 in $\mathbf{S}_{post}$ is greater than the length of queue-1 in $\mathbf{S}_{pre}$ is stated as:

((*length* queue-1) *in* $\mathbf{S}_{post}$) > ((*length* queue-1) *in* $\mathbf{S}_{pre}$)

This kind of assertion is quite useful to show the termination of programs. Furthermore a question about the identity of the queues is easily stated as:

(queue-1 *in* $\mathbf{S}_{post}$) *is-eq* (queue-2 *in* $\mathbf{S}_{pre}$)

Situations are frequently referred to as the time of message arrival, namely at the time when an event takes place. We use the following notations to refer to such situations.

Sit(**[**T <== M**]**), *Sit*(<event>)

4.1.3 States, Identities and Conceptual Representations

An actor may change its state from situation to situation and different actors may have the same state in the same situation. Thus, in developing a specification language, we must distinguish the *state* of an actor from its *identity*.[1]

In order to describe states of actors in our specification language, we use conceptual representations introduced in Chapter 2. Identities of actors are expressed by syntactic constructs different from conceptual representations. The most general form to express the fact that an <actor> has a state expressed by a <conceptual representation> in a <situation> is as follows.

(<actor> *is-a* <conceptual-representation>) *in* <situation>

For example, suppose that the state of an impure queue-actor Q which has three elements A B and C is expressed by a conceptual representation:

(*IMPURE-QUEUE* [A B C])

Then the fact that Q has the above state in a situation **S** is expressed as

(Q *is-a* (*IMPURE-QUEUE* [A B C])) *in* **S**

It is very important that the role of conceptual representations in our specification

---

[1] We assume that the identity of an actor never changes.

language is to describe only <u>states</u> of actors, but not to represent identities of actors. [When we introduced conceptual representations to give formal specifications of data structures in Chapter 2, the separation of states from identities was not made clear.]

A predicate "*is-a*" is used to associate the state of an actor with its identity. In order to differentiate identities of actors, a predicate "*is-eq*" and its negation form "*not-eq*" are used. Since many actors may have the same state in the same situation, when the following assertion holds,

(Q' *is-a* (*IMPURE-QUEUE* [A B C])) *in* **S**,

it may or may not be the case that

(Q' *is-eq* Q).

When the sharing of actors is involved, the separation of states from identities in the formalism considerably simplifies the process of keeping track of changes in situations. For example, suppose that two different cell-actors G and H contain the same impure queue-actor Q in a situation **S**. This is expressed as:

(G *is-a* (*CELL* (*contents:* Q)))

(H *is-a* (*CELL* (*contents:* Q)))

(Q *is-a* (*IMPURE-QUEUE* [A B C]))

Then in the situation **S**, an actor D is enqueued at the rear of Q. A description of the next situation **S'** can be obtained simply by changing the state of Q into

(Q *is-a* (*IMPURE-QUEUE* [A B C D]))

and the assertions about G and H need not be modified.[1] This is an example of our technique of manipulating assertions which will be discussed extensively in the next chapter.

4.2 Types, Views and Conceptual Representations

Before going into the details of our specification language, it would be interesting to consider the roles of conceptual representations in the actor model of computation.

Actors are the only objects in the model of computation. Actors are untyped. We do not assume that actors are intrinsically classified into subcategories such as types and modes. There are two reasons for this. One is that actors are objects in an abstract model of computation, not objects in programming languages which often have types and modes for reasons of reliability and implementation efficiency. Another reason, which is more fundamental, is that we like to emphasize the *behavioral* view of actors. That is to say, we like to be able to use two actors interchangeably and indistinguishably as long as they show the same behavior with respect to some purposes and environments where they are primarily used. Also the same actor should be able to behave quite differently for different purposes and in different environments. In other words, we should be able to take a multiple view for individual actors. We believe that such multiple views encourage one to employ flexible *distribution* of computing power and intelligence such as polymorphic operators[Greif-Hewitt75] and the negotiation style of programming using coroutines in writing programs for distributed systems[Yonezawa-Hewitt77] and Artificial Intelligence

---

[1] To insure the validity of these assertions in S', we need certain rules which will be discussed in Section 5.1.3, Chapter 5.

research[Hewitt75]. Thus it seems beneficial to allow a single actor to have a broad role which would be narrowed by imposing a strict type on it.

Conceptual representations provide us with the means to express not only states of actors, but also multiple views and summaries of behaviors Such views and summaries expressed by conceptual representations facilitate the understanding and implementation of the behavior of actors.

4.3 A Specification Language

In this section, we explain basic constructs of our specification language for *serial computations* and also discuss some issues of the time variant behavior of actors related to specification languages. The specification language presented in this section will be extended to include parallel computations in Chapter 6.

4.3.1 Specifications of Events

A "specification" of an event is a formal description of effects caused by an event which takes place in an actor system. Roughly speaking, the effects of an event E are described by the next event caused by E and assertions which hold in the situation where the next event takes place. The choice of the next event from the set of the subsequent events caused by E is determined by the level of detail and the purpose of the specification.

A general form of specification for an event in our specification language is written in the following notations:

```
<event:  E
          .
          .
          .
    (Case-i:
       <pre-cond:       ... assertions ...     >
       <caused-event:  E'  >
       <post-cond:      ... assertions ...     >)
          .
          .
          .             >
```

E is the event whose effects are described. Since the effects of E may vary depending upon the situation where E takes place, the description of the effect may be divided into more than one case. The assertions in the *<pre-cond:...>* clause state the prerequisite which has to be satisfied in the situation where E takes place. When the prerequisite is satisfied, the event E' in the *<caused-event:...>* clause always takes place and the assertions in the *<post-cond: ...>* clause hold in the situation where E' takes place. More formally,

```
for E,
    if  <assertions-in-precond> in Sit(E)
      then ∃ E'
          such that E --> E' and <assertions-in-postcond> in Sit(E')
```

The prerequisite stated in each (*Case-i:...*) clause must be mutually exclusive. From this, the above notation can always specify the effects of an event deterministically. The *<event: ...>* clause need not contain all possible cases where E might take place. [In other words, the logical union of the prerequisites for each case need not be universally true.] When E does not takes place in any of the stated cases, we assume that the caused effects are undefined. The scope of names and variables in the above notation is always local to each (*Case-i:..*) clause. That is, the same names and variables in different (*Case-i: ...*) clauses do not have to refer to the same object. Names and variables appearing in the expression which represents the event E are global to each (*Case-i:...*) clause.

Though the above notation is broadly applicable, we often use abbreviated forms for events which initiate nested activities (cf. Section 3.1.3, Chapter 3). Suppose that the event E is of the form:

[T <== [*request:* M *reply-to:* C]]

and the corresponding caused event E' is of the form:

[C <== [*reply:* R]]

where R is the actor which is received by the continuation actor C in the message of E. Then we may use the following abbreviated form:

```
<event: [T <= M]
                .
                .
                .
     (Case-i:
        <pre-cond: ... assertions ...>
        <return: R  >
        <post-cond: ... assertions ...>)
                .
                .
                .        >
```

For example, the effects of an event where a cell-actor C which has the contents B receives the retrieving message (*contents:*) is written using the abbreviated form as follow. [Note that there is only one case to be specified in this example. So the (*Case-i:...*) notation can be omitted.]

<*event:* ⟦C <= (*contents:*)⟧
  <*pre-cond:* (C *is-a* (*CELL* (*contents:* B))) >
  <*return:* B >
  <*post-cond:* (C *is-a* (*CELL* (*contents:* B)))>>

Other abbreviated forms are obtained by omitting <*pre-cond:...*>, <*caused-event:...*>, <*return:...*> or <*post-cond:...*> clauses. When an event has no prerequisite, the <*pre-cond:...*> clause may be omitted. For example, the creation of a cell-actor does not have any prerequisites. Its specification is written as follows:

<*event:* ⟦**create-cell** <= A⟧
  <*return:* C^* >
  <*post-cond:* (C *is-a* (*CELL* (*contents:* A)))>>

where **create-cell** is an actor which creates a new cell-actor and A is its initial contents.

In general, in our specification language, underlined words such as **create-cell** are constant symbols which always denote a fixed actor. Non-underlined words which denote an actor are free variables and can be used as pattern variables in the process of symbolic evaluation which we will discuss in the next chapter. The notation <**actor**>* means that an <actor> is newly created and is not *is-eq* (cf. Section 4.1.3) to other actors created before.

When one is not interested in the assertions holding in a situation where the caused event takes place, the <*post-cond:...*> clause may be omitted. Furthermore when one is not interested in the caused event, the <*caused-event:...*> or <*return: ...*> clauses may be omitted too. For example, when the contents of a cell-actor is updated, what event is caused or whether the caused event might take place or not are sometimes not important. In such cases, a specification of the update event may be written as follows.

<*event:* [C <= (*update:* A)]
    <*pre-cond:* (C *is-a* (*CELL* (*contents:* B))) >
    <*post-cond:* (C *is-a* (*CELL* (*contents:* A))) >>

4.3.2 Specifications of Actors (Contracts)

Every actor has its own finite fixed set of message types that it can accept. For example, a cell-actor accepts two types of messages, (*contents:*) and (*update:* <new-element>), and a queue-actor accepts two types of messages, (*nq:* <new-element>) and (*dq:*). A specification of an actor A must contain the specifications of all events, each of which is the receipt of one type of messages that A can accept. It should also contain the specification of the event where A is created, if it is possible to create A during computations.

As an example, let us specify the behavior of pure queue-actor (cf. Section 3.2.2, Chapter 3) in our specification language. First, we describe the creation of a pure queue-actor.

<*event:* [create-pure-queue <= []]
    <*return:* Q^* >
    <*post-cond:* (Q *is-a* (*PURE-QUEUE* [])) >>

This tells us the following three things:

1) A new pure queue-actor Q is created by an event where the create-pure-queue actor receives an empty sequence actor [].
2) The creation event has no prerequisite.
3) States of a pure queue-actor is expressed by conceptual representations of the form: (*PURE-QUEUE*[...]) in the specification.

Next, we specify the enqueue event where a pure queue-actor receives (*nq:* <element>).

```
<event: ⟦Q <= (nq: A)⟧
   <pre-cond: (Q is-a (PURE-QUEUE [!x])) >
   <return:  QQ* >
   <post-cond:
       (QQ is-a (PURE-QUEUE [!x A]))
       (Q is-a (PURE-QUEUE [!x])) >>
```

This tells us that:

1) A new pure queue-actor QQ is created and returned,
2) A becomes the last element of QQ and the rest of QQ's elements are the same as those of Q, and
3) The state of Q does not change.

The specification of the dequeue event can be written in a similar way.

In addition to specifications of events associated with an actor A being specified, the specification of A may include some related information which is necessary or helpful for using and understanding the specification. Definitions of auxiliary conceptual representations used in the specification, definitions of attributes or properties of A and certain rules[1] concerning the validity of assertions used in the specification are examples of such information contained in the specification. In the case of a pure queue-actor, for example, the following definition of a property "length" may be given in the specification.

```
<property:  length-of(Q) ≡ length[!x]
     where  (Q is-a (PURE-QUEUE [!x])) >
```

Length-of is the newly defined property of a pure queue-actor and length is a function

---

[1] Such rules will be explained in the next chapter.

predefined on conceptual sequences. This definition says that *length-of* of a pure queue-actor in a situation where its states is expressed as (Q *is-a* (*PURE-QUEUE* [!x])) is obtained by calculating length[!x].

We often use the term "contract" instead of "specification" to emphasize the fact that it is an agreement or a "treaty" between the implementors of an actor (module) and its designers or clients, and also between its implementor and its users. Users of a module M should rely only on properties stated in the contract of M. On the other hand, when implementors construct the module M, they are required to satisfy only what is stated in the contract of M. In the process of symbolic evaluation of a program which uses a module N, only properties of N which are derived from the contract of N should be used. In Figure 4.2, we give a contract of pure queue-actors. It should be noted that the scope of names and variables in contracts are always local to specifications of events, definitions, and rules. For example, Q in the first *<event:...>* clause in Figure 4.2 does not necessarily denote the same actor as Q in the second *<event:...>* clause.

4.3.3 Validity of Assertions in Specifications

There are two important assumptions about assertion in specifications of events. One assumption is that states of actors which are not explicitly stated in specifications are unknown. That is, we assume that we do not know how an event E effects actors which are not mentioned in the specification of the event E. This assumption requires that effects of an event should be stated in specifications as explicitly as possible in accordance with the level of detail of the specifications. The other assumption is that assertions are usually valid only in the situations where they are stated. If the state of an actor A is given in a *<pre-cond:...>* clause of the specification of an event E and the state of A is not given in the

Fig. 4.2. A Contract for Pure Queues

```
<event: ⟦create-pure-queue <= []⟧
    <return: Q*)
    <post-cond: (Q is-a (PURE-QUEUE [])) > >

<event: ⟦Q <= (nq: A)⟧
    <pre-cond: (Q is-a (PURE-QUEUE [!x])) >
    <return: QQ* >
    <post-cond:
       (QQ is-a (PURE-QUEUE [!x A]))
       (Q is-a (PURE-QUEUE [!x])) > >

<event: ⟦Q <= (dq:)⟧
   (case-1:
      <pre-cond: (Q is-a (PURE-QUEUE [])) >
      <return: (exhausted:) >
      <post-cond: (Q is-a (PURE-QUEUE [])) > )

   (case-2:
      <pre-cond: (Q is-a (PURE-QUEUE [B !y])) >
      <return: (dequeued: BB (rest:QQ*)) >
      <post-cond:
         (QQ is-a (PURE-QUEUE [!y]))
         (Q is-a (PURE-QUEUE [B !y])) > )>

<property: length-of(Q) ≡ length[!x]
      where (Q is-a (PURE-QUEUE [!x])) >
```

corresponding <*post-cond:...*> clause, we assume that the state of A after the event E is unknown. It may or may not remain unchanged. For example, the state of a pure queue-actor after the enqueue event does not change. As stated in the contract of pure queue-actors in Figure 4.2, the assertion about the state of a pure queue-actor:

(Q *is-a* (*PURE-QUEUE* [!x]))

is repeated in the <*post-cond:...*> clause. Since a pure queue-actor does not change its state after the creation [from the definition of "purity"], this repetition of the assertion may be superfluous. But there is no way of knowing whether or not the actor being specified is pure.

4.4 Examples of Specifications

In this section, several other characteristic examples of specifications (contracts) written in our specification language are given. Some of the specifications given here are followed by the corresponding PLASMA implementations.

4.4.1 A Contract for Impure Queues

In contrast to the contract for pure queue-actors in Figure 4.2, we give a contract for impure queue-actors in Figure 4.3. As discussed in Section 3.2.2, an impure queue-actor never creates a new queue-actor in response to (*nq:...*) or (*dq:*) messages: instead it changes its own state.

Fig. 4.3. A Contract for Impure Queues

```
<event: ⟦create-impure-queue <= []⟧
   <return: Q* >
   <post-conditions: (Q is-a (IMPURE-QUEUE [])) > >

<event: ⟦Q <= (nq: A)⟧
   <pre-conditions: (Q is-a (IMPURE-QUEUE [!x])) >
   <return: Q >
   <post-conditions: (Q is-a (IMPURE-QUEUE [!x A])) > >

<event: ⟦Q <= (dq:)⟧
  (case-1:
    <pre-conditions: (Q is-a (IMPURE-QUEUE [])) >
    <return: (exhausted:) >
    <post-conditions: (Q is-a (IMPURE-QUEUE [])) > )

  (case-2:
    <pre-conditions: (Q is-a (IMPURE-QUEUE [B !y])) >
    <return: (dequeued: B (rest: Q)) >
    <post-conditions: (Q is-a (IMPURE-QUEUE [!y])) > )>
```

4.4.2 A Specification for a Message-Change Interaction

As an example of specifications for the Message-Change Type interaction (cf. Section 3.2.4, Chapter 3), a contract for an actor which empties the elements of one impure queue-actor into another impure queue-actor is given in Figure 4.4. A PLASMA implementation of an actor which satisfies the contract above is given in Figure 4.5. This implementation will be verified against the above contract by the technique of symbolic evaluation in Chapter 5.

Fig. 4.4. A Contract for empty-one-queue-into-another

```
<event: [empty-one-queue-into-another <= [Q1 Q2]]
  <pre-cond:
        (Q1 is-a (IMPURE-QUEUE [!x1]))
        (Q2 is-a (IMPURE-QUEUE [!x2]))
        (Q1 not-eq Q2) >
  <return:  (done: [Q1 Q2]) >
  <post-cond:
        (Q1 is-a (IMPURE-QUEUE []))
        (Q2 is-a (IMPURE-QUEUE [!x2 !x1])) >>
```

Fig. 4.5. A PLASMA of empty-one-queue-into-another

```
(empty-one-queue-into-another ≡
  (≡> [=q1 =q2]             ;two impure queues are received by empty-one-queue-into-another
                                                                     ;and bound to q1 and q2.
    (rules (q1 <= (dq:))                                         ;the dequeuing message is sent to q1.
       (≡> (exhausted:)                         ;if q1 is empty, the complaint message is received
          (done: [q1 q2]))                        ;then emptied q1 and extended q2 are returned.
       (≡> (dequeued: =front-of-q1               ;if q1 is not empty, the front element of q1 and
            (rest: =dequeued-q1))                                  ;the remaining queue are received
                                                       ;and bound to front-of-q1 and dequeued-q1.
      (q2 <= (nq: front-of-q1))                               ;front-of-q1 is enqueued at rear of q2.
      (empty-one-queue-into-another <= [dequeued-q1 q2])) ) ))
                          ;dequeued-q1 and q2 are sent to empty-one-queue-into-another.
```

4.4.3 A Specification for a Target-Message-Change Interaction

As an example of specifications for the Target-Message-Change Type interaction (cf. Section 3.2.4, Chapter 3), we give a specification for an interaction between a vender who sells some goods and a customer who buys the goods. The state of a vender who has some amount of money and goods with him is expressed by conceptual representations of the form

(*VENDER* (***bills:*** {...})(***goods:*** {...}))

The state of a customer who is carrying some amount of money and belongings is expressed by conceptual representations of the form

(*CUSTOMER* (***bills:*** {...})(***belongings:*** {...}))

Their interaction is described by the event specification in Figure 4.6.

Fig. 4.6. A Specification for an Interaction Between a Vender and a Customer

```
<event: ⟦V <= C⟧
   <pre-cond:
         (V is-a (VENDER (bills: {!bs}) (goods: {!g !s})))
         (C is-a (CUSTOMER (bills: {!bc !m}) (belongings: {!p})))>
   <return:  C  >
   <post-cond:
         (V is-a (VENDER (bills: {!bs !m}) (goods: {!g})))
         (C is-a (CUSTOMER (bills: {!bc}) (belongings: {!p !s})))
         (worth[!s] = total-amount[!m])  >>
```

4.4.4 Contracts for Generators

A generator is an actor which behaves like a sequence of the possible answers to some problem. When it receives a (*next:*) message, a next answer is generated. As examples, we consider two actors which successively generate increasing squares. One is a pure generator-actor, called a "port-of-squares", and the other is an impure one, called a "stream of squares". A contract for each generator is given in Figure 4.7 and Figure 4.8. In the first event specifications in both contracts, l and u denote the lower bound and the upper bound, respectively.

Fig. 4.7. A Contract for Port-of-Squares

```
<event: [create-port-of-squares <= [l u]]
  <pre-cond:  (l ≤ u) >
  <return:  PS* >
  <post-cond:  (PS is-a (PORT-OF-SQUARES (low: l) (high: u))) >>

<event: [PS <= (next:)]
 (Case-1:
  <pre-cond: (PS is-a (PORT-OF-SQUARES (low: k) (high: k))) >
  <return:  (exhausted:) >
  <post-cond: (PS is-a (PORT-OF-SQUARES (low: k) (high: k))) >)

 (Case-2:
  <pre-cond:
        (PS is-a (PORT-OF-SQUARES (low: l) (high: u)))
        (l  <  u) >
  <return:  [l^2  PSS*] >
  <post-cond:
        (PSS is-a (PORT-OF-SQUARES (low: l + 1) (high: u)))
        (PS is-a (PORT-OF-SQUARES (low: l) (high: u))) ) >
```

Fig. 4.8. A Contract for Stream-of-Squares

```
<event: ⟦create-stream-of-squares <= [l u]⟧
  <pre-cond:  (l ≤ u) >
  <return:  SS* >
  <post-cond:  (SS is-a (STREAM-OF-SQUARES (low: l) (high: u))) >>

<event: ⟦SS <= (next:)⟧
 (Case-1:
  <pre-cond: (SS is-a (STREAM-OF-SQUARES (low: k) (high: k))) >
  <return:  (exhausted:) >
  <post-cond: (SS is-a (STREAM-OF-SQUARES (low: k) (high: k))) >)

 (Case-2:
  <pre-cond:
          (SS is-a (STREAM-OF-SQUARES (low: l) (high: u)))
          (l  <  u) >
  <return:  [l^2  SS] >
  <post-cond:
          (SS is-a (STREAM-OF-SQUARES (low: l + 1) (high: u))) >) >
```

4.4.5 A Contract for **average**

In this subsection, we give a contract for actors whose behavior depends directly on the history of incoming messages. Obviously such actors are impure. An example given here is a contract for the "average" actor, which returns the average of all the numbers which have been sent to it. The contract is given in Figure 4.9.

The conceptual representation (*AVERAGE* [...]) for the actor explicitly represents the history (sequence) of all the numbers which have been received by the actor. This idea is similar to that of Clint[1973] who has introduced a "mythical pushdown stack" to have the history recorded as a kind of comments in program texts to aid the verification of programs. The function **average-of** in the contract in Figure 4.9 is defined on conceptual sequences.

Fig. 4.9. A Contract for average

```
<event: [create-average <= I]
    <return: A* >
    <post-cond: (A is-a (AVERAGE [I]))) >>

<event: [A <= (new-element: N)]
    <pre-cond: (A is-a (AVERAGE [!x])) >
    <return: A >
    <post-cond: (A is-a (AVERAGE [!x N]))  >>

<event: [A <= (average:)]
  <pre-cond: (A is-a (AVERAGE [!x])) >
  <return: average-of[!x] >
  <post-cond: (A is-a (AVERAGE [!x])) >>
```

4.5 Relationship to Other Work

At this point in our exposition, it would be useful to discuss our specification techniques for serial computation in relation to other work in this area.

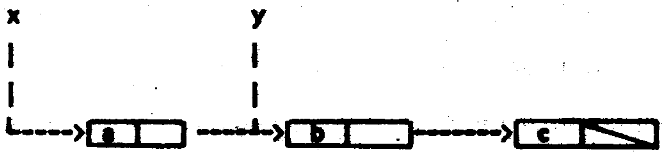
4.5.1 Behavioral Specifications

Based on the actor model of computation, I. Greif and C. Hewitt [Greif-Hewitt75, Greif75] have developed the behavioral approach to the specification technique. In their approach, the behavior of an actor (or an actor system) is specified in the form of axiom about *events* and the *precedes order relation.* Axioms describe the kinds of events that can or must take place and the order in which these events can or must occur. Axioms describe conditions which must be satisfied by computations.

This approach can deal with the time variant behavior of actors and parallelism, but makes no use of the notion of states of an actor A [which we have defined as equivalence classes of messages sent to A]. Therefore, for example, in writing axioms which specify responses to a message sent to A, the previous history of computations of A must be written out explicitly. The lack of the notion of states in their approach makes specifications long and difficult to understand. In particular, axioms for the behavior of impure actors which behave like data structures tend to be very complicated and unnatural. [Imagine the axioms for impure queue-actors.] The reader of such specifications of a data structure could understand only through reinterpreting axioms in terms of his intuitive notion of states of the data structure. In our approach, states of actors play the central roles in specifications and they are described by conceptual representations concisely, clearly and yet rigorously.

4.5.2 Burstall's Work

By extending Floyd-Hoare[Floyd67, Hoare69] approach, R. Burstall[1972] has proposed some specification and verification techniques which are able to deal with list processing languages with "side-effect" primitives such as rplaca and rplacd. To cope with the problem of side-effects in list structures, he uses a special notation for linear list structures. For example, a list structure:



is expressed in his notation as follows.

$$(x \xrightarrow{a} y \xrightarrow{b\ c} \mathit{nil})$$

Though one might find some similarity between Burstall's notations and those based upon conceptual representations, it is difficult to accommodate his notations to a wide variety of data structures.

4.5.3 Rich and Shrobe's Work

C. Rich and H. Shrobe have developed a specification language for LISP which is used in their LISP understanding system[Rich-Shrobe76]. In their system, the reasoning techniques used to deal with the problem of side-effects in LISP are along the same lines as ours. However, the clear separation of identities of objects from states of objects (cf.

Section 4.1.3) is not realized in their formalism. Thus specifications in their language tend to be long and are difficult to use for other programming languages. For example, let us look at an example of specification given in [Rich-Shrobe76].

```
(Spec-for: SWAP
        (Input: PAIR-1)
        (Output: PAIR-2)
        (Assert:
          (ID PAIR-1  PAIR-2)
          (LEFT PAIR-2 [RIGHT PAIR-1])
          (RIGHT PAIR-1 [LEFT PAIR-1])))
```

SWAP operates on a LISP pair to exchange its left element and right element. No new pairs are created by this operation. In the specification above, names PAIR-1 and PAIR-2 denote the same pair object P, which is stated by the first assertion in the (*Assert:...*) clause. The reason why they need to use two different names for the one object P is to distinguish the state of P before the operation from that of P after the operation. In our specification language the SWAP operation can be written without introducing a different name for P. Using a conceptual representation which describes the state of a pair object, a specification for SWAP is given as follows.

```
<event: ⟦SWAP <= P⟧
    <pre-cond:  (P is-a (PAIR (left: A) (right: B)))  >
    <post-cond:  (P is-a (PAIR (left: B) (right: A)))  >>
```

4.5.4 Floyd-Hoare Approach

The traditional Floyd-Hoare approach[Floyd67, Hoare69, Hoare72, Igarashi-et-al75, Suzuki75] to the specification and verification of programs has been limited in its ability to deal with programs which change their behavior. For example, the sharing of data structures in simple ALGOL-like languages is difficult to treat. Suppose that in the following code x and y are two- and one-dimensional arrays, respectively.

```
y ← x[3, ];                    ;a slice of x is shared by y.
x[3, 4] ← x[3, 4] + 1;
```

Their assignment rule cannot derive the correct value of y[4] after the above code is executed. The reason is that the value (i.e. state)[1] of an program variable is not distinguished from its identity.

Furthermore, the lack of the separation of states from identities makes it difficult for their approach to deal with specification and verification of programs written in SIMULA-like object-oriented languages. For example, their formalism is inadequate to deal with the following simple piece of SIMULA code:

```
queue-1 : - new create-impure-queue();
queue-2 : - queue-1.enqueue(2);
queue-2.enqueue(3);.
```

In the next chapter we will demonstrate how this kind of code is treated in our formalism.

---

[1] In the traditional Floyd-Hoare approach, variables in assertions denote literal program variables. Thus the value of a program variable should be considered as its state.

4.5.5 Algebraic Specification Techniques

As discussed in Section 2.4.1, Chapter 2, algebraic techniques [Zilles74, Guttag75] have been developed for the specification of abstract data types[Liskov-Zilles74]. In the algebraic approach, all operations and procedures are specified as *functions*, which leads to a serious problem; the purity and impurity (cf. Section 3.2.1, Chapter 3) of data structures cannot be easily distinguished.

As an example, let us consider an algebraic specification of queues given in [Guttag75]. Important operations on a queue are ADD and REMOVE, whose functionality is as follows.

$$\text{ADD} : \mathit{Queue} \times \mathit{Integer} \longrightarrow \mathit{Queue}$$
$$\text{REMOVE} : \mathit{Queue} \longrightarrow \mathit{Queue}$$

The essential part of the specification is given by the following equation:

$$\text{REMOVE}(\text{ADD}(q, i)) = \text{ADD}(\text{REMOVE}(q), i) \qquad (*)$$

where q is not an empty queue. In their interpretation, operations such as ADD and REMOVE always create new objects and cause no side-effects to the objects that they operate on. Equations of operations such as (*) define congruence relations over the word algebra constructed from the operations and objects. Thus in their approach, algebraic techniques are used to specify the behavior of only pure actors (immutable objects).

There is another interpretation. If we consider the domain and range of operations as sets of states of objects, equations (axioms) of the operations can define congruence relations over the states of objects. In this interpretation, algebraic techniques can be used <u>only</u> for impure actors (mutable objects)

In either interpretation, the algebraic approach has difficulties in dealing with both pure actors and impure actors simultaneously. Techniques developed by J. Spitzen and B. Wegbreit [Spitzen-Wegbreit75, Wegbreit-Spitzen76] have the same problem of distinguishing the purity and impurity of data structures.

5. Verifying Serial Computations

In this chapter, our verification techniques for serial computations are presented. The first section describes the method of symbolic evaluation which is the major instrument in our verification techniques. It also contains a detailed explanation of our reasoning method which can be employed in environments where computations may cause side-effects. The next two sections describe our verification methods, each of which is applied to different types of actors. Then, to close the chapter, we reflect on our method of symbolic evaluation and discuss its application to other areas.

5.1 Symbolic Evaluation

In this section, we will describe our basic method of symbolic evaluation, the major instrument of our verification techniques. A simple example of symbolic evaluation of PLASMA code which involves sharing of actors with side-effects is given at the end of the section. Although in this thesis we consider symbolic evaluation primarily as a tool for program verification, it is also useful for other purposes such as program testing, debugging, optimization, dependency analysis, perturbation analysis etc. The chapter concludes with a discussion of some potential applications.

5.1.1 Overview

Symbolic evaluation is a process which abstractly [symbolically] executes programs on abstract [symbolic, as opposed to "concrete"] data. When a program takes numerical input, the symbolic evaluation of the program does not deal with concrete numbers such as 123, 1776, and 1984, but rather with symbolically expressed numbers such as "n1", "n2", and "m".

Though symbolic evaluation is an extension of ordinary execution of programs, it differs from ordinary execution in the following points.

(1) The only properties of input that can be used are the ones specificed as the prerequisites of a module being symbolically evaluated. [E.g., input numbers are required to be positive integers.]

(2) When the symbolic evaluation of a module M encounters an invocation of some module N, the specification [contract] of N is used to continue the symbolic evaluation. The implementation of N is not used.

Symbolic evaluation can be viewed as a mechanization of the process of a human programmer tracing a program without using concrete values to understand the computations expressed by the program.

In symbolic evaluation, the code of a module is interpreted step by step according to either pre-defined semantics of language primitives or specifications of modules invoked in the module. Each such step is triggered by the symbolic evaluation of an expression in the code which corresponds to an event [cf. Section 3.1.2, Chapter 3]. The state of the program [code] at each moment before and after an interpretation step is referred to as a *situation*. The symbolic evaluator[1] has a data base to record what events occur, what facts hold and what is assumed in each situation. Facts that hold in a situation **S** are recorded as assertions associated with **S**.

Since each expression is interpreted on abstract data, when a conditional expression is interpreted, the subsequent symbolic execution path must split in the usual fashion[Deutch 1973]. For example, consider the symbolic evaluation of

if (P x) ***then*** ... ***else*** ....

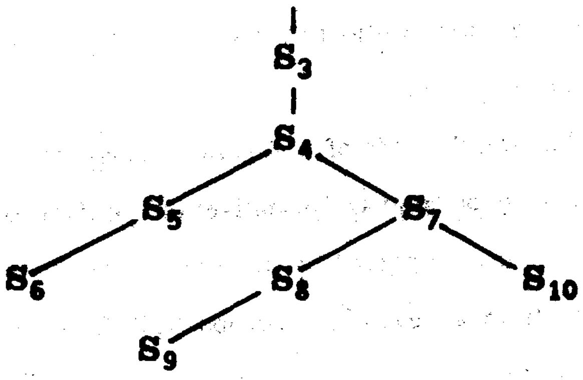
After the symbolic evaluation of the expression (P x), the symbolic execution path splits into two branches: one for the **then**-clause and the other for the **else**-clause. To start the subsequent symbolic evaluation, (P x) must be assumed for the **then**-clause and ¬(P x) for the **else**-clause. If the evaluation of (P x) has no side-effects, the assertions holding in the situation where (P x) is evaluated are inherited for both clauses.

In essence, symbolic evaluation is a process which abstractly evaluates the code

---

[1] In this chapter, we assume that symbolic evaluation is carried out by a hypothetical system.

Fig. 5.1. A Situational Tree



---

forward along the execution path and produces a tree structure whose nodes correspond to situations. At each node of the tree, assertions which hold in the corresponding situations are entered. We call this structure a *situational tree.* [See Figure 5.1.] The assertions entered in the situational tree are used as the primary source of information for answering questions about the implementation. As we shall later see, verification of implementations is carried out by using such situational trees.

5.1.2 Partial Descriptions of Situations

In order to illustrate how assertions are handled in a situational tree, we symbolically evaluate the following piece of code.

```
            - S -
(queue-1 <= (nq: 6))          ;queue-1 receives a message (nq: 6)
            - S' -
(queue-1 <= (nq: 8))          ;queue-1 receives a message (nq: 8)
            - S" -
```

S, **S'**, and **S"** denote situations before or after the events corresponding to statements in the code. We assume that two distinct impure queues, **queue-1** and **queue-2** have been created before the situation **S** and assertions about states of the two queues are already entered at the node for **S** in a situational tree. See the diagram below.

|
S : (queue-1 *is-a* (*IMPURE-QUEUE* [3 7 11]))
| (queue-2 *is-a* (*IMPURE-QUEUE* [2 4]))
|

With these assumptions, the first statement in the code which expresses an event ⟦queue-1 <= (*nq:* 6)⟧ is interpreted. To know what effects are caused by this event, the symbolic evaluator first looks for an assertions about the state of **queue-1** at the node for **S** in the situational tree. It finds that the state [or conceptual type] of **queue-1** is expressed as

(*IMPURE-QUEUE* [3 7 11])

From the form of the conceptual representation [i.e., from "*IMPURE-QUEUE*"], the contract for impure queues in Figure 5.2 is referred to.

The event expression ⟦Q <= (*nq:* A)⟧ in the second *<event:...>* clause in the contract for impure queues in Figure 5.2 matches against the event ⟦**queue-1** <= (*nq:* 6)⟧. Also the assertion

(Q *is-a* (*IMPURE-QUEUE* [!x]))

in the *<event:...>* clause matches against the assertion

(queue-1 *is-a* (*IMPURE-QUEUE* [3 7 11]))

which has been entered at the node for **S**. Thus the whole second *<event:...>* clause can be instantiated as follows.

Fig. 5.2. A Contract for Impure Queues

```
<event: ⟦create-impure-queue <= []⟧
   <return: Q* >
   <post-cond: (Q is-a (IMPURE-QUEUE [])) > >

<event: ⟦Q <= (nq: A)⟧
   <pre-cond: (Q is-a (IMPURE-QUEUE [!x])) >
   <return: Q >
   <post-cond: (Q is-a (IMPURE-QUEUE [!x A])) > >

<event: ⟦Q <= (dq:)⟧
  (case-1:
    <pre-cond: (Q is-a (IMPURE-QUEUE [])) >
    <return: (exhausted:) >
    <post-cond: (Q is-a (IMPURE-QUEUE [])) > )

  (case-2:
    <pre-cond: (Q is-a (IMPURE-QUEUE [B !y])) >
    <return: (dequeued: B (rest: Q)) >
    <post-cond: (Q is-a (IMPURE-QUEUE [!y])) > )>
```

```
<event: ⟦queue-1 <= (nq: 6)⟧
    <pre-cond: (queue-1 is-a (IMPURE-QUEUE [3 7 11]))>
    <return: queue-1 >
    <post-cond: (queue-1 is-a (IMPURE-QUEUE [3 7 11 6]))>>
```

The symbolic evaluator enters the assertion in the above *<post-cond:...>* clause at the node for the next situation **S'**. Also it records what event took place between the two situations. See the upper diagram in Figure 5.3. The second statement in the code expresses an event ⟦queue-1 <= (*nq*: 8)⟧, which is interpreted in the same way as above. The effect of this event is recorded at the node for the next situation **S''** as shown in the lower diagram of

Figure 5.3.

An important point in the manipulation of assertions described above is that the assertion about the other impure queue actor, **queue-2**, is left untouched, neither copied nor modified in going from **S** to **S'** and **S"**. As the diagrams in Figures 5.3 show, the situational tree thus generated by the symbolic evaluation does not contain assertions about the states of **queue-2** at the nodes for **S'** and **S"**. In general, a situational tree generated

---

Fig. 5.3.

```
 |
S  : (queue-1 is-a (IMPURE-QUEUE [3 7 11]))
|      (queue-2 is-a (IMPURE-QUEUE [2 4]))
|
⟦queue-1 <= (nq: 6)⟧
|
S'  : (queue-1 is-a (IMPURE-QUEUE [3 7 11 6]))
```

```
|
S  : (queue-1 is-a (IMPURE-QUEUE [3 7 11]))
|      (queue-2 is-a (IMPURE-QUEUE [2 4]))
|
⟦queue-1 <= (nq: 6)⟧
|
S'  : (queue-1 is-a (IMPURE-QUEUE [3 7 11 6]))
|
⟦queue-1 <= (nq: 8)⟧
|
S"  : (queue-1 is-a (IMPURE-QUEUE [3 7 11 6 8]))
```

by symbolic evaluation is only a *partial* description of situations. When one needs to know states of actors or relations holding in a situation, which are not explicitly asserted at the corresponding node in the situational tree, one must rely on the reasoning method described in the next subsection.

5.1.3 The Method of Reasoning (Uses of the Trans-situational Rules)

In this subsection, we will illustrate how situational trees are used for the reasoning in our formalism. In general, questions about a given situation are answered by reasoning backward. That is, to answer questions such as whether some assertions hold in a situation $\mathbf{S}$ or in what states some actors are in $\mathbf{S}$, a situational tree is looked at from the node for $\mathbf{S}$ to previous situations.

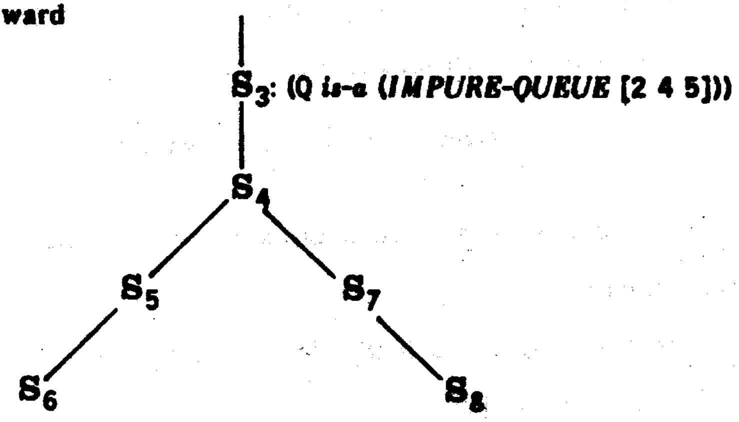
For example, suppose that a situational tree shown in Figure 5.4 is given and we want to know the state of Q in a situation $\mathbf{S}_7$. First we try to find some assertion which describes the state of Q at the node for the situation $\mathbf{S}_7$. Since the given situational tree does not have any assertions about Q at the node for $\mathbf{S}_7$, we look for assertions about Q backward along the branch of the situational tree. [See the dotted line in Figure 5.4.] An assertion

(Q *is-a* (*IMPURE-QUEUE* [2 5 4]))

is found at the node for $\mathbf{S}_3$. However, all we know at this point is that the assertion holds in $\mathbf{S}_3$. but we are not sure that the assertion holds in $\mathbf{S}_7$, because some events which destroy the validity of this assertion in $\mathbf{S}_7$ might have occured between $\mathbf{S}_3$ and $\mathbf{S}_7$. So we must check on such events.

In order to know what events nullify the validity of assertion, each event

Fig. 5.4. Reasoning Backward



---

specification in the contract for impure queues shown in Figure 5.2 is examined. If in the specification for an event E the state of Q stated in the *<pre-cond:...>* clause is different from the one in the corresponding *<post-cond:...>* clause, the event E nullifies the validity of the assertion. In fact, ⟦Q <= (*dq:*)⟧ and ⟦Q <= (*nq:...*)⟧ turn out to be such nullifying events.

The process of finding the nullifying events can be saved if the contract contains an explicit statement which indicates such events. For this purpose, we may add the following clause to the contract for impure queues.[1]

```
<for-assertion: (Q is-a (IMPURE-QUEUE [...]))
        <only-affecting-events-are:
                {⟦Q <= (nq:...)⟧, ⟦Q <= (dq:)⟧} >>
```

This statement says that the validity of assertions of the form

(Q *is-a* (*IMPURE-QUEUE* [...]))

---

[1] *<for-assertion:...>* clauses do not have to be placed in contracts for actors. They can be placed in some global place to which the symbolic evaluator have access.

is destroyed only by the set of events appearing in the *<only-affecting...>* clause.[2]

In our formalism, assertions of the form

(<actor> *is-a* <conceptual-representation>)

can be inherited from an ancestor situation $\mathbf{S}_i$ to a descendant situation $\mathbf{S}_j$ if the following two conditions are met:

(1) The events specified in the corresponding *<for-assertion:...>* clause do not take place between $\mathbf{S}_i$ and $\mathbf{S}_j$.

(2) At the node for the descendant situation $\mathbf{S}_j$, no assertions about the <actor> have been entered which use the same form of conceptual representation as used in the assertion being inherited from $\mathbf{S}_i$.

By virtue of the second condition, we do not have to keep adding events to the *<for-assertion...>* clause every time we implement a new actor which changes the state of the <actor>. For example, suppose that an actor **emptying-queue** which empties the elements of an impure queue-actor is implemented and that its specification is given as follows:

```
<event: ⟦emptying-queue <= Q⟧
    <pre-cond: (Q is-a (IMPURE-QUEUE [!x]))>
    <post-cond: (Q is-a (IMPURE-QUEUE []))>
```

When the PLASMA expression (**emptying-queue** <= Q) is symbolically interpreted in a situation $\mathbf{S}$ where (Q *is-a* (*IMPURE-QUEUE* [1 2 3])) holds, the assertion (Q *is-a* (*IMPURE-QUEUE* [])) is entered at the node for the next situation $\mathbf{S}'$. If we did not have the above condition (2), the assertion (Q *is-a* (*IMPURE-QUEUE* [1 2 3])) could be

---

[2] Note that this reasoning is valid only for serial computations. It is not valid if there are concurrent events.

inherited to S'. To prevent this invalid inheritance without the condition (2), we would need to add the event ⟦emptying-queue <= Q⟧ to the list of nullifying events.

In general, the rule which indicates what conditions guarantee valid inheritances of assertions from one situation to another is called a *trans-situational rule*. For particular assertions or particular forms of assertions, appropriate trans-situational rules are necessary for correct reasoning. The <*for-assertion:...*> clauses given in contracts are one type of trans-situational rules. In Section 5.1.5, some examples of trans-situational rules are listed. The reasoning using trans-situational rules described here is a procedural approach to McCarthy's frame problem [McCarthy-Hayes69]. We will discuss this issue in Section 5.4.

5.1.4 Variables and Identifiers

In this subsection, we will explain how names for actors are handled in symbolic evaluation for programs written in PLASMA. The technique given here allows us to deal efficiently with the problem of both identity and sharing of actors.

Names in PLASMA fall into two classes: *variables* and *identifiers*. A variable x can be declared and also initialized with the value of an expression <E1> by the following form of statements

(*let* (x *initially* <E1>)...)

The value of x can be changed only by executing expressions of the form

(x ← <E2>).

Occurrences of x in programs except in the above form stand for the value of x. A variable x is usually implemented by a cell actor, but in that case an expression x in code does not stand for the cell actor itself, but rather for the contents of the cell actor. In symbolic evaluation, to state that a <variable> has an <actor> as its value in some situation, assertions of the following form are used.

(<variable> ***has-value*** <actor>)

When the symbolic evaluator interprets an expression

(x ← <E>).

in a situation **S**, the following assertion

(x ***has-value*** B)

is entered at the node for the next situation, where B is the value of <E> in **S**.

An identifier is declared and bound to an actor in the course of program execution. To express that an <identifier> is bound to an <actor>, we use assertions of the form

(<identifier> ≡ <actor>)

In the symbolic evaluation of a module M, an identifier x used in the code of M can be always regarded as the actor that it is bound to, because one identifier is not bound to more than one actor throughout the symbolic evaluation of M. This is guaranteed by:

(1) the restriction on the syntax of PLASMA that no names are declared more than once inside a module, and

(2) the fact that symbolic evaluation passes over each expression in a module exactly once.[1]

---

[1] This fact is true only when symbolic evaluation is used for program verification.

When more than one symbol [here, symbols mean ones denoting actors in contracts (such as Q in Figure 5.2) as well as indentifiers in programs] denotes the same actor, we use assertions of the form

(<symbol-1> *is-eq* <symbol-2>)

As an identifier can be regarded as the actor that it is bound to, the relation "*is-eq*" and "≡" can be used indistinguishably. Since the relation "*is-eq*" is an equivalence relation, it forms an equivalence class of identifiers in programs and symbols denoting actors in contracts. Every member of such an equivalence class denotes the same actor. In symbolic evaluation, one identifier (or symbol) is chosen from each class [e.g., the one which is first used among the members of the class] and any uses or occurrences of other members in the same class are always considered as those of the chosen one. To record the state of an actor A, the symbolic evaluator always uses the one chosen identifier or symbol for A throughout all the situations. This arrangement eases the handling of shared actors in symbolic evaluation.

To illustrate the use of identifiers and symbols explained above, let us consider the following piece of code. This code is a PLASMA version of the SIMULA code given in Section 4.5.4, Chapter 4 as an example which is difficult for the Floyd-Hoare technique.

```
              - S0 -
(let (queue-1 ≡ (create-impure-queue ))
 then         - S1 -
 (let (queue-2 ≡ (queue-1 <= (nq: 2)))
   then       - S2 -
      (queue-2 <= (nq: 3))
              - S3 -
```

$S_0,\ldots,S_3$ denote situations before or after the events corresponding to statements in the code. In what follows, the notation

in $\mathbf{S}_{\dots}$: ...<assertion>...

means that <assertion>s are entered at the node for $\mathbf{S}_{\dots}$ in a situational tree.

The event ⟦create-impure-queue <= []⟧ takes place in $\mathbf{S}_0$. By virtue of the contract for impure queues in Figure 5.2, we know an empty impure queue-actor is created. Then the *let* statement binds the identifier queue-1 to the empty queue-actor. We may use a symbol Q for the newly created actor and record this event by two assertions

(Q *is-a* (*IMPURE-QUEUE* []))
(queue-1 ≡ Q),

but one assertion suffices. Namely,

in $\mathbf{S}_1$: (queue-1 *is-a* (*IMPURE-QUEUE* []))

The second statement of the above PLASMA code is interpreted by using the following event specification instantiated from the clause in the contract for impure queues

<*event:* ⟦queue-1 <= (*nq:* 2)⟧
  <*pre-cond:* (queue-1 *is-a* (*IMPURE-QUEUE* []))>
  <*return:* queue-1 >
  <*post-cond:* (queue-1 *is-a* (*IMPURE-QUEUE* [2]))>>

The state of queue-1 is changed as described by the assertion in the <*post-cond:...*> clause and queue-1 is returned. The *let* statement tells us that the returned queue-1 is bound to queue-2. Thus

in $\mathbf{S}_2$: (queue-1 *is-a* (*IMPURE-QUEUE* [2]))
(queue-1 *is-eq* queue-2)

In interpreting the third statement, since we know that **queue-2** and **queue-1** denote the same impure actor, the event ⟦**queue-2** <= (*nq*: 3)⟧ stands for ⟦**queue-1** <= (*nq*: 3)⟧. Thus the change in the state of **queue-1** is recorded as

in $\mathbf{S}_3$: (**queue-1** *is-a* (*IMPURE-QUEUE* [2 3]))

Any references to **queue-2** in the interpretation of the subsequent statements in the code are treated as the references to **queue-1**.

5.1.5 Examples of Trans-Situational Rules

In this subsection, we will give the trans-situational rules which will be used in the examples of symbolic evaluation in this thesis

(⁘) Assertions of the form (**<identifier>** ≡ **<actor>**)
which state that **<identifier>** is bound to **<actor>**, can be passed unchanged between any two situations within the scope of **<identifier>**.

(⁘) Assertions of the form (**<actor1>** *is-eq* **<actor2>**) and (**<actor1>** *not-eq* **<actor2>**), which state the identity of actors, can be always inherited from one situation to another without any conditions.

(⁘) Assertions of the form

(**<c-sequence1>** = **<c-sequence2>**) and (**<c-sequence1>** ≠ **<c-sequence2>**),

which state the equality of conceptual sequences appearing in conceptual representations, can also be inherited without any conditions. [Note that **<c-sequence1>** and **<c-sequence2>** are not sequence-actors but mathematical sequences. All mathematical facts can be inherited without any conditions. This is a special case.]

(*) Assertions of the form (**<actor>** *is-a* (*SEQUENCE* [!x]))
which state that **<actor>** is a sequence-actor whose elements are !x, can be inherited without any conditions because a sequence-actor is a pure actor which never changes its state.

(*) Assertions of the form (**<variable>** *has-value* **<actor>**)
which state that **<variable>** has **<actor>** as its value in some situation **S**, can be inherited to a situation **T** if no assignments to this **<variable>** take place between **S** and **T**. (Cf. Section 5.1.4.)

5.2 Verification of Actors Behaving as Procedures

Methods of verification reflect methods of specification. Roughly speaking, two methods have been employed in the specification technique presented in the previous chapter.

One method is to specify the behavior of an actor A in terms of the states or the changes in the state of other actors which are sent to A, or which are created during the invocations of A. In this method, the state of A is not used in specifying the behavior of A. Most actors which behave purely as "procedures" are specified by this method. A typical example of such actors is **empty-one-queue-into-another**. [See Section 4.4.2, Chapter 4.] In general, this method applies to the specifications of the actors which are targets in the No-Change-Type and Message-Change-Type interactions introduced in Section 3.2.4, Chapter 3.

The other method is to specify the behavior of an actor B in terms of the changes in the state of B itself. Actors which behave as "information storage", such as data structures and generators, are specified by this method.

In this section, we will illustrate our verification techniques for actors behaving like procedures, whose behaviors are specified by the first method mentioned above. The verification techniques corresponding to the second specification method will be discussed in the next section. However, since actors are essentially procedural objects whose implementations are written as programs, most of the techniques that will be discussed in this section [such as the handling of recursion, loop, case splitting and convergence] are necessary bases for the verification of information-storage-like actors.

5.2.1 Symbolic Evaluation in the Context of Specifications

In order to verify an implementation of an actor against its specification, symbolic evaluation of the implementation [i.e. code or script] is carried out in the context of the specification. In our formalism, a specification of an actor which behaves like a procedure is expressed by a specification of the event which initiates the invocation of the actor. A specification consists of the preconditions for the incoming message [i.e. input], and the postconditions to be satisfied by the result of the invocation. Thus the symbolic evaluation of the implementation is started with the assumption that the preconditions are satisfied. Under this assumption the symbolic evaluation is carried out and then the results of the symbolic evaluation are examined as to whether they satisfy the postconditions given in the specification.

Below we will demonstrate the verification of an implementation of **empty-one-queue-into-another** [hereafter **empty**] against its contract. Its contract and PLASMA code are given in Figure 5.5. The code is augmented with situational symbols

Fig. 5.5. A Contract and Implementation of empty-one-queue-into-another

```
            <event: ⟦empty-one-queue-into-another <= [Q1 Q2]⟧
              <pre-cond:
                    (Q1 is-a (IMPURE-QUEUE [!x1]))
                    (Q2 is-a (IMPURE-QUEUE [!x2]))
                    (Q1 not-eq Q2) >
              <return:  (done: [Q1.Q2]) >
              <post-cond:
                    (Q1 is-a (IMPURE-QUEUE []))
                    (Q2 is-a (IMPURE-QUEUE [!x2 !x1])) >>

        ------------------------------------------------------------

(empty-one-queue-into-another ≡
 (≡> [=q1 =q2]              ;two impure queues are received by empty-one-queue-into-another
                                                              ;and q1 and q2 are bound to them.
 - S_received-queues -

   (rules (q1 <= (dq:))                                  ;the dequeuing message is sent to q1.

      (≡> (exhausted:)                   ;if q1 is empty, the complaint message is generated
    - S_exhausted-q1 -
        (done: [q1 q2]) )                   ;then emptied q1 and extended q2 are returned.

      (≡> (next: =front-of-q1                                    ;if q1 is not empty, front-of-q1
           (rest: =dequeued-q1))                                            ;and dequeued-q1
                  ;are bound to the front element of q1 and the remaining queue, respectively.
     - S_dequeued-q1 -

        (q2 <= (nq: front-of-q1))                        ;front-of-q1 is enqueued at rear of q2.

     - S_enqueued-q2 -

        (empty-one-queue-into-another <= [dequeued-q1 q2])) ) ))
                                     ;dequeued-q1 and q2 are sent to empty-one-queue-into-another.
```

which denote situations before/after events corresponding to each statement. Note that this implementation contains a conditional branch and a recursion, the handling of which will be explained below.

First, the preconditions of **empty** in its contract are entered in the data base.

in S_{initial} :
(Q1 *is-a* (*IMPURE-QUEUE* [!x1]))
(Q2 *is-a* (*IMPURE-QUEUE* [!x2]))
(Q1 *not-eq* Q2)

After the symbolic pattern matching is performed, identifiers **q1** and **q2** are bound to Q1 and Q2, respectively. So this is recorded in the data base as the following assertions.

in $S_{\text{received-queues}}$:
(q1 ≡ Q1)
(q2 ≡ Q2)

Then the PLASMA expression (q1 <= (*dq:*)) in the **rules**-statement is interpreted. The dequeuing message (*dq:*) is sent to Q1 that q1 is bound to. To know the result of this event, the symbolic evaluator must consult the <*event*...> clause for the dequeuing in the contract:

<*event:* ⟦Q1 <= (*dq:*)⟧
  (*case-1:*
    <*pre-cond:* (Q1 *is-a* (*IMPURE-QUEUE* [])) >
    <*return:* (*exhausted:*) >
    <*post-cond:* (Q1 *is-a* (*IMPURE-QUEUE* [])) >)
  (*case-2:*
    <*pre-cond:* (Q1 *is-a* (*IMPURE-QUEUE* [B !y])) >
    <*return:* (*next:* B (*rest:* Q1)) >
    <*post-cond:* (Q1 *is-a* (*IMPURE-QUEUE* [!y])) >) >

[Note that the above clause is an instantiation of the (*event*...) clause for the dequeuing in the contract for impure queues in Figure 5.2, which is obtained by substituting Q1 for Q.] Now the symbolic evaluator has to consider two cases: where Q1 is empty and where Q1 is not empty. (See the situational tree for this example in Figure 5.6.)

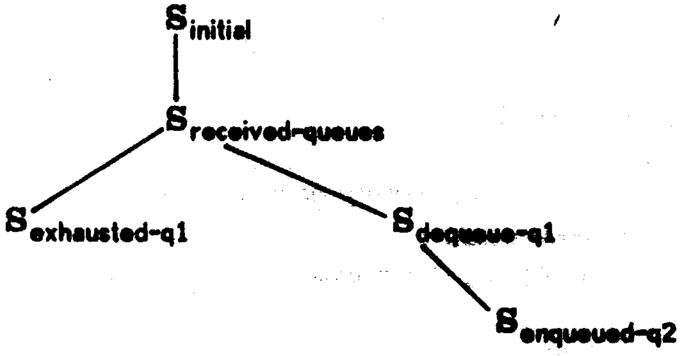
Case I: (Q1 *is-a* (*IMPURE-QUEUE* []))

In this case, the contract specifies that the (*exhausted:*) message should be returned. This message matches against the first (≡>...)-statement inside the (rules...) statement. To follow this path, x1 = [] must be assumed. So at the node for $\mathbf{S}_{\text{exhausted-q1}}$, the following assertions are entered.

in $\mathbf{S}_{\text{exhausted-q1}}$:
(x1 = [])
(Q1 *is-a* (*IMPURE-QUEUE* []))

Then the result of the invocation, the message (*done:* [q1 q2]), is returned in $\mathbf{S}_{\text{exhausted-q1}}$. For this result, there are three postconditions stated in the contract of empty:

Fig. 5.6.



$\mathbf{S}_{\text{initial}}$
$\mathbf{S}_{\text{received-queues}}$
$\mathbf{S}_{\text{exhausted-q1}}$
$\mathbf{S}_{\text{dequeue-q1}}$
$\mathbf{S}_{\text{enqueued-q2}}$

r1: (*done*: [Q1 Q2]) must be returned

r2: (Q1 *is-a* (*IMPURE-QUEUE* [])) must hold, and

r3: (Q2 *is-a* (*IMPURE-QUEUE* [!x2 !x1])) must hold.

It is easy to show that each postcondition is satisfied in $\mathbf{S}_{\text{exhausted-q1}}$:

(☆) for r1, since the trans-situational rules for binding allow the inheritance of the assertions ($\underline{\text{q1}}$ ≡ Q1) and ($\underline{\text{q2}}$ ≡ Q2) from $\mathbf{S}_{\text{received-queues}}$ to $\mathbf{S}_{\text{exhausted-q1}}$, the required message is returned in $\mathbf{S}_{\text{exhausted-q1}}$,

(☆) for r2, the assertion (Q1 *is-a* (*IMPURE-QUEUE* [])) is entered at the node for $\mathbf{S}_{\text{exhausted-q1}}$, and

(☆) for r3, the two facts guarantee that the requirement is satisfied:

(1) (Q2 *is-a* (*IMPURE-QUEUE* [!x2])) can be inherited from $\mathbf{S}_{\text{initial}}$ to $\mathbf{S}_{\text{exhausted-q1}}$ by using the trans-situational rule for (Q2 *is-a* (*IMPURE-QUEUE* [...])) [which is obtained by instantiating the <*for-assertion:...*> clause in the contract for impure queues. Cf. Section 5.1.3.]. This inheritance is legitimate because neither ⟦Q2 <= (*nq:...*)⟧ nor ⟦Q2 <= (*dq:*)⟧ have happended and no assertions of the form (Q2 *is-a* (*IMPURE-QUEUE* [...])) have been entered at the node for $\mathbf{S}_{\text{exhausted-q1}}$.

(2) [!x2] = [!x2 !x1] holds in $\mathbf{S}_{\text{exhausted-q1}}$ because x1 = [] holds in $\mathbf{S}_{\text{exhausted-q1}}$. ([!x2 !x1] = [!x2 ![]] = [!x2])

Therefore (Q2 *is-a* (*IMPURE-QUEUE* [!x2 !x1])) holds in $\mathbf{S}_{\text{exhausted-q1}}$. Thus Case-1 is verified.

Case-2: (Q1 *is-a* (*IMPURE-QUEUE* [b !y]))

In this case, the contract for impure queues tells us that (*next:* B (*rest:* Q1)) is the result of (q1 <= (*dq:*)) where the following assertions are assumed.

(x1 = [B !y])
(Q1 *is-a* (*IMPURE-QUEUE* [!y]))

The result (*next:* B (*rest:* Q1)) is matched against the pattern in the second (=>...) statement inside the (rules...) statement. At the node for $\mathbf{S}_{\text{dequeued-q1}}$, the binding information is also entered together with the above assumption.

in $\mathbf{S}_{\text{dequeued-q1}}$:
(front-of-q1 ≡ B)
(dequeued-q1 ≡ Q1)
(x1 = [B !y])
(Q1 *is-a* (*IMPURE-QUEUE* [!y]))

Then the PLASMA expression (q2 <= (*nq:* front-of-q1)) is interpreted in this situation. Since q2 is bound to Q, and front-of-q1 is bound to B [from the trans-situational rule for the binding], the event taking place is ⟦Q2 <= (*nq:* B)⟧. To know the effects of this event, the system refers to the second <*event:*...> clause in the contract for impure queues in Figure 5.2. The state of Q2 in $\mathbf{S}_{\text{dequeue-q1}}$ is obtained from the assertion (Q2 *is-a* (*IMPURE-QUEUE* [!x2])) at the node for $\mathbf{S}_{\text{initial}}$. Because it can be inherited to $\mathbf{S}_{\text{dequeued-q1}}$ for the same reason as explained above in the case of its inheritance from $\mathbf{S}_{\text{initial}}$ to $\mathbf{S}_{\text{dequeued-q1}}$. Thus the second <*event:*...> clause is instantiated as follows. [Note the substitutions of Q2 for Q, x2 for x and B for A.]

<*event:* ⟦Q2 <= (*nq:* B)⟧
<*pre-cond:* (Q2 *is-a* (*IMPURE-QUEUE* [!x2])) >
<*return:* Q2>
<*post-cond:* (Q2 *is-a* (*IMPURE-QUEUE* [!x2 B])) >>

The assertion in the <*post-cond:...*> clause is entered at the node for $\mathbf{S}_{\textbf{enqueued-q2}}$.

in $\mathbf{S}_{\text{enqueued-q2}}$: (Q2 *is-a* (*IMPURE-QUEUE* [!x2 B]))

Now the last PLASMA statement (empty <= [dequeue-q1 q2]) is interpreted. From the binding information, the corresponding event is ⟦empty <= [Q1 Q2]⟧. To know the effects of this event, the contract for empty in Figure 5.5 is referred to. Since we are trying to verify the code against this contract, this is a "recursive"[1] use of the contract. The preconditions stated in this contract must be satisfied before it can be used. In fact, the assertions:

(Q1 *is-a* (*IMPURE-QUEUE* [!y])) and (Q1 *not-eq* Q2)

can be inherited from $\mathbf{S}_{\textbf{dequeued-q1}}$ by the trans-situational rules for

(Q1 *is-a* (*IMPURE-QUEUE* [...])) and (<actor1> *not-eq* <actor2>),

respectively. Thus the following assertions hold in $\mathbf{S}_{\textbf{enqueue-q2}}$

(Q1 *is-a* (*IMPURE-QUEUE* [!y]))
(Q2 *is-a* (*IMPURE-QUEUE* [!x2 B)])
(Q1 *not-eq* Q2)

Therefore the preconditions of empty are satisfied. Now the postconditions of the contract for empty guarantee that (*done:* [Q1 Q2]) is returned and that the following assertions:

---

[1] Recursion and iteration in symbolic evaluation are discussed in Appendix III.

(Q1 *is-a* (*IMPURE-QUEUE* [])) and
(Q2 *is-a* (*IMPURE-QUEUE* [![!x2 B] !y])) hold.

hold in the situation following $\mathbf{S}_{\text{enqueued-q2}}$. Facts about c-sequences:

[![!x2 B] !y] = [!x2 B !y],
[!x2 B !y] = [!x2 !x1], if x1 = [B !y].

are used to simplify the above assertions. That is, since x1 = [B !y] can be inherited from $\mathbf{S}_{\text{dequeued-q1}}$ by the trans-situational rule for (<c-sequence1> = <c-sequence2>), it follows that

(Q1 *is-a* (*IMPURE-QUEUE* []))
(Q2 *is-a* (*IMPURE-QUEUE* [!x2 !x1])).

Thus the post-conditions for **empty-one-queue-into-another** are also satisfied in Case-2.

Though it has been shown that both Case-1 and Case-2 meet the postconditions for **empty**, we cannot conclude that the implementation of **empty** in Figure 5.5 satisfies its contract, because the convergence of the invocation of the implementation is not guaranteed, although it is explicitly required by the contract. [Recall the meaning of <*return:...*> clauses given in the previous chapter.] For after splitting into two cases at the (**rules...**) statement, the symbolic evaluation for both Case-1 and Case-2 is resumed under the <u>assumption</u> that the control has reached the points corresponding to $\mathbf{S}_{\text{exhausted-q1}}$ and $\mathbf{S}_{\text{dequeued-q1}}$. Therefore, to demonstrate that the above assumption is always guaranteed is another part of the verification process. This issue is discussed in Appendix IV.

5.3 Verification of Actors Behaving as Information Storage

In this section, we will present our specification techniques for actors whose behaviors are specified in terms of their own states [or changes in their own states]. Specifications of actors which behave as "information storage" such as data structures and generators [Section 4.4.4] are often written in terms of their own states. For the verification of implementations of these actors, symbolic evaluation is still the major instrument and all the techniques presented in the previous section are still employed. In addition, however, special considerations are necessary in dealing with conceptual representations of the actors being verified. We will discuss such considerations in the next subsection.

5.3.1 Implementation Invariants

The specification of impure queue actors in Figure 5.2 is written in terms of the changes in their states before and after their invocations, and their states are expressed by conceptual representations of the following form.

(*IMPURE-QUEUE* [...])

When some program which contains invocations of impure queue actors is symbolically evaluated, conceptual representations of the above form are used only to record states of the impure queue actors. One need not pay attention to what those conceptual representations really stand for, as long as they represent the states of the impure queue actors at the conceptual level. However, when an implementation [script or code] of an impure queue actor Q itself is verified against its specification, what the conceptual representation expresses in terms of the implementation, or more precisely, how the state of Q expressed by the conceptual representation corresponds to the states of the constituents of its implementation, must be considered.

Suppose that the PLASMA implementation of an impure queue actor given in Figure 5.7 is to be verified against the contract in Figure 5.2. In this implementation, the elements of the queue are kept as the elements of a sequence actor that is the value of the variable **queuees**. This could be expressed by the diagram in Figure 5.8, where boxes represent actors and arrows express the know-about relations. This diagram is only a partial and static description of the implementation, yet it illustrates an invariant or

---

Fig. 5.7. A PLASMA Implementation of an Impure Queue Actor

```
(create-impure-queue ≡
 (≡> []                                   ;create-impure-queue receives an empty sequence.
  (let (queuees initially [])                         ;a variable queuees is declared
    then                                          ;and initialised with an empty sequence.
    (the-queue-itself ≡                       ;a queue-actor denoted by the-queue-itself is defined
                                                      ; by the cases-statement given below.
     (cases
      (≡> (nq: =new-element)               ;when an enqueue message with an element is received,
                                                      ;new-element is bound to the element.
       (queuees ← [!queuees new-element])          ;a new sequence-actor whose elements are
                                           ;the unpack of the value of queuees and new-element
                                                            ;is created and stored in queuees.
       the-queue-itself)                                 ;and then the-queue-itself is returned.

      (≡> (dq:)                                          ;when a dequeue message is received,
       (rules queuees                                                ;if the value of queuees
        (≡> [] (exhausted:) )                          ;is empty, then the message is returned.
        (≡> [=front !=rest]                  ;if it is a non-empty sequence, front and rest
                             ;are bound to its first element and the rest of its elements, respectively.
         (queuees ← rest)                                       ;the value of queuees is updated.
         (next: front (rest: the-queue-itself))   ) )) ))))               ;(next:...) is returned.
```

integrity condition which must be satisfied among constituents of the implementation. The following *implementation invariant* statement can express the diagram more formally.

```
<Implementation-Invariant:
    if  (the-queue-itself is-a (IMPURE-QUEUE [!a]))
          then
            (queuees has-value S)
            (S is-a (SEQUENCE [!a]))     >
```

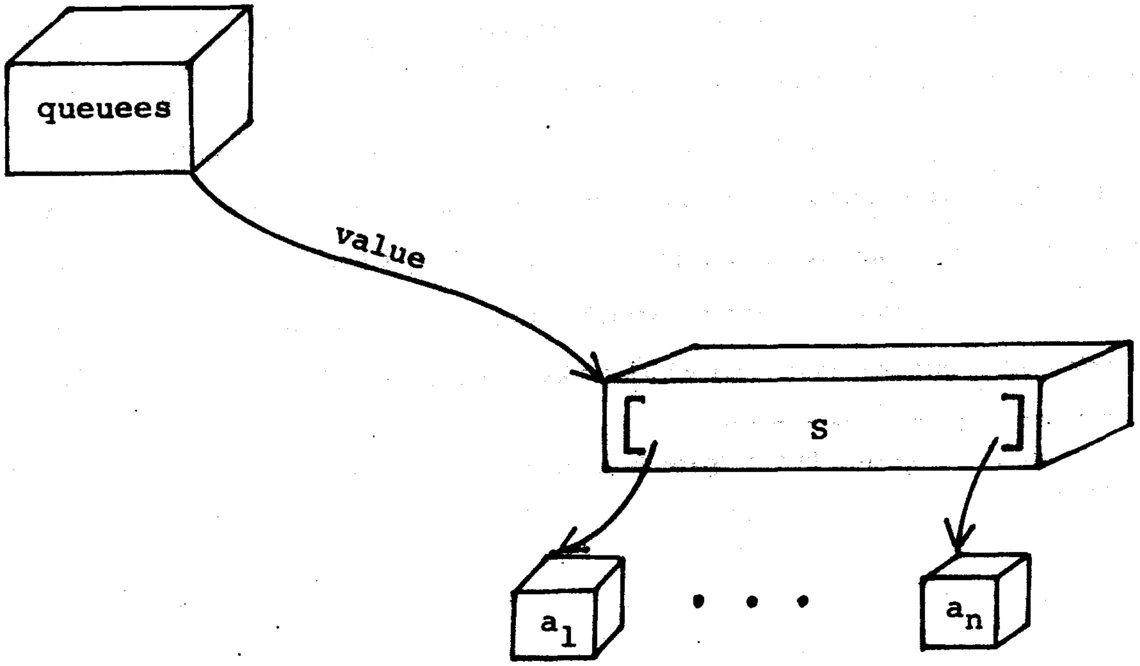
This says: when the state of the actor denoted by the-queue-itself is expressed by the conceptual representation

(*IMPURE-QUEUE* [!a]),

the variable queuees has the value which is always some sequence actor S whose elements are expressed by [!a]. (*SEQUENCE* [!a]) is the conceptual representation for such a sequence

---

Fig. 5.8.



actor.

An implementation invariant describes the <u>mapping</u> from the states of an actor (the "specification space") to the states of the constituents of a given implementation for the actor (the "implementation space").[1] Suppose that the behavior of an actor A is specified by the state of A before or after its invocation. Then an implementation invariant is used in the verification of A in the following way.

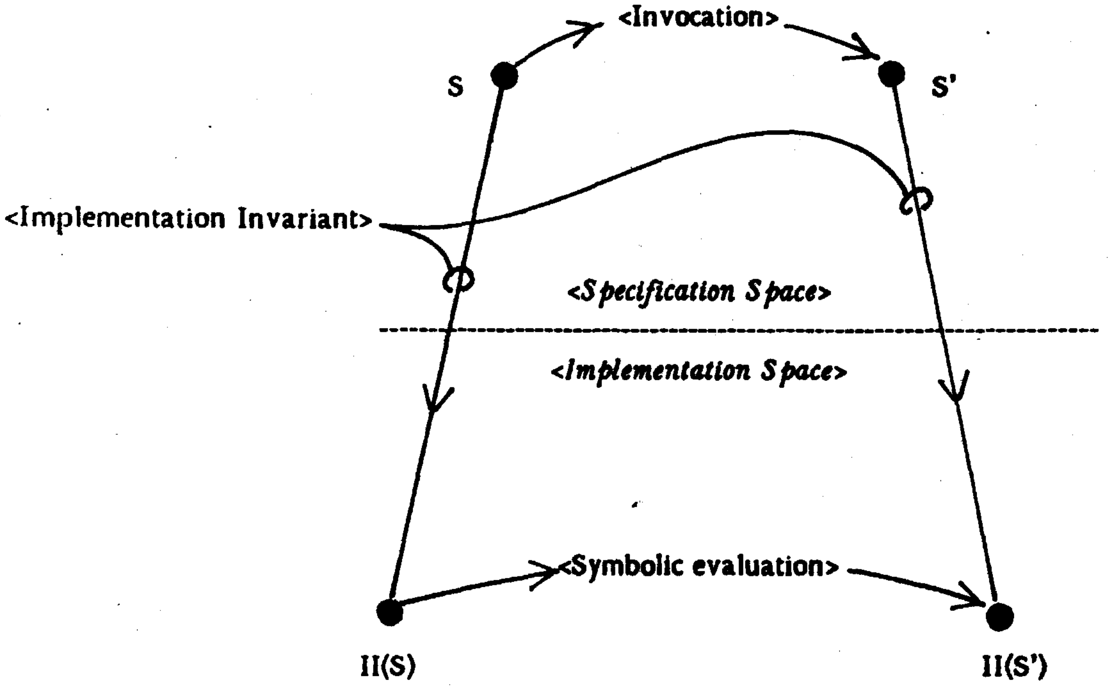
> First, the state S of A before the invocation is translated into the state II(S) of the constituents of the implementation by a given implementation invariant II. Then the implementation [code] is symbolically evaluated and the states of the constituents after the invocation are obtained. Next, by using the implementation invariant again, the state S' of A, specified as the one after the invocation, is translated into the state II(S') of the constituents. Finally, the states of the constituents obtained by the symbolic evaluation are checked to see if they satisfy those translated states. [See Figure 5.9.]

In general, given a state T of an actor A and an implementation I for A, an implementation invariant for I tells us the *relations* which must be satisfied by the states of the constituents of I to realize the state T. Therefore implementation invariants may be *one-to-many* mappings. In such a case, when symbolic evaluation of an implementation is started, only such relations (holding among the states of the constituents of an implementations) are assumed: exact states of each constituent are <u>not</u> used. An example of the one-to-many mapping cases is found in Section 7.4.2, Chapter 7. Implementation invariants are similar to the inverse of Hoare's *abstract functions* [Hoare72], and also serve as *concrete (representation) invariants* which be used additionally in proving correctness of

---

[1] A state in the implementation space is a vector of states of the constituents of the implementation.

Fig. 5.9. Verification of an actor A Behaving like Information Storage



---

representations of data structures. *Interpretation functions* between two formal theories studied by R. Nakajima [Nakajima-et-al77] seem closely related to implementation invariants.

5.3.2 Establishing Event Specifications

An implementation of an actor which behaves as "information storage" is verified by establishing each event specification associated with the actor. In this subsection, we will illustrate this by using an impure queue-actor as an example.

The verification of the implementation of an impure queue-actor is carried out by symbolic evaluation. To aid in the exposition of the symbolic evaluation, we augment the PLASMA code in Figure 5.7 with situational symbols as shown in Figure 5.10. This code is verified against the contract in Figure 5.2. Below we will establish the two *<event:...>* clauses in the contract, which specify the creation and enqueueing events. The dequeuing event can be established similarly.

Establishing the CREATION specification

In the first *<event:...>* clause in the contract in Figure 5.2:

```
<event: [[create-impure-queue <= []]
    <returns: Q*) >
    <post-cond: (Q is-a (IMPURE-QUEUE [])) >>,
```

there are no pre-conditions for this event. Thus no assertions are entered in the data base for the initial situation.

in $S_{\text{pre-creation}}$: <u>empty</u>

The *let* statement in the code declares and initializes a variable queuees with an empty sequence NS. To record this, the following assertions are entered.

in $S_{\text{initialized-queuees}}$:
(<u>queuees</u> *has-value* NS)
(NS *is-a* (*SEQUENCE* []))

Then in this situation an actor whose script (i.e. code) is given as the (cases...) statement after (the-queue-itself ≡ ... is newly created and returned. This actor is denoted by the-queue-itself. The contract for the creation requires two things: (1) that the returned

Fig. 5.10.

```
(create-impure-queue ≡
 (≡> []                                  ;create-impure-queue receives an empty sequence.
  (let (queuees initially [])                         ;a variable queuees is declared
     then                                      ;and initialized with an empty sequence.
  - S_initialized-queuees -
    (the-queue-itself ≡               ;a queue-actor denoted by the-queue-itself is defined
                                                   ;by the cases-statement given below.
      (cases
        (≡> (nq: =new-element)          ;when an enqueue message with an element is received,
                                                   ;new-element is bound to the element.
       - S_received-nq -
         (queuees ← [!queuees new-element])           ;a new sequence-actor whose elements
                                  ;are the unpack of the value of queuees and new-element
                                                   ;is created and is stored in queuees.
       - S_updated-queuees-nq -
         the-queue-itself)                           ;and then the-queue-itself is returned.

        (≡> (dq:)                                  ;when an dequeue message is received,
       - S_received-dq -
        (rules queuees                                         ;if the value of queuees
          (≡> []                                                 ;is an empty sequence,
       - S_empty-queuees -
           (exhausted:) )                          ;then the complaint message is returned.

          (≡> [=front !=rest]            ;if it is a non-empty sequence, front and rest
                     ;are bound to its first element and the rest of its elements, respectively.
       - S_non-empty-queuees -
           (queuees ← rest)                               ;the value of queuees is updated.
       - S_updated-queuees-dq -
           (dequeued: front (rest: the-queue-itself)) ) )) ))  ))       ;(next:...) is returned.
```

actor Q be newly created and (2) that (Q *is-a* (*IMPURE-QUEUE* [])) holds. Since the returned actor is the-queue-itself, what we need to show is that (the-queue-itself *is-a* (*IMPURE-QUEUE* [])) holds. This assertion is translated into the following assertions using the assertions in the *where*-clause in the implementation invariant statement given in the previous subsection. [Note that the assertions in the *where*-clause are instantiated by substituting an empty sequence [] for !a.]

(queuees *has-value* S)
(S *is-a* (*SEQUENCE* []))

These two assertions are matched against the two assertions entered at the node for $\mathbf{S}_{\text{initialized-queuees}}$. Therefore it is concluded that the returned actor the-queue-itself has the correct internal structure prescribed by the implementation invariant. So the result of the event **[** create-impure-queue <= [] **]** meets its specification.

Establishing the ENQUEUING specification

From the instantiation of the event specification for enqueueing:

<*event:* **[** the-queue-itself <= (*nq:* A) **]**
<*pre-cond:* (the-queue-itself *is-a* (*IMPURE-QUEUE* [!x])) >
<*returns:* the-queue-itself >
<*post-cond:* (the-queue-itself *is-a* (*IMPURE-QUEUE* [!x A])) >>

which is obtained by substituting the-queue-itself for Q in the contract for (*IMPURE-QUEUE* [...]) in Figure 5.2, it is assumed that

(the-queue-itself *is-a* (*IMPURE-QUEUE* [!x]))

holds in the initial situation. By the implementation invariant statement, this assumption is translated into the following two assertions: [Note that x is substituted for a in the invariant statement.]

> *in* $\mathbf{S}_{\text{initialized-queuees}}$:
> (queuees *has-value* S))
> (S *is-a* (*SEQUENCE* [!x]))

Now the message (*nq:* A) is sent to the-queue-itself. This message matches against the first clause of the case statement. So new-element is bound to A.

> *in* $\mathbf{S}_{\text{received-nq}}$: (new-element ≡ A)

Then the value of queuees is updated by a newly created sequence-actor NS with its elements [!queuees new-element]. The value of queuees in $\mathbf{S}_{\text{received-nq}}$ is obtained by inheriting from $\mathbf{S}_{\text{initialized-queuees}}$, because no updating events took place between the two situations. Thus the value is a sequence-actor S. !queuees is the result of the unpack operation on S, which is !x. [Note that the sequence actor is pure. Therefore its state can be inherited from $\mathbf{S}_{\text{initialized-queuees}}$.] So the state of the new sequence-actor NS is expressed by (*SEQUENCE* [!x A]). For the assignment of NS to queuees, the new assertion (queuees *has-value* NS) is entered in the data base. So the following assertions hold in the next situation.

> *in* $\mathbf{S}_{\text{updated-queuees-nq}}$:
> (queuees *has-value* NS)
> (NS *is-a* (*SEQUENCE* [!x A]))

The code tells us that the-queue-itself is returned in this situation. The specification for the enqueuing requires that the-queue-itself be returned and that

(the-queue-itself *is-a* (*IMPURE-QUEUE* [!x A])).

So this assertion is translated into the following assertions by the implementation invariant.

(queuees *has-value* S)
(S *is-a* (*SEQUENCE* [!x A]))

These assertions are obviously matched against the assertions entered at the node for $\mathbf{S}_{\text{updated-queuees-nq}}$. So the enqueuing event meets its specification.

5.4 Discussions Related to Symbolic Evaluation

The method of symbolic evaluation presented in this chapter has many interesting facets and significant implications for other research areas besides program verification. In this section, we first reflect on our approach to verification based on symbolic evaluation in the light of other existing approaches. We then discuss the applications of symbolic evaluation. Finally, our reasoning method employed in symbolic evaluation will be discussed in the context of McCarthy's frame problem.

5.4.1 Situational Descriptions vs. Predicate Transformations

Program verification methods based on the Floyd-Hoare proof rules [Floyd67, Hoare69] or predicate transformers [Manna69, Dijkstra76] can be summarized as follows:

> Given a set of predicates P holding in a situation $\mathbf{S}$, the proof rules or the predicate transformer generate a set of predicates P' [from P] which hold in the next[1] situation

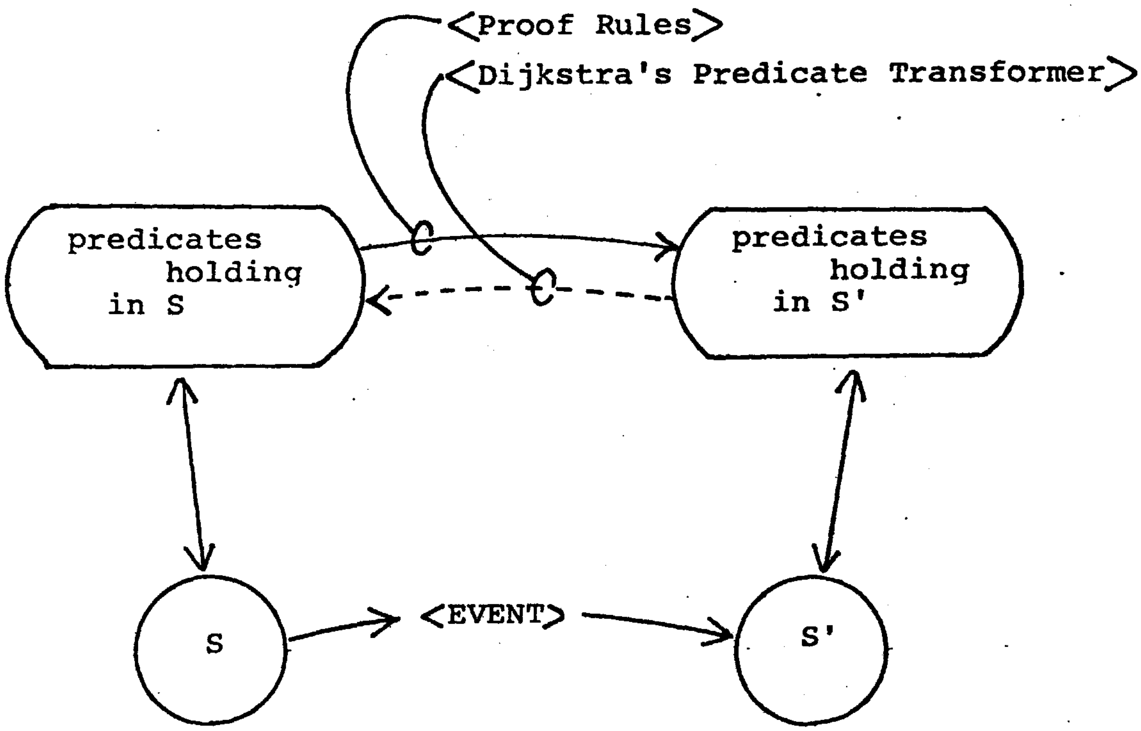
---

[1] For the case of the proof rules, the next situation is the temporal successor situation, and for Dijkstra's predicate transformers, it is the predecessor situation.

S'.

The choice of predicates holding in S determines the generated set of predicates for S'. Those choices are made so that desired assertions may be shown to hold in S'. This approach is schematically described in Figure 5.11. Note that the predicate transformers work backwards.

Fig. 5.11. Floyd-Hoare-Dijkstra Predicate Transformation Approach



In contrast to the approach above, our approach is:

> Given a *description* D of a situation **S**, symbolic evaluation produces a *description* D' of the [forwardly] next situation by using contracts and trans-situational rules.

A description of a situation is a collection of assertions about states of actors which are expressed by conceptual representations. Predicates which hold in a situation are <u>derived</u> from the description of the situation. This approach, which we call the "situational description" approach, is schematically described in Figure 5.12.

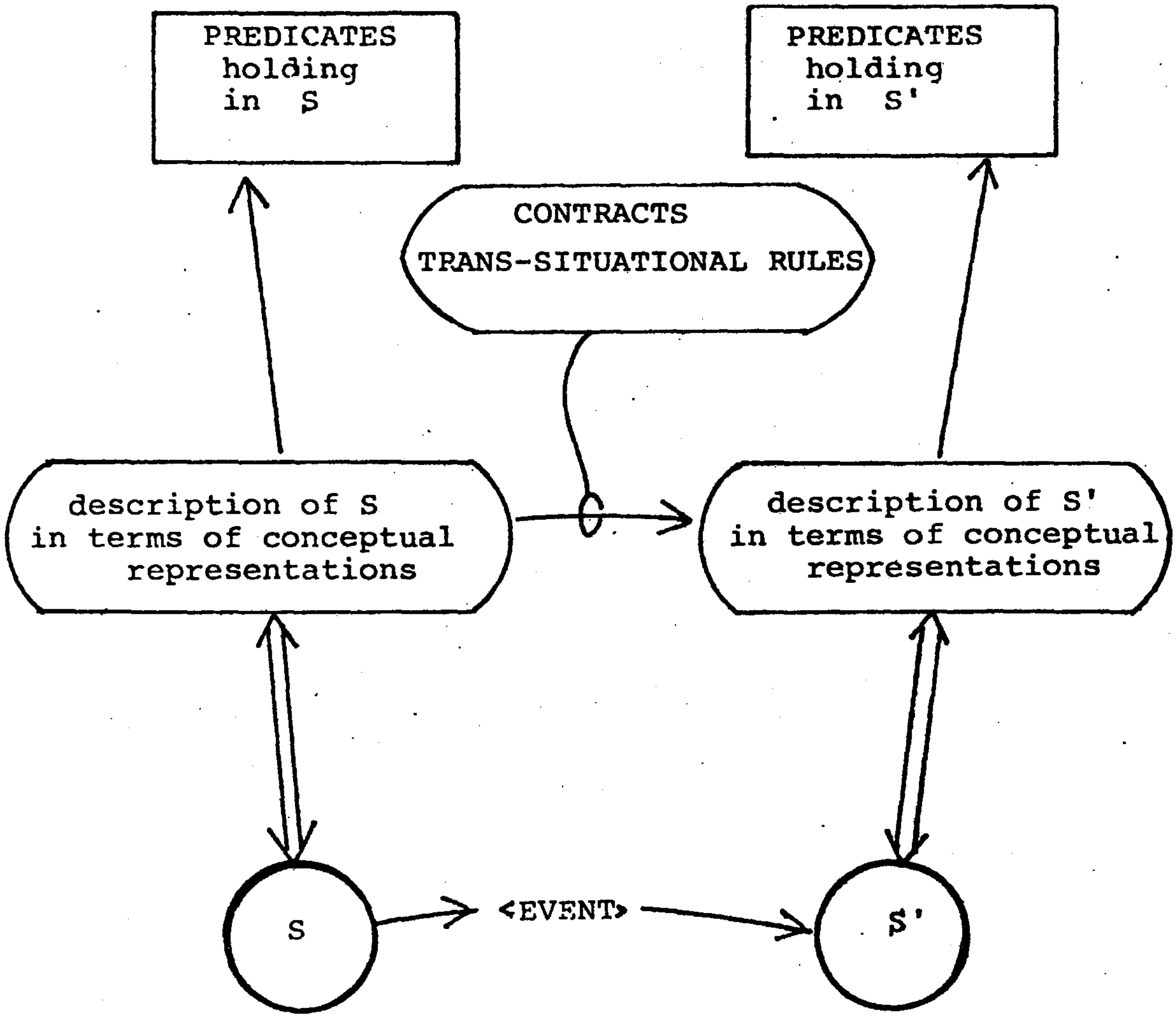
Conceptual representations not only express states of individual actors in a system, but they can also describe how the individual actors are interrelated at various levels. Thus the description of situations in terms of conceptual representations is powerful in dealing with sharing. Furthermore, descriptions of each situation provide us with sources of various information about a program, which is quite useful for other applications in the areas of mechanical program analysis.

5.4.2 Applications of Symbolic Evaluation

Symbolic evaluation based on formalisms different from ours has been studied for various purposes such as proving properties of programs [Boyer-Moore75], program testing and debugging [Boyer-et-al75, King76], program transformation and improvement [Burstall-Darlington75] etc.

Our method of symbolic evaluation can be used in constructing a software system called a Programming Apprentice [Hewitt-Smith75, Rich-Shrobe76], which aids expert programmers in various aspects of programming activities such as verification, debugging, and refinement of programs. In the Programming Apprentice, the purpose of symbolic

Fig. 5.12. The Situational Description Approach


PREDICATES holding in S
PREDICATES holding in S'
CONTRACTS
TRANS-SITUATIONAL RULES
description of S in terms of conceptual representations
description of S' in terms of conceptual representations
S
<EVENT>
S'

evaluation is not simply to verify programs against their specifications. By symbolic evaluation, we try to gather information about dependencies between program modules. Such information is used to understand implications of proposed changes in both specifications and implementations in the subsequent evolutional development of the programs.

For instance, suppose that the implementation of **empty-one-queue-into-another** used as an example of program verification is sent pure queue actors instead of impure queue actors. Using the contracts for pure queue actors in Figure 4.2 in Chapter 4, our method of symbolic evaluation can easily trace and record the behavior of the implementation. The situational tree produced during the symbolic evaluation aids us in modifying the implementation so that it may accept both impure and pure queue actors. Another simple example might be the analysis of the behavior of the same implementation when it is sent the same impure actor. [That is, one of the preconditions, (Q1 ***not-eq*** Q2) is forgotten.] Furthermore, as reported in [Yonezawa-Hewitt76], the efficiency of the implementation of impure queue actors in terms of consumed storage can be revealed by using assertions of the form

(<actor-1> ***knows-about*** <actor2>)

in the process of symbolic evaluation.

The situational description approach based on our method of symbolic evaluation appears to be quite powerful in pursuing these ends. The symbolic evaluator in C. Rich and H. Shrobe's system [Rich-Shrobe76], which understands LISP programs, is based on a method similar to ours.

5.4.3 The Frame Problem

In the context of Artificial Intelligence, J. McCarthy and P. Hayes [McCarthy-Hayes69] pointed out a problem, called the frame problem, which arises in formalizing effects of actions or events taking place in a complex world. A typical example of the frame problem is found in formalizing the effects of actions of a robot in a block world where the robot carries out various physical tasks. Suppose that the robot has moved a block B to a certain location. With this action, the location of B changes, but most of the properties of the blocks, such as color, height, and volume, and relations holding among other blocks, do not change. To formalize the action "move", it is necessary to specify not only which of these properties and relations will change [and how they will change], but also which properties or relations will not change. Since the robot is supposed to perform a number of different actions, for each action such changes in properties and relations in both positive and negative sense must be specified. In most cases, rather a small number of properties and relations change as the result of a single action, while the rest of them do not. Thus the number of such specifications will be unbearably large for a practical system if the tasks of the robot and the world in which it works become complicated.

The same problem arises in the context of program specification and verification. In particular, the frame problem becomes serious when one tries to construct program verification or understanding systems which must deal with actors whose behavior may change with time. To specify the effects of computations [or events], the no-changes as well as the changes in the states of objects in a system must be described even if the objects do not participate in the computations. If we described the changes and no-changes of all the objects in the system in a straightforward way, the same serious problem would arise.

As presented in the first section in this chapter [5.1.2, 5.1.3], we take a procedural approach to this problem. Our reasoning method based on trans-situational rules is powerful in coping with the problem in the domain of Artificial Intelligence as well. R. Waldinger has independently proposed an approach similar to ours for dealing with certain issues in program synthesis and has discussed its application to Artificial Intelligence [Waldinger77]. Those who are interested in comparative studies of the existing approaches to the frame problem should see [Sandwall72, Hayes73, Hewitt75, Waldinger77].

6. Specifying Parallel Computations

In this chapter, the specification language introduced in Chapter 4 is extended to cover parallel computation. Formal specifications of abstract data type objects which are used in *multi-process environments* are written in the extended language. Examples for illustrating our specification techniques include air line reservation systems and bounded buffers. An alternative definition of states of actor (objects) is discussed at the end of the chapter.

6.1 Introduction

In this section, we will discuss the characterisitics of parallel computation which make its specification method different from that for serial computation. Our specification techniques for parallel computations will be described in the subsequent sections of this chapter.

6.1.1 Communicating Parallel Processes

In a serial computation, activations of actors take place sequentially and one at a time. Thus it is modelled as a set of linear ordered events with each event causally related to one another. [Recall the definition of computations in Chapter 3.] In a parallel computation, however, more than one activation may take place concurrently. Some events are causally related to each other, but some may not be. Therefore, a computation is modelled as a set of partially ordered events. A sequence of causally related events can be viewed as a "process". From this view point, parallel computations involve multiple processes and serial computations a single process.

If, in a parallel computation, concurrent processes do not interact with each other, i.e., no events are causally related between processes, the computation can be viewed as a collection of mutually independent serial computations.

However, there are many reasons for the necessity of interaction between concurrently running processes: If arguments in a procedure call are evaluated in parallel, a process which executes the procedure body must wait until all the parallel evaluations of the arguments are completed. In air line reservation systems and inventory control systems, concurrent processes interact with each other by retrieving and updating various information in data bases. In operating systems, concurrent processes interact through sharing resources such as main/secondary memories and I/O peripherals.

In order for such interactions [or cooperations] to be effective and efficient, concurrent processes must communicate and synchronize with each other. Therefore in specifying interesting behaviors of parallel computations, we need techniques which are able to deal with communication and synchronization between processes. In our model of computation, such communication and synchronization is realized by changing states of certain actors. [Cells, buffers and data bases are examples of such actors.] Therefore the central issue in the method for specification of parallel computations is to deal with the behavior of actors which are used for communication and synchronization.

States of actors are extensively used in specifying parallel computations as well as serial computations. But states of actors in parallel computations [or multi-processor environments] need to be dealt with much more carefully than those in serial computations. We will discuss this issue in detail in the next subsection.

6.1.2 Local States

In describing behaviors of parallel computations, there have been many attempts[Milner73, Kahn74, Ashcroft75, Cohen75, Owicki75, Keller76, Owicki-Gries76, Flon-Suzuki77, Lamport77] to use the notion of the global states of an entire system. The global state of a system at a given time is expressed essentially by a vector of states of the subsystems. The use of the global states is often motivated by the use of *non-deterministic* serial computations for the semantic model for parallel computations. In order to study properties of a subsystem, this approach leads to counter-intuitive serialization of concurrent events taking place in unrelated subsystems and it forces us to consider not only changes in other subsystems but also the order in which such changes take place. Thus the number of cases to be examined tends to be exponentially large, but almost all changes in other subsystems are irrelevant to the subsystem under consideration.

In our approach, we do not rely on such notions as the global state and the global clock [uniform time reference]. Rather we take a local and relativistic view. We assume only the local states of individual actors. [Cf. Section 1.3, Chapter 1.] The local state of an actor is determined only at the local time associated with the actor. Thus, when the state of a computer at some site of a computer network is determined, we do not assume that the states of computers at other sites can be defined. The state of an actor is determined at the time when the actor receives a message. This timing is particularly important and useful in parallel computations because it is a well defined moment in a distributed system. [The moments of message transmission at scattered computer sites are difficult to compare with each other.] Recall that the ordering of arrival of messages with respect to a given actor [arrival subordering] is total in our model of computation. [Cf. Section 3.1.3, Chapter 3]

In Section 4.1.1, Chapter 4, we have defined states of an actor as equivalence classes of past histories of messages sent to the actor. As discussed before, this definition subsumes, in serial computations, traditional definitions for data-storing objects, whose states are determined by their current information content. Such traditional definitions are inadequate in parallel computations [or multi-process environments]. For example, imagine a data base system which is concurrently accessed by a number of users. If the state of the data base were defined as its stored data, its state at the time of the arrival of an access request could not be determined, because the stored information might be being changed by previously arrived requests. Also determining the information content inside the data base at the time when a request arrives at the data base is incompatible with our relativistic view introduced above. [Imagine a data base system where an access request may be received by a computer site located at one side of the continent while actual data may stored at the other side.]

States of an actor defined as equivalence classes of the past message histories are not affected by the actual activations of the actor. Also the order of arrival of messages is linear (total). These two facts are essential to our specification techniques for parallel computations because they guarantee that states thus defined are always well defined even if the actor is being activated by the previously arrived messages. In the later sections, examples that illustrate the significance of our state definition will be found. In particular, a model of interaction between a post office and customers in Chapter 8 will provide an intuitive example.

6.2 Extending the Specification Language

Specifications of the behavior of actors in parallel computations are written in a way similar to that in which the behavior of actors in serial computations is specified. That is, when given the state of an actor, the behavior of the actor is specified by the resulting state changes and the subsequently caused events. However, the major difference lies in how the states of actors change and how such changes are expressed. To distinguish such difference, the specification language introduced for serial computations in chapter 4 needs to be extended.

6.2.1 Instantaneous State Changes

Let us try to write a formal specification of a cell actor. A cell actor is used to store information. It accepts updating messages of the form (*update:* **<new-contents>**) and retrieving messages of the form (*contents:*). Its behavior is expressed informally as follows:

"In response to a (*contents:*) message,
  a cell actor returns **<contents>** which was contained
    in the most recently arrived (*update:...*) message if such a message exists,
  otherwise it returns its initial contents"

We would like to express this behavior by using the states of the cell.[1] To express a state of a cell actor, we use conceptual representations. For example,

(*CELL* (*contents:* A))

expresses the state which is defined as a class of histories of messages whose most recent updating message is of the form (*update:* A). If the cell were used only in serial computations, we could specify this behavior by the following two event specifications:

```
<event: ⟦C <= (contents:)⟧
   <pre-cond:  (C is-a (CELL (contents: A))) >
   <return: A >
   <post-cond:  (C is-a (CELL (contents: A))) > >

<event: ⟦C <= (update: B)⟧
   <pre-cond:  (C is-a (CELL (contents: A))) >
   <return: B >
   <post-cond:  (C is-a (CELL (contents: B))) > >
```

Unfortunately, the above event specifications do not precisely express the behavior of a cell in parallel computations, because the states of C expressed in the *<post-cond:...>* clauses are

---

[1] I. Greif and C. Hewitt gave a specification of cells which is expressed by axioms about events in [Greif-Hewitt75, Greif75].

the states at the time A or B are returned, but the state of the cell may be changed by the updating messages subsequently arriving before A or B are returned.

In order to eliminate this impreciseness in the above event specifications, the following two points should be made clear. First, states of a cell expressed by the conceptual representations must be interpreted strictly in terms of equivalence classes of histories of incoming messages. They should not be interpreted to express the current contents of the cell. The second point, which logically follows from the first one, is that in order to be consistent with the definition of the states expressed by the conceptual representations, the state of the cell must change instantaneously when an (*update:...*) message arrives.

In general, in specifying behaviors of actors in parallel computations through their state changes, the fact that states change instantaneously must be taken into account.

6.2.2 <*Next-cond:...*> Clauses

To express the instantaneous state changes in specifications, we introduce a new specification language construct, <*next-cond:...*> clauses. This is usually used in event specifications of the following form.

```
<event: ⟦T <== M⟧
   <pre-cond:        ...          >
   <next-cond:  ... <assertion>... >
   <caused-event:  E  >>
```

This means: when an event ⟦T <== M⟧ takes places, if the preconditions are satisfied, the <assertion>s in the <*next-cond: ...*> clause hold immediately after the event ⟦T <== M⟧ and continue to hold at least until one of the actors appearing in the <assertion>s receives the next message. For example, if the <assertion>s mention T or M, they continue to hold at least until T or M receives its next message. The assertions in the *<next-cond:...>* clause can be viewed as the preconditions for the next event. A *<next-cond:...>* clause differs from a *<post-cond:...>* clause in that assertions in the *<post-cond:...>* clause hold at the time the corresponding caused event take place, but may not hold before the caused event. When a *<next-cond:...>* clause is used in specifying serial computations, its meaning is identical to that for a *<post-cond:...>* clause. The event E in the *<caused-event:...>* clause must take place eventually. It is often the case that concurrent events are caused by ⟦T <== M⟧. In such a case, we use clauses of the form *<caused-events:* {...<event>...}>. Other interpretation rules for event specifications, such as those for absent clauses, abbreviated forms and scope rules for symbols in clauses are the same as for serial computations. [Cf. Sections 4.3.1 and 4.3.3, Chapter 4]

Using this new construct, a specification of the behavior of a cell in parallel

---

Fig. 6.1. A Specification of a Cell

```
<event: ⟦create-cell <= A⟧
   <return: C* >
   <post-cond: (C is-a (CELL (contents: A))) >>

<event: ⟦C <= (contents:)⟧
   <pre-cond:  (C is-a (CELL (contents: A))) >
   <next-cond:  (C is-a (CELL (contents: A))) >
   <return: A >>

<event: ⟦C <= (update: B)⟧
   <pre-cond:  (C is-a (CELL (contents: A))) >
   <next-cond:  (C is-a (CELL (contents: B))) >
   <return: B >>
```

computations is written as depicted in Figure 6.1. *<Return:...>* clauses are used as an abbreviated form of a *<caused-event:...>* clause. When a cell actor is created by the create-cell actor receiving the initial contents, we need not use a *<next-cond:...>* clause in expressing the state of the newly created cell, because before the new cell is released nothing can happen to change the state of the cell. It should be pointed out that the equivalence relation defining the states of a cell (which are expressed by conceptual representations) is expressed *incrementally* by the *<pre-cond:...>* and *<next-cond:...>* clauses in the specification in Figure 6.1.

6.3 Examples of Specifications

In this section, we will discuss three specifications as examples. The first example is a specification of a simple air line reservation system. This example illustrates how the behavior of systems which process requests on a first-come-first-served basis is specified by our technique. In the second example [a specification of semaphores], we will see how processes which have requested some actor for resource usages that have not yet been granted are dealt with in expressing the state of the actor. The third example is a completely external [i.e. implementation independent] specification of a bounded buffer which requires us to express "non-first-come-first-served" scheduling of requests.

As was mentioned before, an actor model of a simple post office is studied in Chapter 8. It is shown that overall task specifications of the post office can be derived by specifications of the individual behavior and mutual interaction of actors in the model.

6.3.1 Modelling an Air Line Reservation System

As an example, let us consider an air line reservation system. For the sake of simplicity, we assume that only one flight is available in the system. A number of travel agencies [parallel processes] try to reserve or cancel seats for the flight concurrently. We model[1] the air line reservation system as a flight actor F which behaves as follows. The flight actor F accepts two kinds of messages,

(*reserve-a-seat:* <passenger-name>) and (*cancel-a-seat:* <passenger-name>).

When F receives (*reserve-a-seat:...*), if free seats are left, the passenger name is appended to the passenger name list for the flight and the number of free seats is decreased by one, and a message (*ok-its-reserved:*) is returned. Otherwise a message (*no-more-seats:*) is returned. When F receives (*cancel-a-seat:...*), if the passenger name is found in the passenger name list, a message (*ok-its-cancelled:*) is returned and the passenger name is deleted from the passenger name list and the number of free seats is increased by one. Otherwise a message (*the-passenger-name-not-found:*) is returned. Furthermore requests by (*reserve-a-seat:...*) and (*cancel-a-seat:...*) are processed on a first-come-first-served basis.

To write a formal specification of the air line reservation system, we need to describe the states of the flight actor. For this purpose, we use the following conceptual representation

(*FLIGHT* (*seats-free:* <m>) (*passenger-name-list:* {!pnl}))

which describes the state of a flight actor. The number of free seats is <m> and {!pnl} is

---

[1] E. A. Ashcroft[1975] gave a flowchart program which models an air line reservation system. In his program, each user (or agency) has its own copy of the request handling program and all the copies are connected with a single *fork* operation. Furthermore, the number of users must be fixed.

the passenger name list for the flight The formal specification of the air line reservation system using this conceptual representation is depicted in Figure 6.2.

Since the states expressed by conceptual representations in the specification are defined as equivalence classes of histories of messages sent to F, the number of free seats and the passenger name list given in the conceptual representations does not necessarily correspond to those that are actually stored in the system.[1] From the view point of a message arriving at F, the states expressed by conceptual representations in <*pre-cond:...*> clauses are virtual. That is to say, those conceptual representations express the information that will be true after all the messages previously arrived at F are processed, although currently some of those messages may be being processed or some may even be suspended in the request queue. Therefore, only air line reservation systems in which the reserve and cancel requests are processed on a first-come-first-served basis satisfy the specification in Figure 6.2.

It is easy to specify the behavior of air line reservation systems which deal with more than one flight and can add and remove flights. To do so, one may use conceptual representations which express the flight information for each flight. For example,

(*RESERVATION-SYSTEM* {...(*flight-i:* (*seats-free:* <n>)(*passenger-name-list:* {!pnl})) ...})

may suffice. In this case, the reservation system thus specified processes the reserve and cancel requests on a flight-wise first-come-first-served basis. This implies that requests for different flights may not be processed on a first-come-first-served basis. The technique to specify the flight-wise first-come-first-served processing can be applied in specifying file

---

[1] If the processing of requests were so fast that each request might be processed before the next one arrives, the information expressed in the conceptual representations would correspond to what is actually stored in the system.

Fig. 6.2. A Specification of an Air Line Reservation System

```
<event: ⟦create-flight <= S⟧
    <pre-cond: (S > 0) >
    <return: F* >
    <post-cond:  (F is-a (FLIGHT (seats-free: S) (passener-name-list: {})))>>

<event: ⟦F <= (reserve-a-seat: NAME)⟧
  (case-1:
    <pre-cond:  (F is-a (FLIGHT (seats-free: 0) (passenger-name-list: {!pnl})))>
    <next-cond:  (F is-a (FLIGHT (seats-free: 0) (passenger-name-list: {!pnl})))>
    <return:  (no-more-seats:) >)
  (case-2:
    <pre-cond:
      (F is-a (FLIGHT (seats-free: N) (passenger-name-list: {!pnl})))
      (N > 0) >
    <next-cond: (F is-a (FLIGHT (seats-free: N - 1) (passenger-name-list: {!pnl NAME})))>
    <return:  (ok-its-reserved:) >)>

<event: ⟦F <= (cancel-a-seat: NAME)⟧
  (case-1:
    <pre-cond:
        (F is-a (FLIGHT (seats-free: N) (passenger-name-list: {!pnl})))
        (pnl ≠ {... NAME ...})>
    <next-cond:  (F is-a (FLIGHT (seats-free: N) (passenger-name-list: {!pnl})))>
    <return:  (the-passenger-name-not-found:) >)
  (case-2:
    <pre-cond:
      (F is-a (FLIGHT (seats-free: N) (passenger-name-list: {!pnl1 NAME !pnl2})))>
    <next-cond: (F is-a (FLIGHT (seats-free: N + 1) (passenger-name-list: {!pnl1 !pnl2})))>
    <return:  (ok-its-cancelled:) > )>
```

systems, large data base systems, and disk-head scheduling systems [Hoare74] as long as individual files and disk tracks are used on a first-come-first-served basis.

6.3.2 A Specification of Semaphores

The behavior of semaphores can be easily specified by our techniques. The state of a semaphore is described by conceptual representations of the following form.

(*SEMAPHORE* **(*counter:* <n>) (*waiting-q:* [!q]))**

where <n> is the number of processes that can still enter the critical section it guards and [!q] is the queue of processes waiting to enter the critical section. A specification of a semaphore is depicted in Figure 6.3.

A message sent to a semaphore consists of a request [i.e., either P-operation or V-operation], and a continuation actor which will be activated when the request to the semaphore is granted. The continuation can be viewed as a process that will be awakened. As stated in the *Case-2* of the second event specification [for P-operation], when the counter is zero, no message is sent to the continuation. Hence the <*caused-event:...*> clause has no events. In the *Case-1* of the third event specification [for V-operation], two events, ⟦C <= (*go-ahead:*)⟧ and ⟦first <= (*go-ahead:*)⟧ are caused concurrently.

Fig. 6.3. A Specification of Semaphores

```
<event: ⟦create-semaphore <= N⟧
    <pre-cond: (N ≥ 0) >
    <return:  S* >
    <post-cond:  (S is-a (SEMAPHORE (counter: N) (waiting-q: []))) >>

<event: ⟦S <== [request: (P-op:), reply-to: C]⟧
   (Case-1:
    <pre-cond:
        (S is-a (SEMAPHORE (counter: N) (waiting-q: [])) )
        (N > 0)  >
    <next-cond:  (S is-a (SEMAPHORE (counter: N - 1) (waiting-q: []))) >
    <caused-event:  ⟦C <= (go-ahead:)⟧ >)
   (Case-2:
    <pre-cond:  (S is-a (SEMAPHORE (counter: 0) (waiting-q: [!q]))) >
    <next-cond:  (S is-a (SEMAPHORE (counter: 0) (waiting-q: [!q C]))) >
    <caused-events: {} >   ) >

<event: ⟦S <== [request: (V-op:), reply-to: C]⟧
   (Case-1:
    <pre-cond:  (S is-a (SEMAPHORE (counter: 0) (waiting-q: [first !rest]))) >
    <next-cond:  (S is-a (SEMAPHORE (counter: 0) (waiting-q: [!rest]))) >
    <caused-events: {⟦C <= (go-ahead:)⟧, ⟦first <= (go-ahead:)⟧  } >)
   (Case-2:
    <pre-cond: (S is-a (SEMAPHORE (counter: N) (waiting-q: [])))>
    <next-cond:  (S is-a (SEMAPHORE (counter: N + 1) (waiting-q: []))) >
    <caused-event: ⟦C <= (go-ahead:)⟧ >   )>
```

6.3.3 A Specification of a Bounded Buffer

As a simple example of specifications for actors which do scheduling of incoming requests, we specify a desirable behavior of a character buffer of a fixed size N with which concurrent processes communicate to one another.

A buffer actor B accepts two kinds of requests, (*remove:*) and (*append:* <character>), and it can hold at most N characters. Characters are appended or removed from the buffer on a first-in-first-out basis. But requests are not necessarily granted on a first-come-first-served basis, because a character should be appended only when the buffer is not full and it should be removed only when the buffer is not empty. This implies that when the buffer is empty, (*remove:*) requests must be suspended until the buffer becomes non-empty by an (*append:...*) request arriving later. Similarly, when the buffer is full, (*append:...*) requests must be suspended until the buffer becomes non-full. Therefore, in determining external states of the buffer, we must take into account such suspended requests (waiting processes).

To express the states of the buffer, we use conceptual representations of the following form.

$$(\mathit{BOUNDED\text{-}BUFFER}\ (q_a\text{: }[\ldots])(q_r\text{: }[\ldots])(\mathit{string}\text{: }[\ldots]))$$

q_a and q_r denote queues of suspended messages for (*append:...*) and (*remove:*) requests, respectively. *String* denotes the string storage used as a buffer. [Remember that the states expressed by the conceptual representations are defined in terms of the equivalence classes of the past message histories. So q_a, q_r and *string* do not necessarily correspond to the queues of requests which are actually suspended or the string of characters which are actually stored.]

In figures 6.4 and 6.5, we give a specification for the behavior of this bounded buffer. The first event specification in Figure 6.4 describes how the buffer is created. Note that the two queues q_a and q_r as well as the string storage are empty when the buffer is created.

The second event specification in Figure 6.4 describes the behavior of the buffer in response to a message M for a *(remove:)* request. Note that the message M explicitly contains a continuation C. There are three cases depending upon the state of the buffer B at the time when the message M arrives. *Case-1* is the one in which the string storage is empty, and no messages for *(append:...)* requests are suspended [i.e., q_a = []], and messages

---

Fig. 6.4. A Specification of a Bounded Buffer of Size N (Creation and Removing a Character)

```
<event: [[create-bounded-buffer <= []]]
   <return: $B^{*}$>
   <post-cond: (B is-a (BOUNDED-BUFFER ($q_a$: [])($q_r$: [])(string: []))) >>

<event: [[B <= M]]
        where  M = [request: (remove:) reply-to: C]
  (Case-1:
     <pre-cond: (B is-a (BOUNDED-BUFFER ($q_a$: [])($q_r$: [!y])(string: []))) >
     <next-cond: (B is-a (BOUNDED-BUFFER ($q_a$: [])($q_r$: [!y M])(string: []))) >
     <caused-events: {} >)
  (Case-2:
     <pre-cond: (B is-a (BOUNDED-BUFFER ($q_a$: [])($q_r$: [])(string: [X !s]))) >
     <next-cond: (B is-a (BOUNDED-BUFFER ($q_a$: [])($q_r$: [])(string: [!s]))) >
     <caused-event: [[C <= (removed: X)]] >)
  (Case-3:
     <pre-cond:
        (B is-a (BOUNDED-BUFFER ($q_a$: [MM !x])($q_r$: [])(string: [X !s])))
        (length([X !s]) = N)
        (MM = [request: (append: XX) reply-to: CC]) >
     <next-cond: (B is-a (BOUNDED-BUFFER ($q_a$: [!x])($q_r$: [])(string: [!s XX]))) >
     <caused-events: {[[C <= (removed: X)]], [[CC <= (append-done:)]]} >) >
```

for (*remove:*) requests may or may not be suspended, [i.e., q_r = [!y][1]]. In this case, the message M is enqueued at the end of q_r and no events are caused. When the string storage is not empty and both q_r and q_a are empty (*Case-2*), the first character X in the string storage is deleted and sent back to the continuation C as a reply message (*removed:* X). *Case-3* is the one in which the string storage is full [i.e., length([X !s]) = N], at least one message for an (*append:...*) request is suspended [i.e., q_a = [MM !x]] and no messages for (*remove:*) requests are suspended. In this case, the following change in the state of B happens: the first element MM in q_a, which is of the form [*request:* (*append:* XX) *reply-to:* CC], is deleted from the queue, the character XX is added at the end of the string storage, and the first character X in the string storage is deleted. Then, two events are caused concurrently: ⟦C <= (*removed:* X)⟧ where X is sent to the continuation C and ⟦CC <= (*append-done:*)⟧ where the acknowledging message for the message MM for an (*append:...*) request is sent to the continuation CC. (Cf. the remarks below.)

The behavior of the buffer in response to messages for (*append :...*) requests is described by the event specifications given in Figure 6.5. This event specification and the one for (*remove:*) requests in Figure 6.4 are symmetrical: By exchanging the roles of q_a and q_r and the conditions expressing the upper bound and lower bound of the length of the buffer, one is obtained from the other.

It should be pointed out that the six cases for the state of the buffer considered in the event specifications in Figure 6.4 and 6.5 are mutually exclusive and enumerate all cases of the states which the buffer can be in if it is created with q_r, q_a, and the string storage

---

[1] Recall that [!y] can be an empty conceptual sequence. Cf. Sections 2.2.3 and 2.3.5, in Chapter 2.

empty. One should be reminded that the states of the buffer are defined in terms of equivalence classes of past histories of messages sent to it and that the state changes described in the specification are instantaneous as they are expressed by assertions in the *<next-cond:...>* clauses. Thus, q_r can be non-empty only if *string* is empty and q_a can be non-empty only if *string* is full, and consequently, q_r and q_a cannot be non-empty at the same time.

From the specification given in Figures 6.4 and 6.5, it is easy to observe the

---

Fig. 6.5. A Specification of a Bounded Buffer (Appending a Character)

```
<event: [[B <= M]]
      where  M = [request: (append: X) reply-to: C]
  (Case-1:
    <pre-cond:
      (B is-a (BOUNDED-BUFFER (q_a: [!x])(q_r: [])(string: [!s])))
      (length([!s]) = N) >
    <next-cond: (B is-a (BOUNDED-BUFFER (q_a: [!x M])(q_r: [])(string: [!s]))) >
    <caused-events: {} >)
  (Case-2:
    <pre-cond:
      (B is-a (BOUNDED-BUFFER (q_a: [])(q_r: [])(string: [!s])))
      (length([!s]) < N) >
    <next-cond: (B is-a (BOUNDED-BUFFER (q_a: [])(q_r: [])(string: [!s X]))) >
    <caused-event: [[C <= (append-done:)]] >)
  (Case-3:
    <pre-cond:
       (B is-a (BOUNDED-BUFFER (q_a: [])(q_r: [MM !y])(string: [])))
       (MM = [request: (remove:) reply-to: CC]) >
    <next-cond: (B is-a (BOUNDED-BUFFER (q_a: [])(q_r: [!y])(string: []))) >
    <caused-events: {[[C <= (append-done:)]], [[CC <= (removed: X)]]} >) >
```

following property of the bounded buffer: It is always the case that the character removed in response to the n-th (*remove:*) request is the one which was appended by the n-th (*append:...*) request. More formally,

Property (First-In-First-Out)

Let E_i^r = ⟦B <== [*request:* (*remove:*), *reply-to:* C_i]⟧
denote the i-th event where B receives a (*remove:*) request, and
E_j^a = ⟦B <== [*request:* (*append:* X_j), *reply-to:* ?]⟧
denote the j-th event where B receives an (*append:...*) request.
For any n > 0, if both E_n^r and E_n^a exist,
then there exist an event E = ⟦C_n <== [*reply:* (*removed:* X_n)]⟧ such that E_n^r -act-> E.

6.4 Behavioral Equations

As noted in the beginning of the previous section, our specification method is roughly summarized as:[1]

> "Given a state of an actor A, the behavior of A in response to a message M is expressed by the new state of A and the finite concurrent events caused by the event ⟦A <== M⟧."

The method suggests to us that a state of A can be viewed as a certain mathematical function F_A whose domain is a set $\mathbf{M}$ of actors (or messages) and whose range is a direct product of a set $\mathbf{S}_A$ of states of A and a finite power set $\mathbf{P}(\mathbf{T} \times \mathbf{M})$ of a direct product of a set $\mathbf{T}$ of target actors and $\mathbf{M}$. [Note that $\mathbf{T} \times \mathbf{M}$ corresponds to a set of events.]

$$F_A : \mathbf{M} \dashrightarrow \mathbf{S}_A \times \mathbf{P}(\mathbf{T} \times \mathbf{M}).$$

---

[1] For the sake of simplicity, we do not take into account the states of the message M and the actors involving in the caused events.

Whether or not the function F_A exists as a well defined mathematical object needs to be proved, but we do believe that the following isomorphism would be shown to hold by a certain domain construction for $\mathbf{S}_A$ similar to that for the lambda calculus done by D. Scott[1972].

$$\mathbf{S}_A \cong (\mathbf{M} \dashrightarrow \mathbf{S}_A \times \mathbf{P}(\mathbf{T} \times \mathbf{M})).$$

where (---->) denotes a set of *continuous functions* with a specified domain and range. The construction of such domains will establish the mathematical meanings of actor states which are described by conceptual representations.

The above isomorphism is inspired by the notion of processes proposed by R. Milner [Milner73]. Extending the work of D. Scott, R. Milner has expressed the meaning of a program by the notion of processes. He defines his notion of processes by the following isomorphism.

$$\mathbf{P} \cong (\mathbf{V} \dashrightarrow \mathbf{P} \times \mathbf{V})$$

which says that a set $\mathbf{P}$ of processes is isomorphic to a set of continuous functions from a domain $\mathbf{V}$ of values to a direct product of $\mathbf{P}$ and $\mathbf{V}$. There are fundamental differences between his approach and ours, due to the framework of the two approaches. Our approach is based on the computation model in which a computation is defined as a partially ordered set of events and for each actor, a total order [called an arrival ordering] is defined. In Milner's approach, a computation is defined as a composition of processes in which parallelism is expressed as a non-deterministic choice of processes by "oracles". The introduction of oracles forces us to consider uninteresting details of the interleaving of concurrent processes. Furthermore, the lack of arrival ordering makes it difficult to deal with the issues of fairness and starvation.

C. Hewitt and H. Baker [Hewitt-Baker77] have shown that the behavior of a <u>pure</u>

actor can be defined as the minimal fixpoint of a continuous functional. This result does not apply to the whole set of actors. Thus we hope that the approach exemplified by the above isomorphism will be able to deal with the whole class of determinate actors.

7. Verifying Parallel Computations

In this chapter, our techniques for verification of actors which are used in parallel computations (in multi-process environments) are presented. In the first section, a special class of actors which are used for synchronization and scheduling of requests is described. To illustrate the verification techniques, an air line reservation system and a bounded buffer which are implemented with such a class of actors are considered in the subsequent sections.

7.1 Introduction

As noted earlier, if, in a parallel computation, concurrent processes do not interact with each other, the parallel computation can be viewed as a collection of mutually independent serial computations and its specification is given as the collection of specifications for the serial computations. The verification of such a parallel computation is nothing but a repetition of the verifications of serial computations. Consequently no special techniques in addition to those for serial computations are required.

In the previous chapter, we have developed specification methods which are applied to computations in which interactions among concurrent processes are involved. Since interactions between processes are performed by sending messages to certain kinds of actors, our specification methods focus upon the behaviors of such actors. We have given various specifications for such actors. But those specifications merely express the behavior that users or implementors of such actors assume or hope they have. There is no guarantee that actually implemented actors behave correctly with respect to their specifications.

In this chapter, we first discuss how such actors are implemented and then explain how they are verified. As examples, we will verify implementations of an air line reservation system and a bounded buffer.

7.2 Serializers

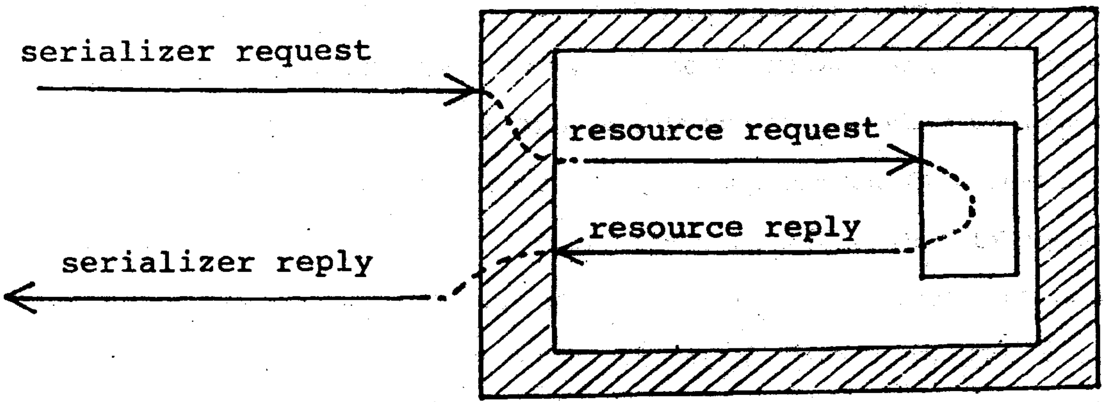
In our model of computation, we use a special class of actors, called *serializers*[Atkinson-Hewitt77], to realize synchronization and scheduling of message transmissions in a uniform and modular fashion. In this section we explain the concept of serializers and give precise specifications for their behavior. The language constructs for serializers, and their relationship to other synchronization primitives such as monitors [Brinch-Hansen73, Hoare74], are discussed in [Atkinson-Hewitt77].

7.2.1 Concept of Serializers

The purpose of a serializer is to enforce orderly uses of resource-like actors [such as I/O devices, message buffers, directories, files, data base systems e.t.c.] by concurrently running processes: Some resources must be used one at a time to guarantee correct functioning of hardware, some should be used on a certain priority basis for special demands and efficiency reasons, and some should receive messages in a proper order for maintaining their integrity.

In order to control access to a resource, we encase the resource in a serializer to intercept the messages sent to it. Any processes which need to use the resource can send a request message to it freely, but all requests are first received by the serializer. The serializer sends the requests to the resource at an appropriate time depending upon the physical requirements of the resource and the scheduling and priority adopted for the resource. No request message arrives at the the resource directly. We call the arrival of such a request message at the serializer, *a serializer request* and the arrival at the resource of a request message which is sent by the serializer, *a resource request*.

In order for a serializer to properly perform such synchronization and scheduling of requests, it must know various information such as what state the resource is in, which requests are being suspended, and which are being granted. To keep such information accurate, the reply (or results) produced upon the completion of the use of the resource is first sent to the serializer, and some of the information kept in the serializer is updated, and then the serializer returns the reply as a response for the original serializer request. We call the former event *a resource reply* and the latter *a serializer reply*.



Thus a typical sequence of events associated with the use of the resource encased by a serializer starts with a serializer request and then the resource request is made when it is appropriate. The resource reply follows upon the completion of the use of the resource, and finally the serializer reply takes place as a response to the original serializer request. The diagram above shows this sequence of events.

7.2.2 Behavior of Serializers

As was mentioned above, a serializer maintains certain kinds of information to make resource requests take place in such a way that desirable resource usage is accomplished. To store and update such information, a serialzer may have three types of information storage: *queues*, *crowds* and *counters*. Below we look into the behavior of a serializer in more detail by explaining the functions of such information storage.

Queues in a serializer are used to store request messages which have arrived at the serializer, but whose corresponding resource requests have not yet taken place. They also record the order of the arrivals of such request messages. A serializer may have more than one queue to sort out request messages by their types. (For example, requests for reading data are stored in a queue different from the one for write requests.) Suppose that a message [*request:* RQ *reply-to:* C] arrives at a serializer G. (This is a serializer request event.) If the request RQ should not be sent to the resource encased by G at that time, the message [*request:* RQ *reply-to:* C] is put at the rear of a queue in G. Later on, when the message is at the front of the queue and certain conditions for synchronization or scheduling are met, the message is removed from the queue and a new message [*request:* RQ *reply-to:* BP] is created and sent to the resource. This is a resource request event. RQ is the request contained in the original message sent to G. BP is a newly created actor, called a *buck passer*, which has the following special properties:

(1) BP remembers (knows about) the serializer G by which it is created.

(2) BP remembers the continuation[1] C contained in the original message sent to G.

(3) BP shares the same arrival ordering with the serializer G.[2]

The third property means that the order between the arrival of a message at G and the arrival of a message at BP is always defined. [More intuitively, BP and G share the same arbiter.] Since BP is sent to the resource as the continuation in the message for the resource request, BP eventually receives a reply from the resource, if the resource replies. This is a resource reply event. Although we explained in the previous subsection that the reply from the resource is sent to the serializer G, the above account is more accurate. However, the

---

[1] See Sections 3.1.2 and 3.1.3. in Chapter 3 for the definition of continuation.

[2] The model of computation defined in Chapter 3 does not assume this kind of "combined" arrival ordering. This assumption is solely for the simplicity of explanation. By letting the buck passer BP send itself to the serializer G together with the message it received, this assumption can be eliminated. See appendix V.

previous explanation is justified by the property of the buck passer BP which shares the same arrival ordering with the serializer G.

Crowds in a serializer are used to store buck passers which are created when requests are sent to the resource by the serializer. The existence of some buck passer BP in a crowd indicates that the corresponding use of the resource has not been completed yet, because BP is taken out from the crowd only when BP receives the reply from the resource (which means the completion of the resource usage). [It is the third property of a buck passer described above that allows the serializer to eliminate the buck passer from the crowd upon the arrival of the reply at the buck passer.] More than one crowd may be used in a serializer to distinguish the types of resource requests being granted. For example, by having two crowds, a serializer encasing some file is able to know whether the file is currently being read or written.

Let us consider the behavior of a serializer in a resource reply event. Suppose that a buck passer BP in a crowd CR receives a reply RP from the resource. If certain synchronization and scheduling conditions are met, the serializer takes out the front element [*request:* RQ *reply-to:* C] from one of the queues, and a new request message of the form [*request:* RQ *reply-to:* NBP] is created and sent to the resource. When the new request message is created, a new buck passer NBP (which remembers C) is created and put in a crowd (which may be different from the crowd CR). At the same time, the old buck passer BP is deleted from CR. The serializer has another responsibility. It must send the reply RP (just received by the buck passer BP) to the continuation remembered by BP. This is the serializer reply event. Recall that BP is created for remembering the continuation originally contained in the message sent to the serializer.

Counters in a serializer are used to record various numbers about events associated with the serializer. For example, a counter records the difference between the numbers of resource reply events of various kinds. A simple example of the uses of a counter will be found in Section 7.4.

7.2.3 One-at-at-Time Serializer (An Example)

The behavior of serializers informally explained in the previous subsections can be rigorously specified in our formalism. To illustrate how their behavior is expressed in our formalism, we give a formal specification of a simple serializer called one-at-a-time in Figure 7.1. A resource encased by this serializer is used, at most, by one process at a time, and on a first-come-first-served basis.

The first event specification in Figure 7.1 says that when an actor `create-one-at-a-time` receives a resource R, it creates a serializer G which has one queue and one crowd, both of which are initially empty.

The behavior of G in response to a request message depends on the state of G. If both the queue and crowd are empty [(*Case-1:*) of the second event specification in Figure 7.1], a buck passer BP is created and put in the crowd and a request message containing BP as the continuation is sent to the resource R. Otherwise (*Case-2:*), the request message is enqueued and no event is caused.

The third event specification says that when a buck passer BP which is inside the crowd of G receives a reply message, if the queue of G is empty (*Case-1:*),[1] BP is deleted

---

[1] Being able to check whether or not the queue of G is empty relies on the assumption that the state of G can be determined at the time when the buck passer BP receives a message. This assumption is implied by one of the general properties of buck passers that a buck passer shares the arrival ordering with the serializer by which it is created. In Appendix V, a specification of one-at-a-time serializers which does not rely on this assumption is given.

Fig. 7.1. A Specification of a One-at-a-Time Scheduler

```
<event: ⟦create-one-at-a-time <= R⟧
  <return: G* >
  <post-cond: (G is-a (ONE-AT-A-TIME (queue: [])(crowd: {})(resource: R))) >>

<event: ⟦G <== M⟧
     where  M = [request: RQ  reply-to: C]
  (Case-1:
    <pre-cond:  (G is-a (ONE-AT-A-TIME (queue: [])(crowd: {})(resource: R))) >
    <next-cond:
      (G is-a (ONE-AT-A-TIME (queue: [])(crowd: {BP*})(resource: R)))
      (BP is-a (BUCK-PASSER (continuation: C)(serializer: G))) >
    <caused-event: ⟦R <== [request: RQ  reply-to: BP]⟧ >)
  (Case-2:
    <pre-cond:  (G is-a (ONE-AT-A-TIME (queue: [!x])(crowd: {BP})(resource: R))) >
    <next-cond:  (G is-a (ONE-AT-A-TIME (queue: [!x M])(crowd: {BP})(resource: R))) >
    <caused-events: {}  >)>

<event: ⟦BP <== [reply: A]⟧
     where   (BP is-a (BUCK-PASSER (continuation: C)(serializer: G)))) >
  (Case-1:
    <pre-cond:  (G is-a (ONE-AT-A-TIME (queue: [])(crowd: {BP})(resource: R))) >
    <next-cond:  (G is-a (ONE-AT-A-TIME (queue: [])(crowd: {})(resource: R))) >
    <caused-event: ⟦C <== [reply: A]⟧ >)
  (Case-2:
    <pre-cond:
      (G is-a (ONE-AT-A-TIME (queue: [WM !x])(crowd: {BP})(resource: R)))
      (WM = [request: RQ reply-to: CC])   >
    <next-cond:
      (G is-a (ONE-AT-A-TIME (queue: [!x])(crowd: {NBP*})(resource: R)))
      (NBP is-a (BUCK-PASSER (continuation: CC)(serializer: G))) >
    <caused-events: { ⟦C <== [reply: A]⟧ , ⟦R <== [request: RQ reply-to: NBP]⟧ }  >)>
```

from the crowd and the reply message is sent back to the continuation C remembered by BP. If the queue is not empty (*Case-2:*), the front element WM which is a suspended request message sent to G before is dequeued and a newly created buck passer NBP replaces BP in the crowd. Then a serializer reply event ⟦C <== [*reply:* A]⟧ and a resource request event ⟦R <== [*request:* RQ *reply-to:* NBP]⟧ take place concurrently.

Before ending this section, we should mention several properties of the one-at-a-time serializer which are easily derived from the specification given in Figure 7.1.

If a resource R is encased by a one-at-a-time serializer before R becomes known to other actors, there is no way to access the resource directly.[1] In order to access the resource, first a request must be sent to the one-at-a-time serializer. This property holds for any kind of serializer (not just for one-at-a-time serializers). We call this property the resource confinement of serializers. More formally,

Property (Resource Confinement of Serializers)
Let E_0 = ⟦create-a-resource <== [*request:* I *reply-to:* create-a-serializer]⟧ and
E_1 = ⟦create-a-serializer <== [*request:* R *reply-to:* C]⟧ such that E_0 -act-> E_1,
where I is used for the creation of a new resource R.
and let G be a serializer created by E_1.
If there exists no event EE = ⟦A <== [*request:* R *reply-to:* ?]⟧
such that E_0 ---> EE ---> E_1,
then for any event ER = ⟦R <== [*request:* RQ *reply-to:* ?]⟧,
there always exists an event E = ⟦G <== [*request:* RQ *reply-to:* ?]⟧
such that E -act-> ER.

We need to give the definition of an assertion (A *is-used-serially*) to state the properties of one-at-a-time serializers. If the assertion (A *is-used-serially*) holds, an actor A

---

[1] We assume that the creator of R does not release any information which makes it possible to have access to R.

does not receive any message until the current invocation of A is completed. Consequently, if the invocation is not completed, no more messages arrive at A. More formally,

Definition (A ***is-used-serially***)
  If there exists an event E_i = ⟦A <== [*request:* RQ_i ***reply-to:*** C_i]⟧,
   then
    if there exists another event E_j = ⟦A <== [*request:* RQ_j ***reply-to:*** C_j]⟧
      such that $i \neq j$ and $E_i \text{ -arr->}_A E_j$,
     then there must exist EE_i = ⟦C_i <== [***reply:*** ?]⟧
        such that E_i ---> EE_i ---> E_j.

Property-I (Serial Use of Resource)
  If an resource actor R encased by a one-at-a-time serializer, then (R ***is-used-serially***) holds.

This property is derived from the fact that the number of buck passer actors in the crowd of the serializer is always one at most.

Definition (A ***is-guaranteed-to-reply***)
  For an event E = ⟦A <== [*request:* RQ ***reply-to:*** C]⟧,
   there always exists an event EE = ⟦C <== [***reply:*** ?]⟧ such that E -act-> EE.

Property-II (Guaranteed Resource Access)
  Suppose that the resource actor R encased by a one-at-a-time serializer G satisfies
    the following condition: if (R ***is-used-serially***), then (R ***is-guaranteed-to-reply***).
  Then, for any event E = ⟦G <== [*request:* RQ ***reply-to:*** ?]⟧,
    there always exists an event ER = ⟦R <== [*request:* RQ ***reply-to:*** ?]⟧ such that E -act-> ER.

This property is derived from Property-I by induction on the number of messages that have already arrived at G.

Property-III (First Come First Resource Access)

Under the same premise given in Property-II,

for any E_i, E_j where E_k = ⟦G <== [*request:* RQ_k *reply-to:* C_k]⟧, k = i, j,

if $E_i \dashrightarrow_G E_j$,

then $ER_i \dashrightarrow E_j$

where ER_k = ⟦R <== [*request:* RQ_k *reply-to:* ?]⟧, k = i, j.

This property is derived from the fact that requests sent to **G** are recorded in the queue of which preserves the order of arrival.

7.3 Verifying Implementations of Actors I

In this section, we discuss our techniques for the following class of verification problems.

> "Given an actor A which shows some behavior in serial computations (i.e., when it is used serially). Suppose that an actor B is implemented as a one-at-a-time serializer encasing the actor A. Then we would like to verify that even if B is sent messages concurrently, B shows the same behavior as A does in serial computations."

This problem is not trivial because the states of A and B which are used to describe their behavior in specifications are expressed by different conceptual representations. The essential part of the verification is the use of the mapping (implementation invariant) between two different conceptual representations. The technique illustrated below is an extension of the one used for the verification of actors behaving as information storage discussed in Section 5.3, Chapter 5. The verification of implementations using more complicated serializers is discussed in the next section (7.4).

In what follows, as an example of such verification problems, we will demonstrate that the implementation of an air line reservation system given below meets its specification depicted in Figure 7.2 (which is the same one given in Figure 6.2 in Chapter 6).

7.3.1 An Implementation of an Air Line Reservation System

We implement an air line reservation system which is supposed to meet the specification in Figure 7.2 in two steps. First, we implement a flight data actor which satisfies the specification in Figure 7.2 as long as it is used serially. Then it is encased by a one-at-a-time serializer. [The flight data actor corresponds to the actor A in the above problem statement.]

The code given in Figure 7.3 is an implementation of such a flight data actor. It

Fig. 7.2. A Specification of an Air Line Reservation System

```
<event: ⟦create-flight <= S⟧
    <pre-cond: (S > 0) >
    <return: F* >
    <post-cond:  (F is-a (FLIGHT (seats-free: S) (passenger-name-list: {})))>>

<event: ⟦F <= (reserve-a-seat: NAME)⟧
  (case-1:
    <pre-cond:  (F is-a (FLIGHT (seats-free: 0) (passenger-name-list: {!pnl})))>
    <next-cond:  (F is-a (FLIGHT (seats-free: 0) (passenger-name-list: {!pnl})))>
    <return:  (no-more-seats:) >)
  (case-2:
    <pre-cond:
      (F is-a (FLIGHT (seats-free: N) (passenger-name-list: {!pnl})))
      (N > 0) >
    <next-cond: (F is-a (FLIGHT (seats-free: N - 1) (passenger-name-list: {!pnl NAME})))>
    <return:  (ok-its-reserved:) >)>

<event:  ⟦F <= (cancel-a-seat: NAME)⟧
  (case-1:
    <pre-cond:
         (F is-a (FLIGHT (seats-free: N) (passenger-name-list: {!pnl})))
         (pnl ≠ {... NAME ...})>
    <next-cond:  (F is-a (FLIGHT (seats-free: N) (passenger-name-list: {!pnl})))>
    <return:  (the-passenger-name-not-found:) >)
  (case-2:
    <pre-cond:
      (F is-a (FLIGHT (seats-free: N) (passenger-name-list: {!pnl1 NAME !pnl2})))>
    <next-cond: (F is-a (FLIGHT (seats-free: N + 1) (passenger-name-list: {!pnl1 !pnl2})))>
    <return:  (ok-its-cancelled:) > ) >
```

Fig. 7.3. A Code For a Flight Data

```
(create-flight-data =s) ≡
 (let (seats-free initially s)                          ;a variable seats-free is initialized to s.
     (passenger-name-list initially (create-empty-set))
    then                                    ;a variable passenger- is initialized to an empty set.
  (cases
    (≡> (reserve-a-seat: =name)                          ;when a (reserve-...) message is received,
      (rules (seats-free = 0)                                        ;if the value of seats-free is 0
        (≡> yes  (no-more-seats:))                         ;then a (no-more-seats:) is returned.
        (≡> no                                                                          ;otherwise
          (seats-free ← (seats-free - 1))             ;the value of seats-free is decreased by one
          (add name to passenger-name-list)                             ;name is added to the list.
          (ok-its-reserved:))))                        ;a message (ok-its-reserved:) is returned.
    (≡> (cancel-a-seat: =name)                            ;when a (cancel-...) message is received,
      (rules (name in passenger-name-list)        ;if name is found in the passenger name list,
        (≡> yes                                                                             ;then
          (delete name from passenger-name-list)                     ;name is deleted from the list
          (seats-free ← (seats-free + 1))             ;the value of seats-free is increased by one
          (ok-its-cancelled:))                                      ;(ok-its-cancelled:) is returned.
        (≡> no  (the-passenger-name-not-found:)) )) )) ;otherwise (the-passenger-...) is returned.
```

---

should be noted that if the flight data actor were sent more than one message concurrently, anomalous results would be caused. For example, if *(reserve-a-seat:...)* and *(cancel-a-seat:...)* message are sent concurrently, *(no-more-seats:)* message might be returned even if there are still vacant seats. Therefore this actor must be used serially.

We give a specification of this actor in Figure 7.4. Though this specification looks similar to that for the air line reservation system in Figure 7.2, there are important diffferences. In this specification conceptual representations of the following form are used.

(FLIGHT-DATA (seats-free: ?)(passenger-name-list: {...}))

Fig. 7.4. A Specification of A Flight Data Actor

```
<event: ⟦create-flight-data <= S⟧
    <pre-cond: (S > 0) >
    <return: FD* >
    <post-cond:  (FD is-a (FLIGHT-DATA (seats-free: S) (passenor-name-list: {})))>>

<event: ⟦FD <= (reserve-a-seat: NAME)⟧
      where  (FD is-used-serially)
  (case-1:
    <pre-cond:  (FD is-a (FLIGHT-DATA (seats-free: 0) (passenger-name-list: {!pnl})))>
    <return:  (no-more-seats:) >)
    <post-cond:  (FD is-a (FLIGHT-DATA (seats-free: 0) (passenger-name-list: {!pnl})))> )
  (case-2:
    <pre-cond:
      (FD is-a (FLIGHT-DATA (seats-free: N) (passenger-name-list: {!pnl})))
      (N > 0) >
    <return:  (ok-its-reserved:) >
    <post-cond:
      (FD is-a (FLIGHT-DATA (seats-free: N - 1) (passenger-name-list: {!pnl NAME})))>)>

<event: ⟦FD <= (cancel-a-seat: NAME)⟧
      where  (FD is-used-serially)
  (case-1:
    <pre-cond:
      (FD is-a (FLIGHT-DATA (seats-free: N) (passenger-name-list: {!pnl})))
      (pnl ≠ {... NAME ...})>
    <return:  (the-passenger-name-not-found:) >
    <post-cond:  (F is-a (FLIGHT-DATA (seats-free: N) (passenger-name-list: {!pnl})))> )
  (case-2:
    <pre-cond:
     (FD is-a (FLIGHT-DATA (seats-free: N) (passenger-name-list: {!pnl1 NAME !pnl2})))>
    <return:  (ok-its-cancelled:) >
    <post-cond:
     (FD is-a (FLIGHT-DATA (seats-free: N + 1) (passenger-name-list: {!pnl1 !pnl2})))>)>
```

Notice that assertions of the form (FD *is-used-serially*) are given in the *where* clauses of the second and third event specifications. This means that those event specifications are valid only if FD is used serially. Furthermore, <*post-cond:* ...> clauses are used instead of <*next-cond:*...> clauses. This means that assertions in the <*post-cond:*...> clauses hold at the time when the caused events take place.

The following property holds for the flight data actor because all the <*event:*...> clauses have the corresponding <*return:*...> clauses. This property is used in the verification in the next subsection.

Property-IV: If (FD *is-used-serially*), then (FD *is-guaranteed-to-reply*).

7.3.2 Verification of the Air Line Reservation System

The implementation is completed by encasing the flight data actor by a one-at-a-time serializer. That is, the implementation of the create-flight actor is expressed by the following PLASMA code:

(create-flight =s) ≡ (create-one-at-a-time (create-flight-data s)).

Below we demonstrate that the above code meets the specification of the air line reservation system shown in Figure 7.2. The symbolic evaluation of the code

(create-one-at-a-time (create-flight-data s))

reveals the following facts:

(1) an actor FD is created by ⟦create-flight-data <= s⟧ [from the specification in Figure 7.4.],

(2) a serializer G is created by ⟦create-one-at-a-time <= FD⟧ [from the specification in

Figure 7.1.] and

(3) the two actors satisfy the following assertions immediately after the creation of G.

(G *is-a* (*ONE-AT-A-TIME* (*queue:* [])(*crowd:* {})(*resource:* FD)))
(FD *is-a* (*FLIGHT-DATA* (*seats-free:* s)(*passenger-name-list:* {})))

We will establish that G satisfies the specification of the flight actor (air line reservation system) given in Figure 7.2. The specification of the flight actor G is written in terms of conceptual representations of the form:

(G *is-a* (*FLIGHT* (*seats-free:* ?)(*passenger-name-list:* {...}))) (*)

(Notice that F in the specification is instantiated as G.) On the other hand, G is <u>implemented</u> as a one-at-a-time serializer that encases the flight data actor FD, which is expressed by the following two assertions:

(G *is-a* (*ONE-AT-A-TIME* (*queue:* [...])(*crowd:* {...})(*resource:* FD)))
(FD *is-a* (*FLIGHT-DATA* (*seats-free:* ?)(*passenger-name-list:* {...}))) (**)

This means that we have <u>two</u> <u>views</u> of G: an external view expressed by (*) and an internal implementation expressed by (**) above. In order to show that the implementation satisfies the specification written in terms of the external view, we must establish a certain relation between the two views. Such a relation is similar to implementation invariants used in the verification of an actor behaving as information storage [Cf. Section 5.3, Chapter 5].

The relation we need is:

"*If* G satisfies the assertion
  (G *is-a* (*FLIGHT* (*seats-free:* N) (*passenger-name-list:* {!pnl})))
    in a situation where G receives a message [*request:* RQ *reply-to:* ?],
  then FD always satisfies the assertion
      (FD *is-a* (*FLIGHT-DATA* (*seats-free:* N) (*passenger-name-list:* {!pnl})))
    in the situation where FD receives a message [*request:* RQ *reply-to:* ?]. "

We actually prove the validity of this relation in the next subsection 7.3.3; this relation is assumed in the subsequent discussion. The following is the formal statement of the above relation.

```
<Implementation-invariant:
    if   (G is-a (FLIGHT (seats-free: N) (passenger-name-list: {!pnl}))) in S
            where  S = Sit([[ G <== [request: RQ reply-to: ?]]])
       then
            (FD is-a (FLIGHT-DATA (seats-free: N) (passenger-name-list: {!pnl}))) in S'
              where  S' = Sit([[ FD <== [request: RQ reply-to: ?]]])  >.
```

Sit(E) expresses the situation where an event E takes place. The *implemenation invariant* can be viewed as the counterpart of an "invariant" in parallel process environments, which was first introduced by C.A.R. Hoare [Hoare 1972] to show correctness of implementations of data structures used in serial computations. (See the remarks in Section 5.3.1, Chapter 5.)

Now let us demonstrate the verification of the implementation against the following event specification given in Figure 7.2.

```
<event: [[ F <= (reserve-a-seat: NAME)]]
  (case-1:
    <pre-cond:  (F is-a (FLIGHT (seats-free: 0) (passenger-name-list: {!pnl})))>
    <next-cond:  (F is-a (FLIGHT (seats-free: 0) (passenger-name-list: {!pnl})))>
    <return:  (no-more-seats:)  >)
  (case-2:
    <pre-cond:
       (F is-a (FLIGHT (seats-free: N) (passenger-name-list: {!pnl})))
       (N > 0)  >
    <next-cond: (F is-a (FLIGHT (seats-free: N - 1) (passenger-name-list: {!pnl NAME})))>
    <return:  (ok-its-reserved:)  >)>
```

There are two cases to be considered. We only consider the (*Case-2...*) clause. The one-at-a-time serializer G receives a (*reserve-a-seat:* NAME) request RQ. Since the flight data actor FD is guaranteed to reply if it is used serially (from Property-IV), the specification for a one-at-a-time guarantees that the (*reserve-a-seat:* NAME) request RQ is received by FD (from Property-II). To know the state of the flight data actor FD at the time of the arrival of RQ, the above implementation invariant is used. Since the state of G at the time of the arrival of RQ at G is described as:

(G *is-a* (*FLIGHT* (*seats-free:* N) (*passenger-name-list:* {!pnl}))),

the state of FD at the time of the arrival of M at FD is described as

(FD *is-a* (*FLIGHT-DATA* (*seats-free:* N) (*passenger-name-list:* {!pnl}))).

Then the (*Case-2*...) clause in the <*event:*...> clause of the specification for flight-data actors in Figure 7.4 is referred to. Since the precondition that FD must be used serially is satisfied (from Property-I), the (*Case-2*...) clause of the specification for flight data actors in Figure 7.4 tells us that

(1) (*ok-its-reserved:*) is returned, and

(2) the state of FD is now expressed as:

(FD *is-a* (*FLIGHT-DATA* (*seat-free:* N - 1) (*passenger-name-list:* {!pnl NAME}))).

(1) is what the <*return:*...> clause in the above event specification requires.[1] To complete the demonstration, we must show that the assertion

(G *is-a* (*FLIGHT* (*seat-free:* N - 1) (*passenger-name-list:* {!pnl NAME})))

in the <*next-cond:*...> clause of the above event specification holds when G receives the next

---

[1] More precisely, (*ok-its-reserved:*) is first sent to the serializer G and then G returns it.

message RQ'. To do so, again the implementation is used. It translates the above requirement as follows:

" (FD *is-a* (*FLIGHT-DATA* (*seat-free:* N - 1) (*passenger-name-list:* {!pnl NAME})))
holds when FD receives RQ'. "

This is guaranteed by (2) because FD does not change its state until the next message RQ' arrives at FD. Thus Case-2 is shown. Case-1 may be shown analogously. The event specification for ⟦G <= (*cancel-a-seat:* NAME))⟧ is also established analogously.

The demonstration above assumes that no one can have access to the flight data actor FD except through the serializer G. This assumption always holds because the flight data actor FD created by ⟦create-flight-data <= s⟧ is sent directly to the create-one-at-a-time actor and never released outside the newly created one-at-a-time serializer G. [Cf. the PLASMA code in the beginning of this subsection and Property (Resource Confinement of Serializers).]

7.3.3 Establishing the Implementation Invariant

The verification in the previous subsection relies critically on the use of the following implementation invariant. In this subsection we will establish the validity of this implementation invariant.

<*Implementation-invariant:*
if (G *is-a* (*FLIGHT* (*seats-free:* N) (*passenger-name-list:* {!pnl}))) *in* **S**
where **S** = *Sit*(⟦G <== [*request:* RQ *reply-to:* ?]⟧)
then
(FD *is-a* (*FLIGHT-DATA* (*seats-free:* N) (*passenger-name-list:* {!pnl}))) *in* **S'**
where **S'** = *Sit*(⟦FD <== [*request:* RQ *reply-to:* ?]⟧) >.

(Proof) The proof is done by induction on the number M of messages which have already arrived at G.

<Induction Base>

M = 0: Since no message has arrived before, when the first message [*request:* RQ *reply-to:* C] arrives at G, G is in the same state as it was in at the time of its creation. So the state of G is expressed as

(G *is-a* (*FLIGHT* (*seats-free:* S)(*passenger-name-list:* {}))).

Since G is created as a one-at-a-time serializer and its queue and crowd are initially empty, the state of G is also expressed as

(G *is-a* (*SERIALIZER* (*queue:* [])(*crowd:* {})(*resource:* FD))) and

(FD *is-a* (*FLIGHT-DATA* (*seats-free:* S)(*passenger-name-list:* {})))

Then from the "guaranteed resource access" property of G (Property-II), the following event is caused.

⟦FD <== [*request:* RQ *reply-to:* ?]⟧

When this event occurs, FD is still in the same state as it was in at the time of its creation because "resource confinement" property of serializers is satisfied. So the state of the FD is expressed as

(FD *is-a* (*FLIGHT-DATA* (*seats-free:* S)(*passenger-name-list:* {})))

Hence the induction base is proved.

<Induction Hypothesis>

M = k: We assume that the following relation holds.

if (G *is-a* (*FLIGHT* (*seats-free:* N) (*passenger-name-list:* {!pnl}))) holds
in Sit(⟦G <== [*request:* RQ$_k$ *reply-to:* ?]⟧)
then (FD *is-a* (*FLIGHT-DATA* (*seats-free:* N) (*passenger-name-list:* {!pnl}))) holds
in Sit(⟦FD <== [*request:* RQ$_k$ *reply-to:* ?]⟧

<Induction Step>

M = k + 1: Let us assume that the antecedent of the Induction Hypothesis holds. Then we must do a case analysis according to the type of the request of k-th event.

Case-1: RQ$_k$ = (*reserve-a-seat:* NAME), and N > 0.

The state of G immediately after the k-th event ⟦G <== [*request:* RQ$_k$ *reply-to:* ?]⟧ is expressed as

(G *is-a* (*FLIGHT* (*seats-free:* N - 1) (*passenger-name-list:* {!pnl NAME})))

(by the specification of the flight actor in Figure 7.2).

This is the state of G when the k + 1 st message [*request:* RQ$_{k+1}$ *reply-to:* ?]⟧ arrives at G.

By the "guaranteed resource access" property of G, the event

E = ⟦FD <== [*request:* RQ$_k$ *reply-to:* ?]⟧

always takes place. From the induction hypothesis, the state of FD at the time of this event E is expressed as

(FD *is-a* (*FLIGHT-DATA* (*seats-free:* N) (*passenger-name-list:* {!pnl})))

Therefore, by the specification for FD in Figure 7.4, the state of FD after the invocation initiated by the event E is expressed as

(FD *is-a* (*FLIGHT-DATA* (*seats-free:* N - 1) (*passenger-name-list:* {!pnl NAME})))

We now claim that this is indeed the state of FD at the time the k + 1 st message [*request:* RQ_{k+1} *reply-to:* ?] arrives at FD. This claim is justified by the fact that no message arrives at FD between [*request:* RQ_k *reply-to:* ?] and [*request:* RQ_{k+1} *reply-to:* ?]. This fact is guaranteed by two properties of a one-at-a-time serializer, the "Confinement of resource" and the "First Come First Resource Access" (Property-III).

Other cases are shown in a similar fashion. (End of Proof)

The above proof relies on the following facts:

(1) When the one-at-a-time serializer G encasing the flight data actor FD is created, each component [such as *seats-free* and *passenger-name-list*] of the conceptual representation expressing the external state of G is the same as the corresponding component of the conceptual representation expressing the state of FD.

(2) As the specifications for G and FD show, such components of conceptual representations for G and FD change in the same way in response to the same request, provided that FD is used serially.

(3) The serial use of the resource encased by a one-at-a-time serializer.

(4) The "Resource Confinement" property of serializers.

(5) The "First Come First Resource Access" property of a one-at-a-time serializer.

7.4 Verifying Implementations of Actors II

In the previous section, we discussed the verification of implementations which use one-at-a-time serializers. The resource actor encased by a one-at-a-time serializer receives requests in the same order as the one-at-a-time serializer does. That is, the one-at-a-time serializer have the first come first resource access property [Property-III in Section 7.2]. In this section, we will discuss the verification of implementations using serializers which do not have the first come first resource access property. The heart of verification in this case is the use of implementation invariants, as it was in the case for implementations using one-at-a-time serializers. To find an appropriate implementation invariant for a given implementation requires human ingenuity. In what follows, we will explain the verification of an implementation of a bounded buffer against the specification depicted in Figure 7.5. [This specification is identical to the one given in Figures 6.4 and 6.5.]

7.4.1 An Implementation of A Bounded Buffer

We consider the following PLASMA implementation of a bounded buffer.

```
(create-bounded-buffer []) ≡ (create-buffer-scheduler (create-string-storage []))
```

Namely, the bounded buffer of length N is implemented as a serializer B which encases a string storage actor S where S is created by ⟦`create-string-storage` <= []⟧ and B is created by ⟦`create-buffer-scheduler` <= S⟧. Note that S is encased by B without becoming known to other actors. Thus the resource confinement property of serializers is satisfied.

The behavior of the string storage actor S is described by the specification in Figure 7.6. Its states are expressed by conceptual representations of the following form.

(*STRING-STORAGE* [...])

When it is created, it contains no character. It accepts (***append:*** **<character>**) and (***remove:***) messages. As stated by assertions of the form (S ***is-used-serially***) in the ***where*** clauses, the behavior described in the specification is guaranteed only when S is used serially.

The creation of the serializer B is described by the following event specification.

Fig. 7.5. A Specification of A Bounded Buffer

```
<event: ⟦create-bounded-buffer <= []⟧
   <return: B*>
   <post-cond:  (B is-a (BOUNDED-BUFFER (q_a: [])(q_r: [])(string: []))) >>

<event: ⟦B <= M⟧
          where  M = [request: (remove:) reply-to: C]
  (Case-1: <pre-cond: (B is-a (BOUNDED-BUFFER (q_a: [])(q_r: [!y])(string: []))) >
           <next-cond: (B is-a (BOUNDED-BUFFER (q_a: [])(q_r: [!y M])(string: []))) >
           <caused-events: {} >)
  (Case-2: <pre-cond: (B is-a (BOUNDED-BUFFER (q_a: [])(q_r: [])(string: [X !s]))) >
           <next-cond: (B is-a (BOUNDED-BUFFER (q_a: [])(q_r: [])(string: [!s]))) >
           <caused-event: ⟦C <= (removed: X)⟧ >)
  (Case-3: <pre-cond: (B is-a (BOUNDED-BUFFER (q_a: [MM !x])(q_r: [])(string: [X !s])))
                      (length([X !s]) = N)
                      (MM = [request: (append: XX) reply-to: CC]) >
           <next-cond: (B is-a (BOUNDED-BUFFER (q_a: [!x])(q_r: [])(string: [!s XX]))) >
           <caused-events: {⟦C <= (removed: X)⟧, ⟦CC <= (append-done:)⟧} >) >

<event: ⟦B <= M⟧
          where  M = [request: (append: X) reply-to: C]
  (Case-1: <pre-cond: (B is-a (BOUNDED-BUFFER (q_a: [!x])(q_r: [])(string: [!s])))
                      (length([!s]) = N) >
           <next-cond: (B is-a (BOUNDED-BUFFER (q_a: [!x M])(q_r: [])(string: [!s]))) >
           <caused-events: {} >)
  (Case-2: <pre-cond: (B is-a (BOUNDED-BUFFER (q_a: [])(q_r: [])(string: [!s])))
                      (length([!s]) < N) >
           <next-cond: (B is-a (BOUNDED-BUFFER (q_a: [])(q_r: [])(string: [!s X]))) >
           <caused-event: ⟦C <= (append-done:)⟧ >)
  (Case-3: <pre-cond: (B is-a (BOUNDED-BUFFER (q_a: [])(q_r: [MM !y])(string: [])))
                      (MM = [request: (remove:) reply-to: CC]) >
           <next-cond: (B is-a (BOUNDED-BUFFER (q_a: [])(q_r: [!y])(string: []))) >
           <caused-events: {⟦C <= (append-done:)⟧, ⟦CC <= (removed: X)⟧} >) >
```

Fig. 7.6. A Specification of a String Storage of Length N

```
<event: [[create-string-storage <= []]
  <return: S* >
  <post-cond: (S is-a (STRING-STORAGE [])) >>

<event: [[S <= (append: X)]
    where (S is-used-serially)
  (Case-1: <pre-cond: (S is-a (STRING-STORAGE [!x]))
                      (length(x) < N) >
           <return: (append-done:) >
           <post-cond: (S is-a (STRING-STORAGE [!x X])) >)
  (Case-2: <pre-cond: (S is-a (STRING-STORAGE [!x]))
                      (length(x) ≥ N) >
           <return: (storage-full:) >
           <post-cond: (S is-a (STRING-STORAGE [!x])) >) >

<event: [[S <= (remove:)]
    where (S is-used-serially)
  (Case-1: <pre-cond: (S is-a (STRING-STORAGE [X !x])) >
           <return: (removed: X) >
           <post-cond: (S is-a (STRING-STORAGE [!x X])) >)
  (Case-2: <pre-cond: (S is-a (STRING-STORAGE [])) >
           <return: (storage-empty:) >
           <post-cond: (S is-a (STRING-STORAGE [])) >) >
```

```
<event: [[create-buffer-scheduler <= S]
   <pre-cond: (S is-a (STRING-STORAGE [!x])) >
   <return: B* >
   <post-cond:
      (B is-a (SCHEDULER (counter: 0)($\hat{q}_a$: [])($\hat{q}_r$: [])(crowd: {})(resource: S)))
      (S is-a (STRING-STORAGE [!x])) >>
```

As expressed by the conceptual representation in the *<post-cond:...>* clause, this serializer has a counter (initially 0), two queues, $\hat{q}_a$ and $\hat{q}_r$ (both are initially empty) and a crowd (also initially empty). The counter is used to record the number of characters stored in the string storage. The crowd is used to contain buck passers. The existence of a buck passer in the crowd indicates that the resource is being used. $\hat{q}_a$ and $\hat{q}_r$ are used to record suspended (*append:...*) and (*remove:*) requests, respectively.

The behavior of the serializer B in response to (***append:...***) and (***remove:***) requests are described the event specifications depicted in Figure 7.7 and Figure 7.8, respectively. Let us look at the behavior of B when it receives a message M of the form

[*request:* (*append:* X) *reply-to:* C].

Case-1: if no (***append:***) requests are suspended [i.e. $\hat{q}_a$ is empty], the string storage S is not being used [i.e. the crowd is empty], and there is room for the new character X [k < N], then the (*append:* X) request with a newly created buck passer BP which remembers the original continuation C is sent to S. The state change of B reflects this: the counter is increased by one and the crowd now contains the buck passer BP.

Case-2: if the conditions for Case-1 do not hold, the message M is enqueued at the rear of $\hat{q}_a$.

Figure 7.7 also includes the specification of the event in which the reply (*append-done:*) from S in response to an (***append:***) request is received by the buck passer BP which is currently stored in the crowd of B. When BP receives (***append-done:***), the request suspended in the front element of either $\hat{q}_r$ or $\hat{q}_a$ is picked up and sent to the string storage. If both queues are not empty, $\hat{q}_r$ has priority over $\hat{q}_a$. There are three cases for this event. Note that the counter k indicating the current length of the string storage cannot be 0 when BP receives an (***append-done:***) reply, because a new character has been just appended before the reply is produced.

Case-1: if no (***remove:***) requests are suspended [i.e. $\hat{q}_r$ is empty], and either the string storage is full [i.e. k = N] or no (***append:...***) requests are suspended [i.e., $\hat{q}_a$ is not empty], then the reply is returned to the original continuation remembered by the buck passer P, but no message is sent to S.

Case-2: if there are some suspended (***remove:***) requests [i.e. $\hat{q}_r$ is not empty], then the the front element M of $\hat{q}_r$ is taken out, and the corresponding (***remove:***) request is sent to S with

Fig. 7.7. The Behavior of the Scheduler in response to an (*Append:..*) Request

```
<event: ⟦B <== M⟧   where M = [request: (append: X) reply-to: C]
  (Case-1:
    <pre-cond: (B is-a (SCHEDULER (counter: k)($\hat{q}_a$: [])($\hat{q}_r$: [!y])(crowd: {})(resource: S)))
               (k < N) >
    <next-cond: (B is-a (SCHEDULER (counter: k + 1)($\hat{q}_a$: [])($\hat{q}_r$: [!y])(crowd: {BP*})(resource: S)))
                (BP is-a (BUCK-PASSER (continuation: C)(serializer: B)))>
    <caused-event: ⟦S <== [request: (append: X) reply-to: BP]⟧ >)
  (Case-2:
    <pre-cond: (B is-a (SCHEDULER (counter: k)($\hat{q}_a$: [!x])($\hat{q}_r$: [!y])(crowd: {!z})(resource: S)))
               (v (x ≠ []) (z ≠ {}) (k = N)) >
    <next-cond: (B is-a (SCHEDULER (counter: k)($\hat{q}_a$: [!x M])($\hat{q}_r$: [!y])(crowd: {!z})(resource: S)))>
    <caused-events: {}  >)>

<event: ⟦BP <== [reply: (append-done:)]⟧
      where (BP is-a (BUCK-PASSER (continuation: C)(serializer: B)))
  (Case-1:
    <pre-cond: (B is-a (SCHEDULER (counter: k)($\hat{q}_a$: [!x])($\hat{q}_r$: [])(crowd: {BP})(resource: S)))
               (v (k = N)  ( 0 < k < N  ∧  x = [] )  )>
    <next-cond: (B is-a (SCHEDULER (counter: k)($\hat{q}_a$: [!x])($\hat{q}_r$: [])(crowd: {})(resource: S)))  >
    <caused-event: ⟦C <== [reply: (append-done:)]⟧ >)
  (Case-2:
    <pre-cond: (B is-a (SCHEDULER (counter: k)($\hat{q}_a$: [!x])($\hat{q}_r$: [M !y])(crowd: {BP})(resource: S)))
               (k > 0)
               (M = [request: (remove:) reply-to: CC])>
    <next-cond: (B is-a (SCHEDULER (counter: k - 1)($\hat{q}_a$: [!x])($\hat{q}_r$: [!y])(crowd: {NBP*})(resource: S)))
                (NBP is-a (BUCK-PASSER (continuation: CC)(serializer: B)))>
    <caused-events: {⟦S <== [request: (remove:) reply-to: NBP]⟧ ⟦C <== [reply: (append-done:)]⟧ }>)
  (Case-3:
    <pre-cond: (B is-a (SCHEDULER (counter: k)($\hat{q}_a$: [M !x])($\hat{q}_r$: [])(crowd: {BP})(resource: S)))
               (0 < k < N)
               (M = [request: (append: XX) reply-to: CC]) >
    <next-cond: (B is-a (SCHEDULER (counter: k + 1)($\hat{q}_a$: [!x])($\hat{q}_r$: [])(crowd: {NBP*})(resource: S)))
                (NBP is-a (BUCK-PASSER (continuation: CC)(serializer: B)))>
    <caused-events: {⟦S <== [request: (append: XX) reply-to: NBP]⟧ ⟦C <== [reply: (append-done:)]⟧ }>)>
```

Fig. 7.8. The Behaviors of the Scheduler in response to a (*Remove:..*) Request

```
<event: ⟦B <== M⟧   where M = [request: (remove:) reply-to: C]
  (Case-1:
    <pre-cond: (B is-a (SCHEDULER (counter: k)($\hat{q}_a$: [!x])($\hat{q}_r$: [])(crowd: {})(resource: S)))
               (k > 0) >
    <next-cond: (B is-a (SCHEDULER (counter: k - 1)($\hat{q}_a$: [!x])($\hat{q}_r$: [])(crowd: {BP*})(resource: S)))
                (BP is-a (BUCK-PASSER (continuation: C)(serializer: B)))>
    <caused-event: ⟦S <== [request: (remove:) reply-to: BP]⟧ >)
  (Case-2:
    <pre-cond: (B is-a (SCHEDULER (counter: k)($\hat{q}_a$: [!x])($\hat{q}_r$: [!y])(crowd: {!z})(resource: S)))
               (∨ (y ≠ []) (z ≠ {}) (k = 0))>
    <next-cond: (B is-a (SCHEDULER (counter: k)($\hat{q}_a$: [!x])($\hat{q}_r$: [!y M])(crowd: {!z})(resource: S)))>
    <caused-events: {}>)>

<event: ⟦BP <== [reply: (removed: X)]⟧
      where (BP is-a (BUCK-PASSER (continuation: C)(serializer: B)))
  (Case-1:
    <pre-cond: (B is-a (SCHEDULER (counter: k)($\hat{q}_a$: [])($\hat{q}_r$: [!y])(crowd: {BP})(resource: S)))
               (∨ (k = 0)  ( 0 < k < N  ∧  y = [] )  )>
    <next-cond: (B is-a (SCHEDULER (counter: k)($\hat{q}_a$: [])($\hat{q}_r$: [!y])(crowd: {})(resource: S)))  >
    <caused-event: ⟦C <== [reply: (removed: X)]⟧ >)
  (Case-2:
    <pre-cond: (B is-a (SCHEDULER (counter: k)($\hat{q}_a$: [M !x])($\hat{q}_r$: [!y])(crowd: {BP})(resource: S)))
               (k < N)
               (M = [request: (append: XX) reply-to: CC])>
    <next-cond: (B is-a (SCHEDULER (counter: k + 1)($\hat{q}_a$: [!x])($\hat{q}_r$: [!y])(crowd: {NBP*})(resource: S)))
                (NBP is-a (BUCK-PASSER (continuation: CC)(serializer: B)))>
    <caused-events: {⟦S <== [request: (append: XX) reply-to: NBP]⟧ ⟦C <== [reply: (removed: X)]⟧}>)
  (Case-3:
    <pre-cond: (B is-a (SCHEDULER (counter: k)($\hat{q}_a$: [])($\hat{q}_r$: [M !y])(crowd: {BP})(resource: S)))
               (0 < k < N)
               (M = [request: (remove:) reply-to: CC]) >
    <next-cond: (B is-a (SCHEDULER (counter: k - 1)($\hat{q}_a$: [])($\hat{q}_r$: [!y])(crowd: {NBP*})(resource: S)))
                (NBP is-a (BUCK-PASSER (continuation: CC)(serializer: B)))>
    <caused-events: {⟦S <== [request: (remove:) reply-to: NBP]⟧ ⟦C <== [reply: (removed: X)]⟧}>)>
```

a new buck passer NBP and concurrently the reply is sent to the original continuation C.

Case-3: if no *(remove:)* requests are suspended [i.e. $\hat{q}_r$ is empty], there are some suspended *(append:...)* request [i.e. $\hat{q}_a$ is not empty], and there is room for a new character in S [i.e., $0 < k < N$], then the *(append:...)* request at the front of $\hat{q}_a$ is granted and sent to S with a new buck passer NBP, and concurrently the reply is returned to the original continuation C.

It should be noted that all the three cases are mutually exclusive and enumerate all cases of the states which B can be in when BP receives a *[reply: (append-done:)]* message. The behavior of B in response to *(remove:)* is described in Figure 7.8 in a similar way; the roles of $\hat{q}_a$ and $\hat{q}_r$ are symmetrical and conditions expressing the upper bound for the *counter* is replaced by the lower bound. $\hat{q}_a$ has priority over $\hat{q}_r$ when a buck passer BP receives a *(removed: ?)* from the string storage.

7.4.2 Verification of a Bounded Buffer

In order to show that the implementation of the bounded buffer given in Figures 7.7 and 7.8 satisfies the specification given in Figure 7.5, we need the implementation invariant which is the mapping between the states of a bounded buffer used to write its specification and the states used for describing the implementation. More precisely, we need the mapping from the set of states, called the "specification space", expressed by conceptual representations of the form

(BOUNDED-BUFFER $(q_a$*:* [...])$(q_r$*:* [...])*(string:* [...]))

to the set of states, called the "implementation space", expressed by conceptual representations of the form

(SCHEDULER (counter: ?)$(\hat{q}_a$*:* [...])$(\hat{q}_r$*:* [...])*(crowd:* {...})*(resource:* S)).

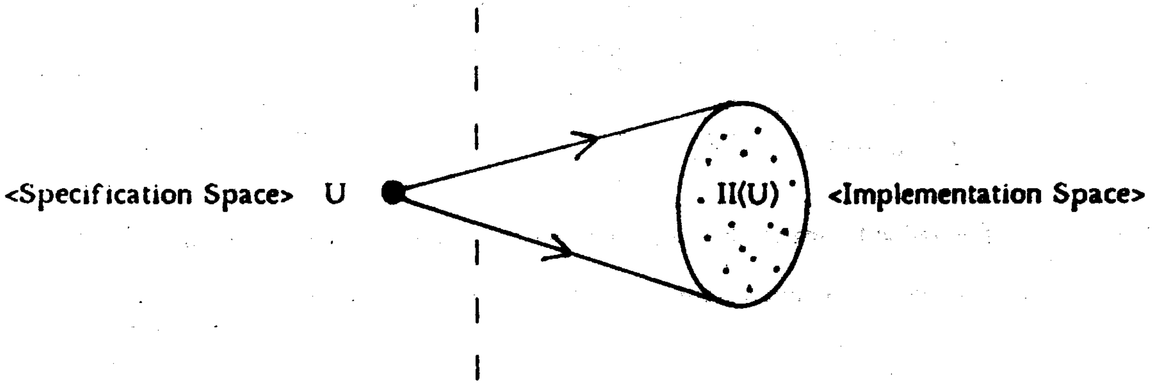
For this purpose, we use the following implementation invariant:

" *If* a bounded buffer B is in the state (of the specification space)
which is expressed by the conceptual representation
(*BOUNDED-BUFFER* (q_a: [!x])(q_r: [!y])(*string*: [!s]))
then
B is in one of the states (of the implementation space)
which are expressed by the conceptual representation
(*SCHEDULER* (*counter:* k)($\hat{q}_a$: [!xx !x])($\hat{q}_r$: [!yy !y])(*crowd:* {!z})(*resource:* S)),
and the following constraints must be satisfied
(1) [!stored-in(S) !characters-appended(xx)] = [!characters-removed(yy) !s]
(2) length(stored-in(S)) = k "

characters-appended(xx) means the sequence of characters that will be appended by the sequence of (*append:...*) requests denoted by xx. characters-removed(yy) means the sequence of characters that will be removed by the sequence of (*remove:*) requests denoted by yy. stored-in(S) means the sequence of characters stored in the string storage S.

Note that q_a and $\hat{q}_a$ share x and q_r and $\hat{q}_r$ share y at their tails. $\hat{q}_a$ and $\hat{q}_r$ denote the queues of requests which are actually waiting inside the scheduler. Thus xx and yy in $\hat{q}_a$ and $\hat{q}_r$ denote the sequences of actually suspended requests that are considered (at the external specification level) to have already been processed. [x and y have not been processed yet.] The first constraint in the above implementation invariant says: the concatenation of the character string that is actually stored in S and the sequence of characters that will be appended by xx is equal to the concatenation of the sequence of characters that will be removed by yy and the character string that is considered (at the external specification level) to be stored in *string*. The second constraint says that the counter k indicates the length of the character string stored in S.

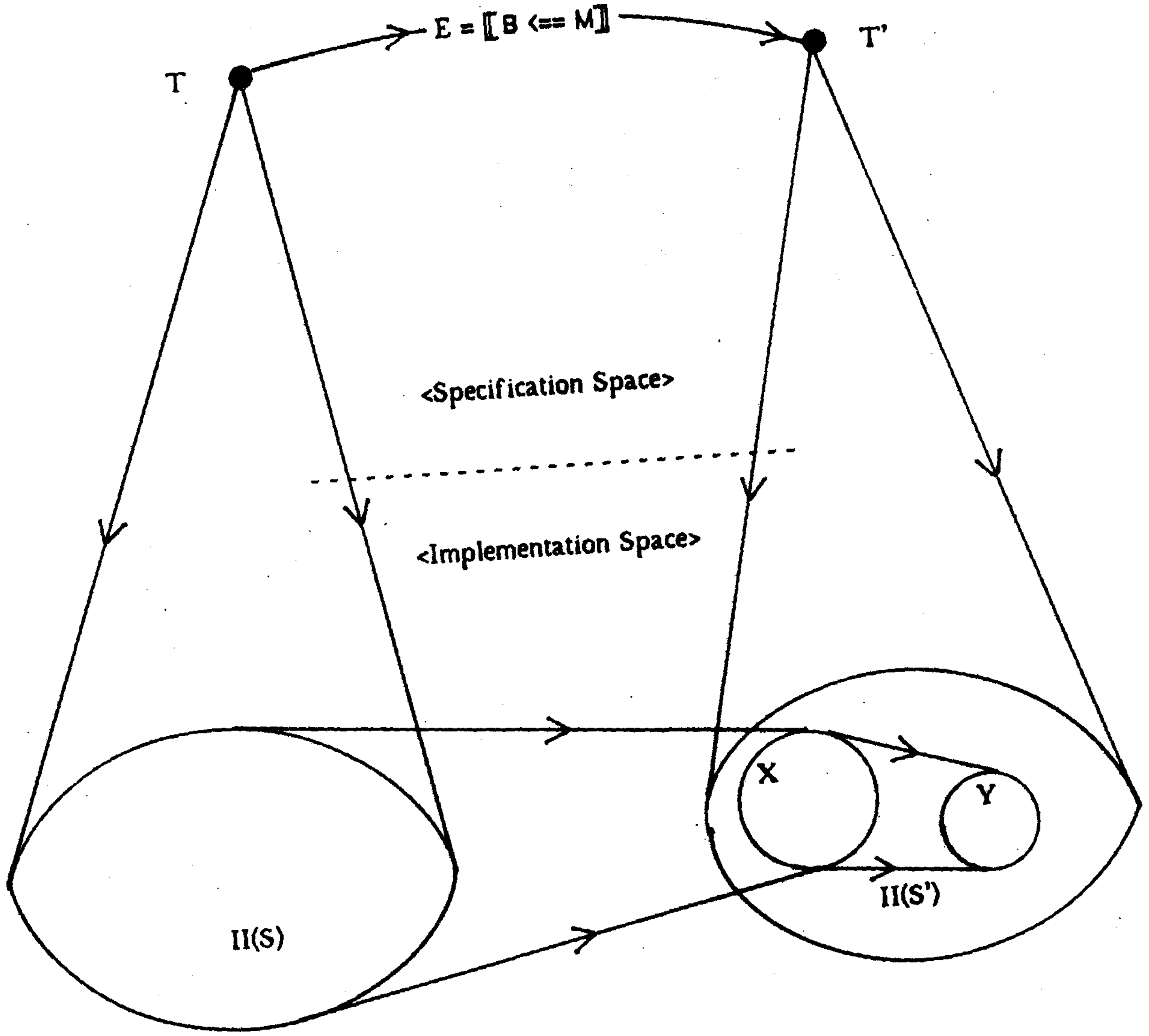
Since, for given x, y and s, only the relation (or constraints) that must be satisfied by xx, yy and k is specified, the above implementation invariant defines a one-to-many correspondence from the specification space to the implementation space. (Cf. Section 5.3.1, Chapter 5) Namely, for a given state U in the specification space, the implementation invariant II give a set II(U) of the corresponding states in the implementation space. See the diagram below.



To verify the implementation against the specification in Figure 7.5, for each event specification in the specification, the implementation must be verified. The diagram in Figure 7.9 illustrates the verification for an event E = **[**B <== M**]**. T and T' are the states of the bounded buffer B given in the ***<pre-cond:...>*** and ***<next-cond:...>*** clauses (of the event specification for E), respectively. II(T) and II(T') are the sets of states (in the implementation space) obtained by applying the implementation invariant II to T and T', respectively.

To establish the event specification, we must first show that if the bounded buffer B is in a state belonging to II(T) before the event E, B is in a state belonging to II(T') immediately after E. To show this, we do not have to deal with individual states in II(T) and II(T'). We use the *relations* among the constituents of the implementation which *define* II(T) and II(T'). [Of course, such relations are obtained from the constraints given in the

Fig. 7.9. Establishing an Event Specification



implementation invariant.] By using the description of the implementation given in Figures 7.7 and 7.8, we obtain (from the defining relation for H(T)) the relation which defines the set X of states in which B can be immediately after E. We check to see whether or not the obtained relation satisfies the defining relation for H(T'), i.e., we check whether or not X is a subset of H(T'). If the obtained relation satisfies the defining relation for H(T'), it is verified that the state of B immediately after the event E is T' in the specification space.

But this does not mean that the implementation satisfies the *<next-cond:...>* clause. We must show that the state of B in the specification space does not change until the next request message (either *(append:...)* or *(remove:)*) arrives at B, because at the implementation level (i.e., when B is considered as a scheduling serializer), a buck passer in the crowd of B may receive a reply message from the string storage S and consequently, the state of B which is currently one of states belonging to X may not belong to H(T') after such a reply event. Therefore we must also show that the state of B stays inside H(T'), which means that such reply events do not change the state of B in the specification space. To do so, we check if the relation defining the set Y of states in which B can be immediately after the resource reply event satisfies the defining relation for H(T').

To complete the verification of the event specification, we must show that the events given in the *<caused-events:...>* clause eventually take place. To do so, we use the fact that the sequence of requests xx in $\hat{q}_a$ and the sequence of requests yy in $\hat{q}_r$ are eventually removed and sent to S. This is easily done by checking the implementation given in Figures 7.7 and 7.8 and the specification of the string storage given in Figure 7.6.

8. Modelling a Post Office

In this chapter, we discuss an actor model of a simple post office which is an intuitive example of systems, such as operating systems and multi-user data base systems, which are characterized by complex concurrent internal activities. In the first section, an informal description of the post office is followed by formal specifications of the individual behavior and mutual interaction of the components of the model. In the second section, the specification of the overall functions (task specifications) of the post office is stated formally. In the last section, we demonstrate that the task specifications are satisfied by the individual behavior and mutual interaction.

8.1 A Model of a Simple Post Office

In this section, we present the actor model of a simple post office. The behavior of each component in the model is described by our specification techniques and the overall properties and effects of the post office as a whole are stated formally. Furthermore, using this model as an example, we would like to shed light on some of the interesting issues related to distributed information processing systems.

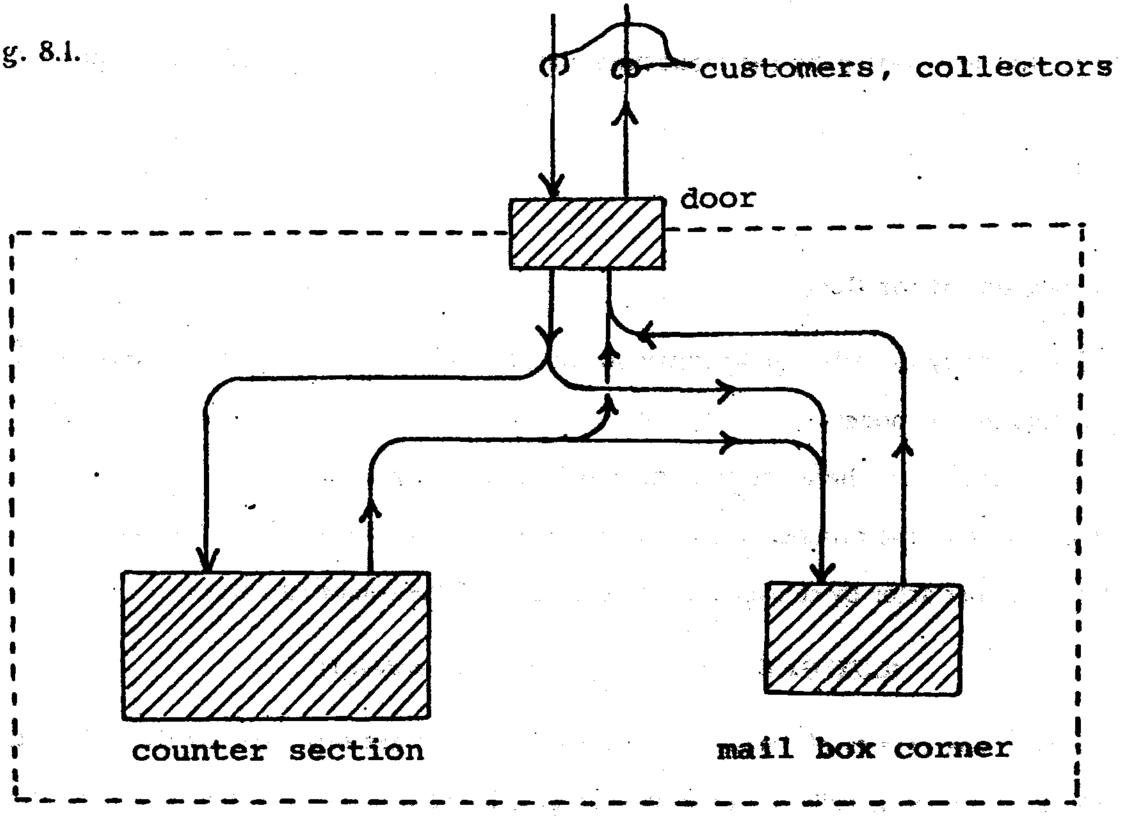
8.1.1 Overview of the Model

An informal description of activities in a simple post office is:

A number of customers and mail collectors visit the post office, possibly simultaneously. The post office has only one door for customers and collectors. Inside the post office, there is a counter section which has several counters and a mail box corner which has a mail box. After a customer enters the post office through the door, if he needs stamps, he goes to the counter section, otherwise he goes to the mail box corner. At the counter section, a customer gets the stamps he needs and then, if he is carrying letters, he goes to the mail box corner, otherwise he goes out of the post office through the door. Customers are served at the counter section on a first-come-first-served basis, but the time spent at the counter varies from person to person. At the mail box corner, a customer puts all the letters he has been carrying in the mail box and goes out through the door. A collector also enters the post office through the door and then goes to the mail box corner. At the mail box corner, the collector collects all the mail in the mail box after waiting in the queue, if there is one, and then he carries the collected mail out of the post office through the door. Customers and collectors make a single queue at the mail box corner and arrive and leave the corner on first-in-first-out basis.

We model this post office with five kinds of actors: customer actors, collector actors, the door actor, the counter section actor, and the mail box corner actor. [See Figure 8.1] The movement of customers and collectors is modelled as message-passing where messages are customer and collector actors and targets are the door actor, the counter section actor and the mail box corner actor. Components of the office, collectors and customers

Fig. 8.1.



---

have their own local time. Thus, arrivals of customers and collectors at these components are in general mutually independent. Furthermore, we assume that the walking speed of customers and collectors may vary from person to person. So, for example, a customer arriving at the door after another customer may arrive at the counter section before him. This corresponds to the fact that the actor model of computation assumes nothing about the duration of message-passing except its finiteness. Besides such concurrent events, services at different counters are carried out concurrently, and of course depositing and collecting the mail in the mail box corner takes place independently of the activities at the counter section.

In the subsections that follow, formal specifications of the behavior of each actor will be given and we will state the task specifications that describe the overall properties and effects that are created by the interaction and individual behavior of the component actors.

8.1.2 Interactions at the Door

To formally describe the activities in the post office, first we need to define the states of actors in the model.

For a customer, there are two internal factors which determine his behavior: the letters he carries and the number of stamps he needs at a given time. Thus we express the states of a customer actor by conceptual representations of the following form.

(CUSTOMER (letters: {...}) (#-stamps-needed: ?))

For a collector, the effects of interactions with other actors are expressed by the collected mail. So the state of a collector actor is expressed by conceptual representations of the following form.

(COLLECTOR (collected-mail: {...}))

We cannot define the state of the post office as a whole in terms of the states of its components, because people can be in transit between the components. Customers and collectors may be constantly entering and exiting through the door while other customers and collectors may be changing the states of the mail box corner by depositing and removing the mail. Only the local states of the component actors are well defined. However, we can use the state of the door actor to describe useful aspects of the state of the whole post office if it is defined as below.

The state of the door actor must be defined as an equivalence class of histories of message sent to it. The informal description of the model tells us that customers and collectors arrive at the door when they enter and exit from the post office. So we assume that the door actor accepts four kinds of messages:

(*customer-entering:* <customer>), (*customer-exiting:* <customer>),

(*collector-entering:* <collector>), and (*collector-exiting:* <collector>).

Thus the states of the door actor are defined in terms of these kinds of messages. Since the states of customer and collector actors are well defined at the time they arrive at the door actor, their states can be used to define the state of the door actor. This means that the information available in conceptual representations for customer and collector actors can be used.

We define the state of the door actor at the time of message arrival by

(1) the set of all customers inside the post office,

(2) the set of all collectors inside the post office and

(3) the set of all mail inside the post office.

These three sets are sufficient to characterize useful aspects of the state of the post office as a whole and yet well defined as information local to the door actor, because, for example, the set of mail inside the post office is determined by the difference between letters brought in and letters taken out through the door by customers and collectors. We express the states of the door actor by conceptual representations of the following form. The key word, *POST-OFFICE*, reflects the intention that they serve as the states of the whole post office.

(*POST-OFFICE* (*mail:* {...})(*customers:* {...})(*collectors:* {...}))

A formal specification of the effects of interactions between the door actor and customer and collector actors is depicted in Figure 8.2. One should note the <*caused-event:*...> clauses: After a customer actor arrives at the door actor, a message (*go-to-counter-section-if-necessary:*) instructs him to decide where to go next. Other

Fig. 8.2. A Specification of Interactions at the Door

```
<event: [[the-door <= (customer-entering: C)]]                                          (sp-1)
  <pre-cond:
    (the-door is-a (POST-OFFICE (mail: {!m})(customers: {!cs})(collectors: {!cls})))
    (C is-a (CUSTOMER (letters: {!l})(#-of-stamps-needed: N))) >
  <next-cond:
    (the-door is-a (POST-OFFICE (mail: {!m !l})(customers: {!cs C})(collectors: {!cls})))
    (C is-a (CUSTOMER (letters: {!l})(#-of-stamps-needed: N))) >
  <caused-event: [[C <= (go-to-counter-section-if-necessary:)]] >>

<event: [[the-door <= (customer-exiting: C)]]                                           (sp-2)
  <pre-cond:
    (the-door is-a (POST-OFFICE (mail: {!m1 !l !m2})(customers: {!cs1 C !cs2})(collectors: {!cls})))
    (C is-a (CUSTOMER (letters: {!l})(#-of-stamps-needed: N))) >
  <next-cond:
    (the-door is-a (POST-OFFICE (mail: {!m1 !m2})(customers: {!cs1 !cs2})(collectors: {!cls})))
    (C is-a (CUSTOMER (letters: {!l})(#-of-stamps-needed: N))) >
  <caused-event: [[street <= C]] >>

<event: [[the-door <= (collector-entering: CL)]]                                        (sp-3)
  <pre-cond:
    (the-door is-a (POST-OFFICE (mail: {!m})(customers: {!cs})(collectors: {!cls})))
    (CL is-a (COLLECTOR (collected-mail: {!cm}))) >
  <next-cond:
    (the-door is-a (POST-OFFICE (mail: {!m !cm})(customers: {!cs})(collectors: {!cls CL})))
    (CL is-a (COLLECTOR (collected-mail: {!cm}))) >
  <caused-event: [[mail-box-corner <= (collectors: CL)]] >>

<event: [[the-door <= (collector-exiting: CL)]]                                         (sp-4)
  <pre-cond:
    (the-door is-a (POST-OFFICE (mail: {!m1 !cm !m2})(customers: {!cs})(collectors: {!cls1 CL !cl2})))
    (CL is-a (COLLECTOR (collected-mail: {!cm}))) >
  <next-cond:
    (the-door is-a (POST-OFFICE (mail: {!m1 !m2})(customers: {!cs})(collectors: {!cls1 !cls2})))
    (CL is-a (COLLECTOR (collected-mail: {!cm}))) >
  <caused-event: [[street <= CL]] >>
```

<caused-event:...> clauses indicate where a customer or collector actor is sent after it arrives at the door. In particular, customers and collectors are sent to the **street** actor after they exit from the post office.

8.1.3 Interactions at the Counter Section

Upon entering the post office, a customer must decide where he should go, i.e. to the counter section or the mail box corner. The decision is made in response to a message (*go-to-counter-section-if-necessary:*), according to whether or not he needs stamps. This behavior of the customer is expressed by the following event specification.

```
<event: [[C <= (go-to-counter-section-if-necessary:)]]                    (sp-5)
  (Case-1:
     <pre-cond:
        (C is-a (CUSTOMER (letters: {!l})(#-of-stamps-needed: N)))
        (N > 0) >
     <next-cond:  (C is-a (CUSTOMER (letters: {!l})(#-of-stamps-needed: N)))>
     <caused-event: [[counter-section <= (customer: C)]]  >)
  (Case-2:
     <pre-cond: (C is-a (CUSTOMER (letters: {!l})(#-of-stamps-needed: 0))) >
     <next-cond:  (C is-a (CUSTOMER (letters: {!l})(#-of-stamps-needed: 0)))>
     <caused-event: [[mail-box-corner <= (customer: C)]]  >)>
```

Two points should be made about the specification above. First, the customer C sends himself to the counter section or the mail box corner. Second, the customer C does not change his state as described in the *<next-cond:...>* clauses.

The effects of interaction between customers and the counter section are described by the following simple event specification.

```
<event: [[counter-section <= (customer: C)]                                  (sp-6)
    <pre-cond:  (C is-a (CUSTOMER (letters: {!!})(#-of-stamps-needed: N))) >
    <next-cond:  (C is-a (CUSTOMER (letters: {!!})(#-of-stamps-needed: 0))) >
    <caused-event: [[C <= (go-to-mail-box-corner-if-necessary:)]]  >>
```

This specification might look too simple. Of course, by using conceptual representations for the counter section which include more detailed information, we could express various activities and interactions such as customers waiting in a queue, and buying stamps at a counter. Also, we could define the state of the counter section in a way similar to that in which we defined the sates of the door actor. But for our present purpose, the event specification above is sufficient.

When a customer leaves the counter section, he must again decide where to go next, the mail box corner or the door. The decision is made in response to a message (*go-to-mail-box-if-necessary:*), according to whether or not he is carrying letters. This is expressed as follows.

```
<event: [[C <= (go-to-mail-box-corner-if-necessary:)]]                       (sp-7)
  (Case-1:
    <pre-cond:
       (C is-a (CUSTOMER (letters: {!!})(#-of-stamps-needed: N)))
       ({!!} ≠ {}) >
    <next-cond:  (C is-a (CUSTOMER (letters: {!!})(#-of-stamps-needed: N))) >
    <caused-event: [[mail-box-corner <= (customer: C)]]  >)
  (Case-2:
    <pre-cond:  (C is-a (CUSTOMER (letters: {})(#-of-stamps-needed: N))) >
    <next-cond:  (C is-a (CUSTOMER (letters: {})(#-of-stamps-needed: N))) >
    <caused-event: [[the-door <= (customer-exiting: C)]]  >>
```

Note that no conditions are made for the number of stamps needed N in the preconditions in the above specification. [See, Section 8.1.5.]

8.1.4 Interaction at the Mail Box Corner

To complete the local specifications, we must specify the interaction between the mail box corner and its users. An important fact stated in the informal description of the model is that customers and collectors wait in the same queue before the mail box and that they deposit or collect mail on a first-in-first-out basis. This fact allows us to define the state of the mail box corner by the set of letters brought by the customers who arrived at the mail box corner after the collector who arrived most recently. Letters brought do not necessarily mean letters that are already put in the mail box. They may still be carried by customers in the waiting queue. We use conceptual representations of the following form for the mail box corner. *(MAIL-BOX-CORNER (posted-mail: {...}))* The interaction is described by the event specifications in Figure 8.3.

Fig. 8.3. A Specification of the Interactions at the Mail Box Corner

```
<event: [[mail-box-corner <= (customer: C)]]                              (sp-8)
    <pre-cond:
        (mail-box-corner is-a (MAIL-BOX-CORNER (posted-mail: {!m})))
        (C is-a (CUSTOMER (letters: {!l})(#-of-stamps-needed: N))) >
    <next-cond:
        (mail-box-corner is-a (MAIL-BOX-CORNER (posted-mail: {!m !l})))
        (C is-a (CUSTOMER (letters: {})(#-of-stamps-needed: N))) >
    <caused-event: [[the-door <= (customer-exiting: C)]] >>

<event: [[mail-box-corner <= (collectors: CL)]]                           (sp-9)
    <pre-cond:
        (mail-box-corner is-a (MAIL-BOX-CORNER (posted-mail: {!m})))
        (CL is-a (COLLECTOR (collected-mail: {!cm}))) >
    <next-cond:
        (mail-box-corner is-a (MAIL-BOX-CORNER (posted-mail: {})))
        (CL is-a (COLLECTOR (collected-mail: {!cm !m}))) >
    <caused-event: [[the-door <= (collector-exiting: CL)]] >>
```

8.1.5 Assumptions of No Implicit Interactions

In addition to the above specifications of local interactions, we must make the following assumptions of global nature to describe the post office model completely.

<u>Assumption-I</u>

Customer and collector actors do not receive any messages except those explicitly stated in the event specifications sp-1 to sp-9.

<u>Assumption-II</u>

The counter section actor and the mail box corner actor interact with only the customer and collector actors which have entered through the door. The door actor interacts with only the *(customer-exiting:...)* and *(collector-exiting:...)* messages which contain collector or customer actors which have entered through the door. (No customer or collector actor can arrive directly at these actors without going through the door.)

The first assumption implies that customer or collector actors do not change their states immediately after an event E until the event caused by E, where E is one of the events specified by sp-1 to sp-9. For example, immediately after the event [[<u>counter-section</u> <= (*customer:* C)]], the state of a customer C which is stated in the *<next-cond:...>* clause of the event specification sp-6 do not change until C receives the *(go-to-mail-box-corner...)* message. Thus, in the events specification sp-7, the number N of stamps needed (by the customer C) is zero, because it was zero immediately after [[<u>counter-section</u> <= (*customer:* C)]] as stated in the *<next-cond:...>* clause of sp-6.

8.2 Task Specifications

We have specified the individual behavior and mutual interaction of actors in the post office model. These specifications are local in nature. In this section, we will state some of the overall [global] task specifications of the post office that should be implied by the local specifications. It is important that such task specifications be stated in terms of externally visible actors because the function of the post office should be specified and understood without knowledge of the details of what is going on inside. These actors are the door actor, and customer and collector actors which are outside the post office.

Four task specifications of the post office are in order. For each task specification, an informal statement is followed by the formal one.

The first task specification is expressed in terms of a customer's two states: one before he enters the post office and one after he exits. This may be considered as a specification of the function of the post office from the view point of a customer.

Task-I (Customer is Guaranteed to Return without Letters)

If a customer visits the post office, he must eventually leave there. When he leaves the post office, he must not be carrying letters and he does not need stamps.

```
<event: [[the-door <= (customer-entering: C)]]
    <pre-cond:  (C is-a (CUSTOMER (letters: {!l})(#-of-stamps-needed: N))) >
    <caused-event: [[street <= C]]  >
    <post-cond:  (C is-a (CUSTOMER (letters: {})(#-of-stamps-needed: 0))) >>
```

The second task specification is the collector version of the first one

Task-II (Collector is Guaranteed Not to Lose Any Mail)

If a collector visits the post office, he must eventually leave there. When he leaves the post office, he must be carrying the newly collected mail [which may be empty] in addition to the mail he brought into the post office.

<*event:* ⟦the-door <= (*collector-entering:* CL)⟧
<*pre-cond:* (CL ***is-a*** (***COLLECTOR*** (***collected-mail:*** {!cm1}))) >
<*caused-event:* ⟦street <= CL)⟧ >
<*post-cond:* (CL ***is-a*** (***COLLECTOR*** (***collected-mail:*** {...!cm1...}))) >>

The next task specification is expressed in terms of the interaction between customers and collectors through a set of letters. This may be considered as a specification of the function of the post office from the view point of individual letters.

Task-III (Guaranteed Collection of Mail)

Suppose that a set {!m} of letters is brought into the post office by a customer C.
Then if there is a collector CL who enters the post office after the customer C leaves,
then there always exists a collector CLL (who may be the collector CL) who brings the set {!m} of letters out of the post office to the street.

For an event $E_{\text{c-enter}}$ = ⟦the-door <= (*customer-entering:* C)⟧
where (C ***is-a*** (***CUSTOMER*** (***letters:***{!m})(***#-of-stamps-needed:*** N))),
if there exists an event $E_{\text{cl-enter}}$ = ⟦the-door <= (***collector-entering:*** CL)⟧
such that $E_{\text{c-enter}}$ -act-> $E_{\text{cl-enter}}$ -arr->$_{\text{the-door}}$ $E_{\text{c-exit}}$
where $E_{\text{c-exit}}$ = ⟦the-door <= (***customer-exiting:*** C)⟧,
then there must exist an event $E_{\text{cll-street}}$ = ⟦street <= CLL⟧
such that (CLL ***is-a*** (***COLLECTOR*** (***collected-mail:*** {...!m...}))).

It should be noted that the mail of a customer C could be collected even if no collector enters the post office before C leaves. But in this case there must be some collector which arrives at the mail box corner after C arrives there. (Of course. this cannot be stated in the task specification because the mail box corner which is an internal component of the post office should not be mentioned in the task specifications.)

The next task specification is expressed in terms of the states of the-door (more precisely, sets of mail inside the post office) at different times. This task specification is derived from Task-III.

Task-IV (No Stagnation of Mail)

Let UM, UC and UCL respectively be the set of letters, the set of customers, and the set of collectors inside the post office in a given situation S. If there is a collector CL who enters the post office after all the customers UC and all the collectors UCL (who were inside the post office in the situation S) leave the post office, the set of letters which are inside the post office after the collector CL leaves does not share any letters with the set UM of letters (that were inside the post office in the situation S).

```
Suppose that
  (the-door is-a (POST-OFFICE (mail: {!m})(customers: {!cs})(collectors: {!cls}))) holds
                      in  S = Sit[⟦the-door <= M⟧].
If there exists an event E = ⟦the-door <=  (collector-entering: CL)⟧
    such that
      for any customer $C_i$ in  {!cs} and any collector $CL_j$ in {!cls},
           the following ordering relations hold
                 $E_{c_i}$ -arr->$_{\text{the-door}}$ E  and  $E_{cl_j}$ -arr->$_{\text{the-door}}$ E
              where  $E_{c_i}$ = ⟦the-door <= (customer-exiting: $C_i$)⟧
                     $E_{cl_j}$ = ⟦the-door <= (collector-existing: $CL_j$)⟧,
  then for any event EE = ⟦the-door <= MM⟧
          such that  E -arr->$_{\text{the-door}}$ E' -arr->$_{\text{the-door}}$ EE  or  E' = EE
                      where E' = ⟦the-door <= (collector-exiting: CL)⟧,
    it is the case that
       (the-door is-a (POST-OfFICE (mail: {!mm})(customers: {...})(collectors: {...}))) holds
          in  Sit[EE]   where   {!m} ∩ {!mm} = $\phi$
```

8.3 Verification for the Task Specifications

In this section we will demonstrate that the event specifications, which are given in Sect 8.1 as the description of the behavior of individual actors in the model and their interaction, satisfy the task specifications in the previous section. Also, some of the interesting properties of the event specifications given in Section 8.1 will be revealed in the course of the verification.

8.3.1 Verification for Customer's Guaranteed Return without Letters

First we will verify the following task specification. Some of the properties observed in the process of the verification will be used later in the verification for other task specifications.

Task-I (Customer's Guaranteed Return without Letters)
<*event:* **[[the-door <= (*customer-entering:* C)]]**
<*pre-cond:* **(C *is-a* (*CUSTOMER* (*letters:* {})(*#-of-stamps-needed:* N))) >**
<*caused-event:* **[street <= C]** >
<*post-cond:* **(C *is-a* (*CUSTOMER* (*letters:* {})(*#-of-stamps-needed:* 0))) >>**

(Verification) This task specification is established by tracing sequences of events which involve a customer actor. Such sequences are obtained by checking causal relations among events described by the event specifications given in Sect 8.1. Tracing such a sequence can be done by examining (local) states of actors participating in each event, but certain cautions are necessary in dealing with the state of the-door actor which represents external state of the whole post office. Furthermore, it should be noted in the following demonstration that the reasoning from one event to another crucially depends on Assumption-I in Section 8.1.5. Namely, we assume that the state of a customer C does not change from an event E to the next event caused by E. Below this assumption will be used without being mentioned.

First we assume that an event E_{enter} takes place as described below.

E_{enter}: ⟦the-door <= (*customer-entering*: C)⟧
where (C *is-a* (*CUSTOMER* (*letters*: {!l})(*#-of-stamps-needed*: N)))
(the-door *is-a* (*POST-OFFICE* (*mail*: {!m})(*customers*: {!cs})(*collectors*: {!cls})))

The event E_{enter} and the first assertion are assumed by the task specification to be verified, and the second assertion is assumed in the <*pre-cond:...*> clause in the event specification sp-1. Note that as sp-1 specifies, the state of the-door immediately after this event is expressed as

(the-door *is-a* (*POST-OFFICE* (*mail*: {!m !l})(*customers*: {!cs C})(*collectors*: {!cls})))

which means that the customer C is now inside the post office. The <*caused-event:...*> and <*next-cond:...*> clauses of sp-1 tell us what will happen to C next and what state C will be in.

$E_{\text{decision-1}}$: ⟦C <= (*go-to-counter-section-if-necessary:*)⟧
where (C *is-a* (*CUSTOMER* (*letters*: {!l})(*#-of-stamps-needed*: N)))

To know what event will take place after $E_{\text{decision-1}}$, the event specification sp-5 is referred to. Two cases need to be considered: (1) E_{counter} is caused if $N > 0$ and (2) $E_{\text{mail-box}}$ is caused if $N = 0$.

E_{counter}: ⟦counter-section <= (*customer*: C)⟧
where (C *is-a* (*CUSTOMER* (*letters*: {!l})(*#-of-stamps-needed*: N))), ($N > 0$).

The event specification sp-6 tells that the following event $E_{\text{decision-2}}$ is caused by E_{counter} and that the number of stamps needed becomes zero.

$E_{\text{decision-2}}$: ⟦C <= (*go-to-mail-box-corner-if-necessary:*)⟧
where (C *is-a* (*CUSTOMER* (*letters*: {!l})(*#-of-stamps-needed*: 0)))

To know what event will take place next, the event specification sp-7 is referred to. We need a case analysis: (1) $E_{\text{mail-box}}$ is caused if $l \neq \{\}$ and (2) E_{exit} is caused if $l = \{\}$.

$E_{\text{mail-box}}$: ⟦mail-box-corner <= (*customer*: C)⟧
where (C *is-a* (*CUSTOMER* (*letters*: {!l})(*#-of-stamps-needed*: 0)))

Note that $E_{\text{mail-box}}$ is also caused by $E_{\text{decision-1}}$ as well as $E_{\text{decision-2}}$. Both $E_{\text{decision-1}}$ and $E_{\text{decision-2}}$ insure that the number of stamps needed is zero. On the other hand, the letters {!l} the customer C is carrying may or may not be empty, because $E_{\text{decision-2}}$ insures that l is not empty, but $E_{\text{decision-1}}$ does not. The event specification sp-8 tells us the next event E_{exit}.

E_{exit}: ⟦the-door <= (*customer-exiting:*C)⟧
where (C *is-a* (*CUSTOMER* (*letters:* {})(*#-of-stamps-needed:* N)))
(the-door *is-a* (*POST-OFFICE* (*mail:* {...})(*customers:* {...C...})(*collectors:* {...})))

The first assertion is guaranteed by the <*next-cond:...*> clause of the event specification sp-8. The second assertion that the customer C is still inside the post office must hold in order for the event specification sp-2 to be applied. This assertion is guaranteed by the following facts:

(1) Examining all the event specifications sp-1 through sp-9, events of the form ⟦the-door <= (*customer-exiting:* C)⟧ are the only way for C to exit from the post office (i.e. to eliminate C from the (*customers:* {...}) component of the conceptual representation for the door actor).

(2) An event of the form ⟦the-door <= (*customer-exiting:* C)⟧ have not taken place since C entered the post office.

Now the event specification sp-2 insures the following event E_{street} will happen and the assertion will hold.

E_{street}: ⟦street <= C⟧
where (C *is-a* (*CUSTOMER* (*letters:* {})(*#-of-stamps-needed:* 0)))

The causal relations among the events E_{enter} through E_{street} are illustrated as follows

$$\begin{tikzcd}
E_{enter} \arrow[r] & E_{decision\text{-}1} \arrow[r] \arrow[d, dotted] & E_{mail\text{-}box} \arrow[r] & E_{exit} \arrow[r] & E_{street} \\
& E_{counter} \arrow[r] & E_{decision\text{-}2} \arrow[u, dotted] \arrow[ur, dotted] & &
\end{tikzcd}$$

Since all the event specifications used in the above discussion guarantee that the events given in their <*caused-event:...*> clauses always take place, E_{street} is guaranteed to take place. And the state of the customer C in the situation E_{street} is exactly what is required by the task specification. (End of Verification)

The second task specification given in the previous section can be verified in the same way as above. In fact, applications of the event specifications sp-3, sp-9 and sp-4 in this order will do. It should be noted that in using the event specification sp-4, a justification similar to the one we made, in the reasoning from E_{exit} to E_{street}, for applying the event specification sp-2 is necessary.

8.3.2 Verification for Guaranteed Collection of Mail

Task-III (Guaranteed Collection of Mail)

For an event $E_{\text{c-enter}}$ = ⟦ $\underline{\text{the-door}}$ <= (*customer-entering:* C)⟧
where (C *is-a* (*CUSTOMER* (*letters:*{!m})(*#-of-stamps-needed:* ?))),
if there exists an event $E_{\text{cl-enter}}$ = ⟦ $\underline{\text{the-door}}$ <= (*collector-entering:* CL)⟧
such that $E_{\text{c-enter}}$ -act-> $E_{\text{c-exit}}$ $\text{--->}_{\text{the-door}}$ $E_{\text{cl-enter}}$
where $E_{\text{c-exit}}$ = ⟦ $\underline{\text{the-door}}$ <= (*customer-exiting:* C)⟧,
then there must exist an event E = ⟦ $\underline{\text{street}}$ <= CLL⟧
such that (CLL *is-a* (*COLLECTOR* (*collected-mail:* {...!m...}))) holds.

To verify this task specification, we rely on the following lemma which is easily derived from the event specifications given in Sect 8.1. This lemma guarantees that if a customer enters the post office carrying a set {!l} of letters, he always arrives at the mail box corner carrying the same set of mail.

Lemma

For an event $E_{\text{c-enter}}$ = ⟦ $\underline{\text{the-door}}$ <= (*customer-entering:* C)⟧
where (C *is-a* (*CUSTOMER* (*letters:* {!l})(*#-of-stamps-needed:* ?))),
there always exists an event $E_{\text{c-mail-box}}$ = ⟦ $\underline{\text{mail-box-corner}}$ <= (*customer:* C)⟧
where (C *is-a* (*CUSTOMER* (*letters:* {!l})(*#-of-stamps-needed:* ?)))
such that $E_{\text{c-enter}}$ -act-> $E_{\text{c-mail-box}}$.

This was justified during the verification of the first task specification.

[Note that E_{enter} ---> $E_{\text{mail-box}}$ in the demonstration of Task-I.]

(Verification of Task-III)

Suppose that an event $E_{c\text{-enter}}$ = ⟦the-door <= (*customer-entering:* C)⟧ takes place where

(C *is-a* (*CUSTOMER* (*letters:* {!l})(*#-of-stamps-needed:* ?)))

holds. By the above lemma, an event $E_{c\text{-mail-box}}$ = ⟦mail-box-corner <= (*customer:* C)⟧ always takes place and the same assertion

(C *is-a* (*CUSTOMER* (*letters:* {!l})(*#-of-stamps-needed:* ?)))

still holds. Here we assume that the following assertion holds when $E_{c\text{-mail-box}}$ takes place.

(mail-box-corner *is-a* (*MAIL-BOX-CORNER* (*posted-mail:* {!pm}))).

Then, by the event specification sp-8, the assertion

(mail-box-corner *is-a* (*MAIL-BOX-CORNER* (*posted-mail:* {!pm !l})))

holds immediately after $E_{c\text{-mail-box}}$ and until the next message arrival at the mail-box-corner. Sp-8 also guarantees that $E_{c\text{-exit}}$ = ⟦the-door <= (*customer-exiting:* C)⟧ will take place.

Then suppose that the following event takes place after $E_{c\text{-exit}}$

$E_{cl\text{-enter}}$ =⟦the-door <= (*collector-entering:* CL)⟧

where (CL *is-a* (*COLLECTOR* (*collected-mail:* {!cm}))) holds. By the event specification sp-3,

$E_{cl\text{-mail-box}}$ = ⟦mail-box-corner <= (*collectors:* CL)⟧

takes place where (CL *is-a* (*COLLECTOR* (*collected-mail:* {!cm}))) still holds. At this point, the ordering of the events which have already occurred is expressed as follows.

$$E_{c\text{-enter}} \xrightarrow{act} E_{c\text{-mail-box}} \xrightarrow{act} E_{c\text{-exit}} \xrightarrow{arr}_{\text{the-door}} E_{cl\text{-enter}} \xrightarrow{act} E_{cl\text{-mail-box}}$$

The important fact here is that $E_{c\text{-mail-box}}$ precedes $E_{cl\text{-mail-box}}$. We shall consider two cases:

Case-1: If any collectors do not arrive at the mail box corner between $E_{c\text{-mail-box}}$ and $E_{cl\text{-mail-box}}$, the state of the mail box corner at the time of $E_{cl\text{-mail-box}}$ is expressed as

(mail-box-corner *is-a* (*MAIL-BOX-CORNER* (*posted-mail:* {...!pm...!l...})))

because customers arriving between $E_{c\text{-mail-box}}$ and $E_{cl\text{-mail-box}}$ only deposit, but never collect mail. Then as the event specification sp-9 states, the collector CL collects all the mail {...!pm...!l...} and then go to the door.

Case-2: If there are collectors who arrive at the mail box corner between $E_{c\text{-mail-box}}$ and $E_{cl\text{-mail-box}}$, then the first one among such collectors will collect the mail which includes {!l} and {!pm} and then go to the door.

In both cases, some collector carrying {!l}, say CLL, arrives at the door from the mail box. To insure that the collector CLL goes out to the street, the two assertions given in the <*pre-cond:...*> clause of the event specification sp-4 must be satisfied. One assertion says that CLL must be one of the collectors who appear in the conceptual representation of the door actor at the time CLL arrives, namely, the following must hold.

(the-door *is-a* (*POST-OFFICE* (*mail:*{...})(*customers:*{...})(*collectors:*{...CLL...}))).

Assumption-II in Section 8.1.5 guarantees that this assertion holds, because it assumes that all the collectors arriving at the door from the mail box corner must have entered through the door, so by sp-3 CLL must appear in the (*collectors:...*) component of the conceptual representation of the door. This completes the verification. Note that Assumption-I was used throughout the above demonstration. (End of Verification)

The last task specification "No Stagnation of Mail" can be verified by using already established task specifications. As was done in this task specification, let us suppose that the state of the post office is expressed by the following assertion.

(the-door *is-a* (*POST-OFFICE* (*mail:* {!m})(*customers:* {!cs})(*collectors:* {!cls})))

Then it is the case that every letter l which is an element of the mail {!m} inside the post office is brought in either by a customer or by a collector. If l is brought in by a customer, we can use the third task specification which has been just established above. If l is brought in by a collector, the second task specification "Collector is Guaranteed Not to Lose Any Mail" insures that l will be brought out by the same collector that brought l into the post office. So both cases are proved.

9. Conclusions and Future Research

In this thesis, we have presented the *local state approach* to specification and verification techniques for both serial and parallel computations. As stated in the Introduction (Chapter 1), the work reported here has made four major technical contributions. In concluding the thesis, we would like to first review these contributions and then discuss their implications in the light of our projections for future research.

9.1 Summary and Conclusions

As was demonstrated in Chapters 4 and 6, the local state approach provides powerful and convenient specification techniques for *abstract data types* with *parallelism* and *side-effects* with which previous techniques had failed to deal.

As the post office model in Chapter 8 illustrates, specification techniques based on local states enable us to describe the complex internal concurrent activities of a system, such as an operating system or a multi-user data base system, in terms of the individual behavior of its subsystems and their mutual interaction. In order to express the overall functional behavior of such systems (task specifications), the use of local states turns out to be not only useful, but crucial. In addition, however, we sometimes need to state *temporal ordering constraints* among events that are difficult to express in terms of the state changes of individual subsystems. For this purpose we have used an event-oriented specification language[Greif-Hewitt75, Hewitt-Baker77] in which the ordering concepts in the underlying computation model can be talked about directly. Thus, with the complementary use of the ordering constraint statements, the effectiveness and versatility of the local state approach in specifying the behavior of systems with high internal concurrency is strengthened.

To describe the states of individual data and procedural objects, we have developed a system of notation called *conceptual representations.* Based on this notational device, we have presented a formalism for specification and verification. As was seen throughout the thesis, this formalism allows us to express states of individual objects *directly* and *explicitly*. Thus we believe that specifications written in our formalism are easy to understand and are less error-prone in their *completeness* and *consistency*, as compared with those written in other formalisms. Moreover, the separation of the states of an object from its identity makes it possible for conceptual representations to express *sharing* structures among objects and *multiple instances* of a class of objects.

The ability of our formalism to express sharing structures and multiple class instantiation enabled us to develop a method for symbolic evaluation of programs written in object-oriented languages, which has not been attempted before. The developed method is used for verification of serial computations and has suggested an approach to mechanical program analysis (Section 5.4, Chapter 5).

9.2 Future Research

We have defined the states of an individual object (actor) as *equivalence classes* on the past histories of messages (operations) sent to the object. Local states thus defined are expressed by conceptual representations which mathematically comprise sequences, collections and tuples. On the other hand, the state of an object can be identified with a mathematical function which is obtained as a solution of the behavioral equations introduced in Section 6.4, Chapter 6. So far the relationships between the above two interpretations of states have not been made clear. We foresee that the investigation of these relations will reveal very rich mathematical structures and that, consequently, the properties of *implementation invariants* (Section 5.3.1, Chapter 5) which we have left informal will be understood precisely.

The techniques exemplified by the model of a simple post office can be applied to the specification and verification of various distributed information processing systems. Furthermore, the techniques used in this thesis have a direct application in the area of business automation. We expect that actor-like procedural objects will enormously increase the flexibility and security of message and document systems by replacing "paper" forms and letters and "paper" documents with "active" (procedural) counterparts that are sent to work stations in computer networks. Moreover, we can apply our techniques to the specification and verification of object-oriented simulation and system description languages such as the DELTA system[Holbaek-Hassen-et-al77].

The verification process for *parallel* computations described in this thesis is informal. The formalization of such a process is desirable. For this purpose, a formal specification language in which <u>both</u> local states of objects and ordering constraints of events can be expressed in a coherent fashion must be developed, together with sound and powerful inference rules which are effective in dealing with the *partial* ordering of events. With such a formal system available, we will be able to construct practically useful software tools which assist us in the construction of *parallel* programs and distributed message passing systems. Various important properties, such as no-deadlock, no-starvation, and the property that a system meets its specifications, will be mechanically analyzed with such software tools.

10. Bibliography

[Ashcroft75] E. A. Ashcroft, "Proving Assertions about Parallel Programs" JCSS, Vol.10, pp.110-135, 1975.

[Atkinson-Hewitt77] R. Atkinson and C. Hewitt, "Synchronization in Actor Systems" SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Los Angles, January, 1977.

[Berzins76] V. Berzins, Proposal for Ph.D thesis research, submitted to Department of Electrical Engineering and Computer Science, MIT, 1976.

[Boyer-et-al75] R. S. Boyer, B. Elspas, and K. N. Levitt, "SELECT -- A Formal System for Testing and Debugging Programs by Symbolic Execution" International Conference on Reliable Software, Los Angles, 1975.

[Boyer-Moore75] R. S. Boyer, and J. S. Moore, "Proving Theorems about LISP Functions" JACM, Vol.22, No.1, January, 1975.

[Burstall72] R. M. Burstall, "Some Techniques for Proving Correctness of Programs Which Alter Data Structures" *Machine Intelligence 7*, Edinburgh University Press, Edinburgh, 1972.

[Burstall-Darlington75] R. M. Burstall, and J. Darlington, "Some Transformations for Developing Recursive Functions" International Conference on Reliable Software, Los Angles, April, 1975.

[Clint73] M. Clint, "Program Proving: Coroutines" Acta Informatica, Vol.2, pp.50-63, 1973.

[Cohen75] E. S. Cohen "A Semantic Model for Parallel Systems with Scheduling" ACM SIGPLAN-SIGACT Conference, Palo Alto, California, January, 1975.

[Dahl-et-al70] O. J. Dahl, B. Myhrhang, and K. Nygaard, "The SIMULA-67: Common Base Language" Publication S-22, Norwegian Computing Center, Oslo, 1970.
also, G. Birtwistle, O. J. Dahl, B. Myhrhang, and K. Nygaard, *SIMULA Begin* Auerbach, Philadelphia, 1973.

[Deutch73] L. P. Deutch, "An Interactive Program Verifier" Ph.D Thesis. University of California at Berkeley, June, 1973.

[Dijkstra76] E. W. Dijkstra, *A Discipline of Programming* Prentice-Hall, Englewood Cliffs, N.J., 1976

[Flon-Suzuki77] L. Flon and N. Suzuki, "Nondeterminism and the Correctness of Parallel Programs" IFIP Working Conference, New Brunswick, 1977.

[Floyd67] R. W. Floyd, "Assigning Meaning to Programs" in J. T. Schwartz (Ed.) *Mathematical Aspect of Computer Science*, American Mathematical Society, Providence, Rhode Island, 1967.

[Greif75] I. Greif, "Semantics of Communicating Parallel Processes" Ph.D Thesis, MIT, also Technical Report TR-154, Laboratory for Computer Science (formerly Project MAC), September, 1975.

[Greif-Hewitt75] I. Greif, and C. Hewitt, "Actor Semantics of PLANNER-73" ACM SIGPLAN-SIGACT Conference, Palo Alto, California, January, 1975.

[Guttag75] J. V. Guttag, "The specification and Applications to Programming of Abstract Data Types" Ph.D Thesis, University of Toronto, also Computer System Research Group Report CSRG-59, 1975.

[Hayes73] P. Hayes, "The Frame Problem and Related Problems in Artificial Intelligence" in A. Elithorn and D. Jones (Ed.) Artificial Intelligence and Human Thinking, Jossey-Bass Inc., 1973.

[Hewitt75] C. Hewitt, "How to Use What You Know" International Joint Conference on Artificial Intelligence, USSR, September, 1975.

[Hewitt77] C. Hewitt, "Viewing Control Structures as Patterns of Passing Messages" Journal of Artificial Intelligence, Vol.8, pp.323-364, 1977.

[Hewitt-Baker77] C. Hewitt and H. Baker Jr., "Laws for Communicating Parallel Processes" IFIP-77, Toronto, August, 1977.

[Hewitt-Smith75] C. Hewitt and B. C. Smith, "Towards a Programming Apprentice" IEEE Transaction on Software Engineering, Vol.SE-1, No.1, March, 1975.

[Hoare69] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming" CACM Vol.12, October, 1969.

[Hoare72] C. A. R. Hoare, "Proof of Correctness of Data Representation" Acta Informatica, Vol.1, pp.271-281, 1972.

[Holbaek-Hanssen-et-at77] E. Holbaek-Hanssen, P. Handlykken, and K. Nygaard, "System Description and the DELTA language" Publication No.523, Norwegian Computing Center, 1977.

[Igarashi-et-al75] S. Igarashi, R. L. London, and D. C. Luckham, "Automatic Program Verification I: A Logical Basis and its Implementation" Acta Informatica, Vol.4, pp.145-182, 1975.

[Kahn74] G. Kahn, "The Semantics of A Simple Language for Parallel Programming" IFIP-74, Stockholm, 1974.

[Keller76] R. M. Keller, "Formal Verification of Parallel Programs" CACM, Vol.19, No.7, July, 1975,.

[King69] J. King, "A Program Verifier" Ph.D Thesis, Carnegie-Mellon University, 1969.

[King76] J. King, "Symbolic Execution and Program Testing" CACM, Vol.19, No.7, July, 1976.

[Learning-Research-Group76] Learning Research Group, "Personal Dynamic Media" SSL-76-1. Xerox Palo Alto Research Center, April, 1976

[Liskov-Zilles74] B. Liskov and S. Zilles, "Programming with Abstract Data Types" ACM SIGPLAN Conference on Very High Level Languages, SIGPLAN NOTICE, Vol.9, No.4, April, 1974.

[Liskov-Zilles75] B. Liskov. and S. Zilles, "Specification Techniques for Data Abstractions" IEEE Transactions on Software Engineering, Vol.SE-1, No.1, March, 1975.

[Liskov-Berzins77] B. Liskov and V. Berzins, "An Appraisal of Program Specifications" Computation Group Memo, No.141-1, Laboratory for Computer Science, MIT, 1977., also to appear in P. Wegner (Ed.) Research Directions in Software Technology, MIT Press, Cambridge, 1978.

[Lamport77] L. Lamport, "Proving the Correctness of Multiprocess Programs" IEEE Transactions on Software Engineering, Vol.SE-3, No.2, 1976.

[London-et-al76] R. L. London, M. Shaw, and Wm. A. Wulf, "Abstraction and Verification in ALPHARD: A Symbol Table Example" Department of Computer Science, Carnegie-Mellon University, December, 1976.

[Majster77] M. E. Majster, "Limits of the 'Algebraic' Specification of Abstract Data Types" SIGPLAN NOTICE, Vol.12, No.10, 1977.

[Manna69] Z. Manna, "The Correctness of Programs" JCSS, Vol.3, pp.119-127, 1969.

[McCarthy-Hayes69] J. McCarthy and P. Hayes, "Some Philosophical Problems from the Standpoint of Artificial Intelligence" Machine Intelligence 4, American Elsevier, New York, 1969.

[Milner73] R. Milner, "Processes: A Mathematical Model of Computing Agents" Logic Colloquium, Bristol, England, 1973.

[Nakajima-et-al77] R. Nakajima, M. Honda, and H. Nakahara, "Describing and Verifying Programs with Abstract Types" IFIP Working Conference, New Brunswick, 1977.

[Owicki75] S. Owicki, "Axiomatic Techniques for Parallel Programs" Ph.D Thesis, Department of Computer Science, Cornell University, 1975.

[Owicki-Gries76] S. Owicki and D. Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach" CACM, Vol.19, No.5, 1976.

[Parnas72] D. L. Parnas, "A Technique for the Specification of Software Modules with Examples" CACM, Vol.15, No.5, 1972.

[Rich-Shrobe76] C. Rich, and H. E. Shrobe, "Initial Report on a LISP Programmer's Apprentice" AI-TR No.354, Artificial Intelligence Laboratory, MIT, December, 1976

[Sandwall72] E. Sandwall, "An Approach to the Frame Problem and its Implementation" in Machine Intelligence 7, Edinburgh University Press, Edinburgh, 1972.

[Scott72] D. Scott, "Lattice Theoretic Models for Various Type-free Caluculi" 4-th International Congress in Logic, Methodology and Philosophy of Science, Bucharest, 1972.

[Schaffert-et-al75] C. Schaffert, A. Snyder, and R. Atkinson, "The CLU Reference Manual" Laboratory for Computer Science (formaly Project MAC), MIT, September, 1975

[Spitzen-Wegbreit75] J. Spitzen, and B. Wegbreit, "The Verification and Synthesis of Data Structures." Acta Informatica, Vol.4, pp.127-144, 1975.

[Steiger74] Steiger, R. "Actor Machine Architecture" Master Thesis, Department of Electrical Engineering and Computer Science, MIT, June, 1974.

[Suzuki75] N. Suzuki, "Automatic Program Verification II: Verifying Programs by Algebraic and Logical Reduction" International Conference on Reliable Software, Los Angles, April, 1975.

[Waldinger77] R. Waldinger "Achieving Several Goals Simultaneously" in Machine Intelligence 8 Ellis Horwood Ltd., Chichester, 1977.

[Wegbreit-Spitzen76] B. Wegbreit, and J. M. Spitzen, "Proving Properties of Complex Data Structures" JACM, Vol.23, No.2, April, 1976.

[Wulf-et-al76] Wm. A. Wulf, R. L. London, and M. Shaw, "Introduction to the Construction and Verification of ALPHARD Programs" IEEE Transactions on Software Engineering, Vol.SE-2, No.4 1976.

[Yonezawa75] A. Yonezawa, "Meta-evaluation of Actors with Side-effects" Working paper No.101, Artificial Intelligence Laboratory, MIT, June, 1975.

[Yonezawa-Hewitt76] A. Yonezawa, and C. Hewitt, "Symbolic Evaluation using Conceptual Representations for Programs with Side-effects." AI-Memo, No.399, Artificial Intelligence Laboratory, MIT, December, 1976.

[Yonezawa-Hewitt77] A. Yonezawa, and C. Hewitt, "Modelling Distributed Systems" International Joint Conference on Artificial Intelligence, Cambridge, August, 1977., also to appear in Machine Intelligence 9, Edinburgh University Press, Edinburgh, 1978.

[Zilles74] S. Zilles, "Algebraic Specifications of Data Types" Project MAC Progress Report Vol. 11, pp.52-58, MIT, Cambridge, 1974.

Appendix I – Derivation of Axiom (5)

The following axiom which was given in the algebraic specification of queues in Figure 2.6, Chapter 2.

Axiom (5)
if ¬IS-EMPTY(Q) ∧ DEQUEUE(Q, A) = <B, Q'>
then DEQUEUE(ENQUEUE(Q, A)) = <B, ENQUEUE(Q', A)>

This is derived from the following specification of queues based on conceptual representations [which is identical to the one given in Figure 2.2, Chapter 2, except that the functionality of the operations is omitted].

(E1) CREATE-QUEUE() ----> (*QUEUE* [])

(E2) ENQUEUE((*QUEUE* [!x]), A) ----> (*QUEUE* [!x A])

(E3) DEQUEUE((*QUEUE* [])) ----> *ERROR*

(E4) DEQUEUE((*QUEUE* [A !x])) ----> <A , (*QUEUE* [!x])>

(E5) IS-EMPTY((*QUEUE* [])) ----> *TRUE*

(E6) IS-EMPTY((*QUEUE* [A !x])) ----> *FALSE*

(Derivation)

(1) ¬IS-EMPTY(Q) ;given as the premise of the axiom.

(2) DEQUEUE(Q) = <B, Q'> ;given as the premise of the axiom.

From (1) and (E6), Q must be of the form

(*QUEUE* [**front-element !rest**])

From (2) and (E4), **front-element = B** and Q' contains [**!rest**]. Thus (3) and (4) hold.

(3) Q = (*QUEUE* [B !rest])

(4) Q' = (*QUEUE* [!rest])

(5) DEQUEUE(ENQUEUE(Q, A)) ;given in the consequence of the axiom.

= DEQUEUE(ENQUEUE((*QUEUE* [B !rest]), A) ;from (3).

= DEQUEUE((*QUEUE* [B !rest A])) ;from (E2).

= <B, (*QUEUE* [!rest A]))> ;from (E4).

= <B, ENQUEUE((*QUEUE* [!rest]), A)> ;from (E2).

= <B, ENQUEUE(Q', A)> ;from (4).

Hence, DEQUEUE(ENQUEUE(Q, A)) = <B, ENQUEUE(Q', A)> (End of Derivation)

Appendix II – Limits of Algebraic Specification

To show the existence of abstract data types which cannot be expressed by a finite set of axioms in the algebraic approach, M. E. Majster[1977] gave a stack type which allows us to look at any stack elements by using a position information i. The functionality of this type is as follows.

CREATE: ---> *stack* ;creates an empty stack.

PUSH: *stack* X *item* ---> *stack* or *error*
;tries to insert an item at the top.
;if i is not pointing to the top, undefined
;otherwise i points to the new top item.

DOWN: *stack* ---> *stack* or *error*
;tries to increment i by one.
;if i already points to the bottom item, error.

POP: *stack* ---> *stack* or *error*
;tries to remove the top item.
;if i is not pointing to the top, error
;otherwise, i points to the new top item.

READ: *stack* ---> *stack* or *error*
;tries to read the item pointed by i.
;if stack is empty, error.

RETURN: *stack* ---> *stack* or *error*
;tries to cause i to point to the top item.
;if stack is empty, error.

Unfortunately, the axioms for these operations cannot be characterized finitely. For example, we need infinitely many axioms expressed as follows.

$$\mathrm{RETURN}(\mathrm{DOWN})^m(\mathrm{PUSH})^n(i_1,\dots,i_n) = (\mathrm{PUSH})^n(i_1,\dots,i_n)$$

for all $m > 0$ and $m < n$

where $\mathrm{PUSH}^n(i_1,\dots,i_n) = \mathrm{PUSH}(\dots\mathrm{PUSH}(\mathrm{CREATE}(), i_1)\dots, i_n)$

This data type can be easily specified by using conceptual representations of the following form.

(*STACK*(***position:*** i)(***items:*** [...]))

The (*position:...*) component keeps the position information and the conceptual sequence in the (*items:...*) represents stack elements. A specification based on the conceptual representations is given below.

(1) CREATE() ---> (*STACK*(***position:*** 1)(***items:*** []))

(2) PUSH((*STACK*(***position:*** i)(***items:*** [!s])), I)
if i = 1 ---> (*STACK*(***position:*** i)(***items:*** [I !s]))
otherwise ---> *ERROR*

(3) DOWN((*STACK*(***position:*** i)(***items:*** [!s]))
if i < length[!s] ---> (*STACK*(***position:*** i + 1)(***items:*** [!s]))
otherwise ---> *ERROR*

```
(4)  POP((STACK(position: i)(items: [!s]))
          if i = 1 and s = [I !rest]   --->  (STACK(position: i)(items: [!rest]))
          otherwise    ---> ERROR

(5)  READ((STACK(position: ?)(items: []))  --->  ERROR

(6)  READ((STACK(position: i)(items: [!x1 I !x2]))  --->  I
                    where  length[!x1] = i - 1

(7)  RETURN((STACK(position: i)(items: [!s]))
          if s = []   ---> ERROR
          otherwise   ---> (STACK(position: 1)(items: [!s]))
```

Appendix III - Recursion, Iteration and Loop Invariants

The handling of recursive invocations of modules in symbolic evaluation has been illustrated in the example of **empty-one-queue-into-another** in Section 5.2.1, Chapter 5. In general, recursive invocations are treated as the same as ordinary invocations of modules. When a [recursive] invocation of a module M is encountered in symbolic evaluation, the contract of M is referred to and the specified results and postconditions are used to continue the symbolic evaluation after making sure that all the preconditions of M are satisfied.

Iterations in implementations can be handled almost in the same way, because the iteration construct in PLASMA allows us to treat an iteration as a module. Thus if specifications of such modules are supplied, loops can be treated as ordinary modules.

Another way of dealing with iterations is to rely on assertions which hold every time the control reaches the beginning point of a loop. Such assertions are called *loop invariants* or *inductive assertions*[Floyd67, Hoare69]. Since loop invariants are usually not derived from the process of symbolic evaluation, they must be supplied externally. Symbolic evaluation of the part of a code which follows such assertions is carried out under the assumption that the assertions hold in the situation corresponding to the beginning point of the loop. To illustrate this technique, we will consider a simple example.

Fig. III.1. An Iterative Version of **empty-one-queue-into-another**

```
(empty-one-queue-into-another-a ≡
  (≡> [=q1 =q2]
    ([q1 q2] =>
      (loop ≡
        (≡> [=qq1 =qq2]
              **
          (rules (qq1 <= (dq:))
            (≡> (exhausted:)

                - $S_{exhausted-qq1}$ -
                (done: [qq1 qq2]))
            (≡> [=front-of-qq1 =dequeued-qq1]

                - $S_{dequeued-qq1}$ -
                (qq2 <= (nq: front-of-qq1))
                (loop <= [dequeued-qq1 qq2])) ))))))
```

In Figure III.1, an iterative version of **empty-one-queue-into-another-a** is given. The loop invariant for **loop** which holds at the point where ** is placed in the code is

[!xx1 !xx2] = [!x1 !x2]

where xx1 and xx2 are the elements of the impure queues which are bound to qq1 and qq2, respectively, and x1 and x2 are the elements of the impure queues bound to q1 and q2, respectively. This invariant is expressed in our formalism as follows.

<*loop-Invariant:* [!xx1 !xx2] = [!x1 !x2]
where
in *Sit*[⟦loop <= [QQ1 QQ2]⟧]
(QQ1 *is-a* (*IMPURE-QUEUE* [!xx1]))
(QQ2 *is-a* (*IMPURE-QUEUE* [!xx2])),

in *Sit*[⟦empty-one-queue-into-another-a <= [Q1 Q2]⟧]
(Q1 *is-a* (*IMPURE-QUEUE* [!x1]))
(Q2 *is-a* (*IMPURE-QUEUE* [!x2])) >

Given the above invariant, it is easily demonstrated that the implementation in Figure III.1 satisfies the contract for **empty-one-queue-into-another** given in Figure 5.5 in Chapter 5. The key point of the demonstration is that when the control reaches $\mathbf{S}_{\textbf{exhausted-qq1}}$, the impure queue QQ1 that qq1 is bound to is empty, i.e. xx1 = []. Therefore, the elements of the impure queue QQ2 that qq2 is bound to, which are expressed as xx2, are equal to [!x1 !x2] because [!xx1 !xx2] = [!x1 !x2] (from the invariant), and xx1 = [] imply xx2 = [!x1 !x2]. The rest of the demonstration can be carried out almost in the same way as that for the recursive version shown in Section 5.2.1, Chapter 5.

Appendix IV - Convergence of **empty-one-queue-into-another**

Most event specifications written in our specification language contain *<caused-event:...>* or *<return:...>* clauses. As explained in Section 4.3.1, Chapter 4, the existence of these clauses in an event specification indicates that an event E stated in such a clause is required to take place. Thus, to verify an implementation against specifications, we have to demonstrate that the event E always takes place, as well as that the postconditions are satisfied.

As an example, let us consider the convergence of the implementation of **empty-one-queue-into-another** [hereafter **empty**] given in Figure 5.5 in Chapter 5. [The following discussion is based on the symbolic evaluation of the implementation presented in Section 5.2.1, Chapter 5.] For the demonstration of the convergence, we need to show that the control always reaches the situation $\mathbf{S}_{\text{exhausted-q1}}$, provided that the two actors sent to **empty** are distinct and both are impure queue actors.

If the impure queue bound to **q1** becomes empty during the recursive invocation of **empty**, $\mathbf{S}_{\text{exhausted-q1}}$ can be reached. Thus it is sufficient to show that the length of the impure queue eventually becomes zero. Since the length of the impure queue is an arbitrary non-negative integer when it arrives at **empty** for the first time, we need to show that its length decreases at its every subsequent arrival at **empty**. What has to be shown can be stated in our formalism as follows.

(*length-of*(q1) *in* $\mathbf{S}_{\text{received-queues}}$)
is-greater-than (*)
(*length-of*(dequeue-q1) *in* $\mathbf{S}_{\text{enqueued-q2}}$)

To show this, the situational tree produced by the symbolic evaluation of the implementation is examined. *Length-of* on impure queues is defined as

<*property*: *length-of*(Q) ≡ length(x)
where (Q *is-a* (*IMPURE-QUEUE* [!x])) >

By using assertions about Q1 and Q2 in conjunction with the binding information for q1 and dequeued-q1, we obtain the following facts.

length-of(q1) = length(x1) *in* $\mathbf{S}_{\text{received-queues}}$,
length-of(dequeue-q1) = length(y) *in* $\mathbf{S}_{\text{enqueued-q2}}$

Since x1 = [B !y] holds, the desired relation (*) is shown.

Note that the precondition that Q1 and Q2 are distinct actors was used in obtaining the assertion about the state of Q1 in $\mathbf{S}_{\text{enqueue-q2}}$. This precondition guarantees that ⟦Q2 <= (*nq*:...)⟧ does not change the state of Q1, and hence that assertion could be inherited from $\mathbf{S}_{\text{dequeued-q1}}$

Appendix V - Another Specification of One-a-at-Time Serializers

Another specification of *one-at-a-time* serializers is given as the following four event specifications. The first event specification is concerned with the creation of a one-at-a-time serializer. The second one describes the event where the serializer receives a request. A buck passer actor BP is created and placed in the crowd. Note in (*Case-1*:...) clause that BP is sent to the resource R as the continuation of the message in the caused event. A reply from the resource is always sent to a buck passer BP. This is described in the third event specification. Then the buck passer sends the reply from the resource to the serializer G which created BP. The fourth event specification describes how the reply sent from the buck passer is handled by a serializer.

```
<event: ⟦create-one-at-a-time <= R⟧
    <return:  G* >
    <post-cond:  (G is-a (ONE-AT-A-TIME (queue: [])(crowd: {})(resource: R)))  >>

<event: ⟦G <== M⟧
    where  M = [request: RQ  reply-to: C]
  (Case-1:
   <pre-cond:  (G is-a (ONE-AT-A-TIME (queue: [])(crowd: {})(resource: R))) >
   <next-cond:
     (G is-a (ONE-AT-A-TIME (queue: [])(crowd: {BP*})(resource: R)))
     (BP is-a (BUCK-PASSER (continuation: C)(serializer: G))) >
   <caused-events: ⟦R <== [request: RQ  reply-to: BP]⟧ >)
  (Case-2:
   <pre-cond:  (G is-a (ONE-AT-A-TIME (queue: [!x])(crowd: {BP})(resource: R))) >
   <next-cond:  (G is-a (ONE-AT-A-TIME (queue: [!x M])(crowd: {BP})(resource: R))) >
   <caused-events: {}  >)>
```

```
<event: [[BP <== [reply: A]]
   <pre-cond: (BP is-a (BUCK-PASSER (continuation: C)(serializer: G))) >
   <caused-event: [[G <== [reply: (buck: A (continuation: C) (buck-passer: BP))]]>>

<event: [[G <== [reply: (buck: A (continuation: C) (buck-passer: BP))]]
  (Case-1:
   <pre-cond:  (G is-a (ONE-AT-A-TIME (queue: [])(crowd: {BP})(resource: R))) >
   <next-cond:  (G is-a (ONE-AT-A-TIME (queue: [])(crowd: {})(resource: R))) >
   <caused-events: [[C <== [reply: A]] >)
  (Case-2:
   <pre-cond:
     (G is-a (ONE-AT-A-TIME (queue: [WM !x])(crowd: {BP})(resource: R)))
     (WM = [request: RQ reply-to: CC])  >
   <next-cond:
     (G is-a (ONE-AT-A-TIME (queue: [!x])(crowd: {NBP*})(resource: R)))
     (NBP is-a (BUCK-PASSER (continuation: CC)(serializer: G)))) >
   <caused-events: { [[C <== [reply: A]] , [[R <== [request: RQ reply-to: NBP]] }  >)>
```

CS-TR Scanning Project
Document Control Form

Date : 10 / 26 / 95

Report # LCS-TR-191

Each of the following should be identified by a checkmark:
Originating Department:

☐ Artificial Intellegence Laboratory (AI)
☒ Laboratory for Computer Science (LCS)

Document Type:

☒ Technical Report (TR) ☐ Technical Memo (TM)
☐ Other:

Document Information

Number of pages: 222 (228-IMAGES)
Not to include DOD forms, printer intstructions, etc... original pages only.

Originals are:
☐ Single-sided or
☒ Double-sided

Intended to be printed as :
☐ Single-sided or
☒ Double-sided

Print type:
☐ Typewriter ☐ Offset Press ☐ Laser Print
☐ InkJet Printer ☐ Unknown ☐ Other:

Check each if included with document:

☐ DOD Form ☐ Funding Agent Form ☒ Cover Page
☒ Spine ☐ Printers Notes ☐ Photo negatives
☐ Other:

Page Data:

Blank Pages(by page number): FOLLOWS LAST PAGE

Photographs/Tonal Material (by page number):

Other (note description/page number):

Description : Page Number:
IMAGE MAP: (1-222) UN#'ED TITLE PAGE, 2-221, UN#BLANK
(223-228) SCANCONTROL, COVER, SPINE, TRGT'S(3)

Scanning Agent Signoff:
Date Received: 10/26/95 Date Scanned: 11/16/95 Date Returned: 11/22/95

Scanning Agent Signature: Michael W. Cook

Rev 9/94 DS/LCS Document Control Form cstrform.vsd