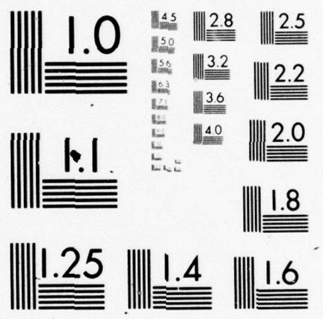
AD-A048 980 MASSACHUSETTS INST OF TECH CAMBRIDGE LAB FOR COMPUTE--ETC F/G 17/2
FORMAL SPECIFICATIONS FOR PACKET COMMUNICATION SYSTEMS.(U)
NOV 77 D J ELLIS N00014-75-C-0661
UNCLASSIFIED MIT/LCS/TR-189 NL

1 of 2
ADA048980


1.0
4.5
5.0
5.6
6.3
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.1
1.8
1.25
1.4
1.6

LABORATORY FOR COMPUTER SCIENCE
(formerly Project MAC)

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

(12)

MIT/LCS/TR-189

FORMAL SPECIFICATIONS FOR PACKET COMMUNICATION SYSTEMS

David J. Ellis

This research was supported by the National Science Foundation under grant DCR75-04060 and by the Advanced Research Projects Agency of the Department of Defense and was monitored by the Office of Naval Research under contract no. N00014-75-C-0661.

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

(9) Doctoral thesis,

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER
(14) MIT/LCS/TR-189

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE (and Subtitle)
(6) Formal Specifications for Packet Communication Systems.

5. TYPE OF REPORT & PERIOD COVERED
PH.D.Thesis, Oct. 26, 1977

6. PERFORMING ORG. REPORT NUMBER
MIT/LCS/TR-189

7. AUTHOR(s)
(10) David J./ Ellis

8. CONTRACT OR GRANT NUMBER(s)
(15) N00014-75-C-0661,
NSF-DCR75-04060

9. PERFORMING ORGANIZATION NAME AND ADDRESS
MIT/Laboratory for Computer Science
545 Technology Square
Cambridge, Ma 02139

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Office of Naval Research/Associate Prog. Director
Department of the Navy/Off. Computing Activities
Information Systems Program/National Sci.Foundation
Arlington, Va 22217/Washington, D.C. 20550

12. REPORT DATE
(11) Nov[redacted] 1977

13. NUMBER OF PAGES
140 (12) 141 p.

14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)

15. SECURITY CLASS. (of this report)
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Formal specifications
Verification
Packet communication architecture
Systems description
System correctness
Asynchronous systems
Nondeterminacy

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

One of the most difficult tasks facing computer scientists is that of designing systems and making sure that they perform their intended functions correctly. As computer systems have grown in size and complexity, the problems of system design and verification have become increasingly acute. Formal specifications, which are precise desciptions of a systems function, provide a basis for understanding system operation as well as for proving correctness. Although there has been much work in formal specification and verification of

409 648 [illegible]

20.
computer programs, relatively little research has been done on system specification. A particular class of asynchronous systems, known as packet communication systems, has been chosen as the subject of this study. Packet communication systems are composed of independently operating units that interact only by transmitting packets of information. These systems possess a number of desirable structuring properties that make them suitable for formal analysis.

We have developed a model for formally describing the behavior of packet systems and for proving correctness. The model is based on the fact that packet systems may be viewed both externally in terms of their interaction with the outside world and internally, in terms of their structural composition from smaller units. A packet system is shown to be correct by proving that its formal characterizations corresponding to these two views are equivalent.

Our model is used to prove the correctness of three sample packet systems and a general characterization of acyclic systems is stated and proved.

MIT/LCS/TR-189

FORMAL SPECIFICATIONS FOR PACKET COMMUNICATION SYSTEMS

by

DAVID J ELLIS

November, 1977

This research was supported by the National Science Foundation under grant DCR75-04060 and by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract no. N00014-75-C-0661.

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

LABORATORY FOR COMPUTER SCIENCE
(formerly Project MAC)

CAMBRIDGE MASSACHUSETTS

FORMAL SPECIFICATIONS FOR PACKET COMMUNICATION SYSTEMS

by

DAVID J ELLIS

Submitted to the Department of Electrical Engineering and Computer Science on October 26, 1977 in partial fulfillment of the requirements for the Degree of Doctor of Philosophy.

Abstract

One of the most difficult tasks facing computer scientists is that of designing systems and making sure that they perform their intended functions correctly. As computer systems have grown in size and complexity, the problems of system design and verification have become increasingly acute. Formal specifications, which are precise descriptions of a system's function, provide a basis for understanding system operation as well as for proving correctness.

Although there has been much work in formal specification and verification of computer *programs*, relatively little research has been done on *system* specification. A particular class of asynchronous systems, known as *packet communication systems*, has been chosen as the subject of this study. Packet communication systems are composed of independently operating units that interact only by transmitting *packets* of information. These systems possess a number of desirable structuring properties that make them suitable for formal analysis.

We have developed a model for formally describing the behavior of packet systems and for proving correctness. The model is based on the fact that packet systems may be viewed both *externally*, in terms of their interaction with the outside world, and *internally*, in terms of their structural composition from smaller units. A packet system is shown to be correct by proving that its formal characterizations corresponding to these two views are equivalent.

Our model is used to prove the correctness of three sample packet systems, and a general characterization of acyclic systems is stated and proved.

Thesis Supervisor: Jack B. Dennis

Title: Professor of Computer Science and Engineering

Acknowledgments

The hardest part of this research was the interminably long time spent grappling with vague, abstruse notions, trying to find appropriate underlying concepts. During this period, I needed -- and received -- assistance from many people around me.

Professor Jack Dennis, my research supervisor, showed a remarkable technical insight in working with me on developing new approaches for new areas. I am grateful that he matched my stubbornness by insisting I investigate certain ideas that seemed barren at first but later bore fruit.

Professors Liba Svobodova and Vaughan Pratt, my readers, have both been tremendously helpful in evaluating my ideas and approaches. They shared generous amounts of time with me and made sure I kept my ideas in the proper perspective.

All three of my committee members made substantial contributions to the form and content of this document, pointing out obscurities, errors and misconceptions to be corrected.

I wish to thank my fellow graduate students for listening patiently to many versions of my presentations of my ideas, progressing from the unclear to the coherent. Thanks to Bill Ackerman, Valdis Berzins, Andy Boughton, Dean Brock, Clement Leung, Glen Miranker, Craig Schaffert, Ken Weng and a number of other students for helping me clarify my ideas and come up with the right definitions and proofs.

The MIT Laboratory for Computer Science has been a stimulating environment to work in. Thanks are due for their generous financial support throughout the research as well as for the use of their computer facilities in preparing this document.

My parents gave me constant advice, support and encouragement, standing by me when I needed it and stimulating my confidence in myself through their confidence in me.

Finally, I wish to express my appreciation and gratitude to Toby, my fiancee and wife to be, for patiently bearing with me for two and a half long and difficult years, giving me emotional support and companionship, and believing in me.

Table of Contents

Abstract 2

Acknowledgments 3

Contents 4

Chapter 1: Introduction 5
- 1.1. System design and specification 5
- 1.2. Packet communication architecture and its background 9
- 1.3. Formal specifications 17
- 1.4. The approach to be presented 21

Chapter 2: Packet Systems and their Structure 26
- 2.1. Overview 26
- 2.2. A closer look at packet systems 29
- 2.3. Correctness 34
- 2.4. Structural descriptions 38

Chapter 3: Specifications for Packet Modules 41
- 3.1. The slice relation approach 41
- 3.2. *Streams and their operations* 44
- 3.3. Examples 48
- 3.4. Evaluation 54

Chapter 4: Specifications for Packet Systems 56
- 4.1. Internal specifications 56
- 4.2. Execution sequences (introductory) 60
- 4.3. Properties of execution sequences 68
- 4.4. Execution sequences (formally) 78
- 4.5. Characterization of internal specifications 84

Chapter 5: Proving Packet Systems Correct 86
- 5.1. Setting up a correctness proof 86
- 5.2. Proof for the system ID 89
- 5.3. Correctness of a cyclic system 94
- 5.4. Proof for a nondeterminate system 100
- 5.5. Proving correctness of more complex packet systems 109

Chapter 6: Conclusions 120
- 6.1. Review of the research 120
- 6.2. Future work 124
- 6.3. Parting shots 129

Bibliography 131

Glossary 136

Biographical Note 138

CHAPTER 1: INTRODUCTION

1.1. System design and specification

The fields of computer hardware and software both deal with the same fundamental goal: building systems to perform designated functions. A hardware system is constructed from physical components, while a software system is realized by writing programs in a language implemented on some computer. As both hardware and software systems have grown in size and capability over the years, their structure and operation have grown tremendously in complexity. This has made the task of designing systems increasingly difficult, especially so for large, high-performance systems. It is important that both system designers and users have confidence that their systems perform their functions as intended. System testing, debugging and modification constitute a significant fraction of the time and expense involved in designing systems. The issues of making certain that a system being designed will operate correctly are thus of particular importance to both hardware and software system designers.

Verifying the logical correctness of system designs has been accomplished in practice mostly by "seat of the pants" techniques. The drawbacks of such an informal approach are clear: one can be intuitively certain that a system design is correct, but this is far from a guarantee of correctness. There are numerous "horror stories" about systems that had to be redesigned or scrapped because their designs had serious conceptual errors that went undetected in the verification process. Such errors indicate a lack of understanding on the part of the designers as to exactly what functions the systems are supposed to perform. In order to have a sound understanding of the way a system operates, and in order to be sure that it behaves correctly, it is necessary to make use of precise descriptions of a system's logical function. It is for just this reason that the discipline of *formal specifications* has arisen. Specifications are descriptions of the behavior desired of a system, and a system is shown to be correct by verifying that it satisfies its specifications, *i.e.* operates as it is intended to behave. There are two significant benefits that may be realized by using formal specifications. First, it becomes possible to develop formal verification methodologies, which makes it feasible to *prove* that systems are correctly designed to perform their intended tasks. Second, formal descriptions provide a model through which complex systems can be better understood. Thus, the task of system design may be facilitated through the study of formal system specification and verification techniques.

Formal specification techniques cannot be used blindly without considering the nature of the systems being described. For large, complex systems, the specifications may become so complicated as to make correctness proofs intractably difficult. However, this problem can be alleviated by treating only those systems that satisfy "nice" properties. By insisting on appropriate system constraints that must be satisfied, one can identify classes of systems that have more orderly and structured design with no real sacrifice in functional capability. Through judicious use of this concept of *structured system design*, the system designer can be assured of working with systems that can be more easily understood, described and verified.

Formal specifications have been the center of much research activity within the software field [Rustin, 1972; Liskov and Berzins, 1976]. In addition, an entire discipline known as *structured programming* has arisen to study ideas of structured system design and their ramifications on the programming process [Dijkstra, 1972; Wortman, 1977]. However, there has been relatively little research in corresponding areas of the hardware field. One might explain this difference by saying that there is a much greater concentration of theoreticians specializing in software than hardware, but there is a more crucial underlying reason. Design costs for hardware systems have long been overshadowed by the costs of materials, fabrication and assembly. Once a machine is into production, the design process is ended; all further costs lie in replication and maintenance. For these economic reasons, the physical construction of systems has been the dominant factor in hardware development. With software, on the other hand, design costs have always predominated, since everything is realized on paper. Moreover, software systems are designed for specific applications; if the problems to be handled are changed, then the programs must often be rewritten or redesigned. Hardware systems are general-purpose in that for a change in application it is the program and not the machine that is modified. Software is thus far more transient than hardware, which makes design costs even more important for software. It is therefore no wonder that the initiatives for studying design and specification methodologies have been strongest in the software field.

The rapid developments in semiconductor technology over the past few years are beginning to alter the economic balance in hardware development. Integrated circuit chips can be mass-produced at extremely low cost. Construction costs for hardware systems are dropping dramatically as new fabrication techniques are coming into practice. Since design costs are remaining essentially the same, they are becoming more and more significant in relation to system development. This means that system design techniques and approaches will soon be on the cutting edge of hardware technology. For large and complex systems, whose logical functions are especially difficult to comprehend and work with, the approaches to system design are even more crucial. It is therefore important to open up a thorough investigation of formal specification and structured system design methodologies for hardware systems. And since much of the initiative in this area has come from software research, it is natural to look for ways to apply new technologies used in software design to the hardware field.

A particular class of systems called *packet communication systems*, which are described in the next section, has been chosen as the domain for the research presented here. Packet communication systems are based on a set of structural properties which provide for the building of large, high-performance systems and which also support the development of a theoretical framework for formal specification and verification. In this thesis, we shall develop techniques for formal specification of the behavior of packet communication systems. We shall also take a look at how the formal specifications may be applied towards verifying the logical correctness of these systems. Because there has been so little formal study of methodologies for hardware system design, specification and verification, the research here may be considered as the first step in a new direction.

1.2. Packet communication architecture and its background

Packet communication architecture is a set of principles according to which systems may be designed and structured. The systems satisfying these principles are collectively known as *packet communication systems*; for brevity, they shall also be called *packet systems*. As introduced by Dennis in [Dennis, 1975b], packet systems are essentially interconnections of independently functioning units that interact only by sending each other *packets* of information. The information contained in a packet may have arbitrarily complex structure.

In this research, we have taken a particular point of view, regarding packet systems as being physically composed from hardware units. Some of the important concepts underlying packet communication architecture are particularly advantageous when applied to the design and implementation of hardware systems. It is equally valid, though, to implement packet systems in software. There are no existing techniques for formally specifying or verifying packet systems viewed from the software standpoint, so our work here may also be seen as an advance in the study of software specification as well.

There are two particular notions from the study of structured programming that are directly supported by the principles of packet communication architecture: *modularity* and *hierarchy*. These notions play a large role in the suitability of applying formal specification techniques to packet systems.

Modularity is an approach for structuring programs or systems by composing them from smaller units called *modules.* The basic idea is that the use of a module is separated from the internal details of its implementation. In this way, a module can be developed and changed without affecting other modules. The concepts of modularity are discussed in more detail in [Myers, 1975; Yourdon, 1975; Dennis, 1972b]. An example of a mechanism for supporting modularity in software systems is the notion of *data abstraction* [Liskov and Zilles, 1974]. A desirable design goal for modular systems is that individual modules be as self-contained and independent as possible. This goal can be realized by making interaction among different modules as simply structured as possible through clean, well-defined *interfaces.* Although the advantages of modularly structured systems are clear, the issues of deciding where to draw module boundaries present an open problem. We shall not investigate this problem here.

The notion of hierarchy relates to how systems may be viewed and described. A hierarchically structured system is one that may be stratified into different levels of conceptual detail. Each level makes use of mechanisms whose internal details are hidden away in lower levels. Each mechanism within the system is used at higher, more abstract levels than where it is defined. In this way, low-level detail is isolated so that it will not interfere with higher-level conceptual views of the system. The basic principles and concepts of hierarchy in systems have been presented by Parnas [Parnas, 1974; Parnas, 1975].

The properties of modularity and hierarchy make systems in general easier to understand and work with. Each module in a hierarchical and modular system has a set of "neighbor" modules with which it communicates. The behavior of a given module depends on the conventions by which it interacts with its neighbors, but it is completely independent of the internal characteristics of the other modules in the system. Consequently, the designer of a module need not worry about what goes inside any other modules; the only relevant concerns are the internal construction and the interface conventions for the particular module being designed. In this way, design information is partitioned along the boundaries of the modules, insulating the system designer from irrelevant detail. This insulation is further enhanced in hierarchical system structures. Each level of abstraction in the hierarchy is isolated from the other levels. The designer of a module has to know the external behavioral characteristics of the submodules from which the module is composed, but the internal structures of the submodules should be totally irrelevant to the design of the given module. Thus, systems that are both modular and hierarchical have two dimensions along which design details are partitioned. When the structure of a system prevents certain design information from affecting areas it does not concern, the system design is simpler to understand. Conceptual simplicity is an important design goal whenever system specification and verification are to be taken into account.

Although the concepts of modularity and hierarchy have been given far less theoretical attention in relation to hardware than software, they are almost universally regarded as fundamental to good hardware design practice. Hardware systems have for a long time been built up from modules such as adders, clocks and shift registers, and now there is an even greater variety of off-the-shelf component chips to use as modules. At a higher level of abstraction, a typical microcomputer is composed of a microprocessor, some RAM storage, I/O drivers and interface elements. Each of these components can be treated as a module, and these modules can themselves be decomposed. For example, the processor has as submodules an adder, various registers, gating logic, an instruction decoder and other components; these submodules can in turn be further decomposed. This example shows how digital system design exhibits hierarchical and modular properties. In general, these properties are realized by intelligent system design, but they are difficult to achieve when designing large computing systems. Features such as virtual memory, multi-user environments, parallel programming and the sharing of data among different processes are difficult to realize; they are usually implemented in practice either by simulating them in software or by adding new components as afterthoughts to a basic Von Neumann machine. The interactions among these added components are anything but modular in nature, which is one of the reasons why large computing systems are so difficult to build. Packet communication architecture, as we shall see, provides direct support for hierarchy and modularity.

Packet systems are both modular and hierarchical in structure. The modules in a packet system are simply the independently operating units that comprise it. Packet systems can easily be structured so that their modules correspond to the conceptual units in the designer's view of the system. Further, the principles of packet communication architecture allow the modules that form a packet system to be viewed individually as systems that may themselves be decomposed into interconnected component modules. This hierarchical property of packet systems provides some of the major conceptual foundations of the approach to specification and verification that will be developed. By making hierarchy and modularity explicit, packet communication architecture not only facilitates formal specification and verification, but in addition serves to encourage good system design practice.

One of the most important design goals for packet systems is that the modules within a system operate as independently as possible. In support of this goal, it is required that modules communicate with each other by passing packets *asynchronously.* This principle eliminates the need for a centralized control facility to coordinate the action of all the modules, which greatly simplifies system structure. Moreover, it provides for concurrent operation of the modules, leading to enhanced system performance. A module, while awaiting response from other modules in order to perform certain tasks, can busy itself with other tasks for which the required responses have already arrived. An operation may proceed as soon as the information it needs is received, as opposed to what happens with conventional architectures, in which operations cannot be performed until they are explicitly initiated by the sequential control. It is this distinction that provides for concurrency and thus allows packet systems to make more effective use of the available resources than do conventional large systems.

The microcomputer example given above exhibits a number of hierarchical levels of abstraction. It may be noted that the interfaces between modules at different levels of the hierarchy have completely different characteristics. At the top level, one deals with transmission of applications data; within the microprocessor, microinstructions are passed; and at a still lower level, it is basic logical signals that are passed and gated. In digital systems as they are currently designed, interface protocols depend on the speed at which the various modules process control and data signals. This dependence limits the degree of modularity that can be achieved in existing systems, since a module's interface with its outside world is not free of internal speed and timing considerations.

Packet systems are not subject to such limitations; one of their important properties is that the timing characteristics of an individual module in a packet system do not affect the operation of any other module. A module in a packet system can be replaced with another unit that performs the same task orders of magnitude faster or slower than the original module, and this change will not alter the logical functioning of the system. Packet systems are thus speed independent, which removes from the designer the burden of having to take into account the speed and timing properties of system components in order to assure logical correctness. Speed independence enhances the degree of modularity in a system and thus provides an additional element of structuring in systems, which further assists system design and verification. It should be noted that a system must operate asynchronously in order to achieve the goal of speed independence. Packet systems, since they are speed independent, can accommodate a uniform protocol for communication of packets among their component modules. This uniformity of interface provides the basis for the method of system specification that will be described here.

The idea of building systems by connecting independent modules under an asynchronous and speed-independent discipline is not new. An early exposition was given by Muller [Muller, 1963]. There was a major research effort several years later directed towards realizing systems that were to be physically constructed from hardware units called *macromodules* [Ornstein, 1967]. Patil has investigated logical designs for modules with which asynchronous systems may be built [Dennis and Patil, 1971], and more recently he has been working with applying programmable logic arrays to this task [Patil, 1975]. All of these designs differ from packet communication architecture in that control signals and data values are passed through the systems separately, traveling on two distinct sets of communication pathways. In packet systems, the notions of control and data are unified, eliminating the need for separate pathways. This is yet another respect in which the principles of packet communication architecture serve to simplify system structure.

Since packet systems operate concurrently, a significant area of application for packet communication architecture lies in realizing computer systems that provide direct support for parallel programming. If different parts of a program can be executed in parallel, then it is advantageous to run the program on a machine for which the hardware can overlap their execution. In this way, one can optimize running speed and utilization of resources such as memory, processing elements and peripheral devices. The study of *data flow* computation has precisely this goal in mind. Data flow is the representation of programs in such a way as to make the data dependencies and inherent parallelism explicit. Given any two operations O_1

and O_2 in a data flow program, it should be immediately apparent from the program structure whether O_1 should be performed before O_2, whether O_1 needs results of O_2 in order to be performed, or whether O_1 and O_2 are independent (can be done in parallel). Data flow programming has been treated extensively in the literature; for both exposition and references, see [Dennis, 1975a; Weng, 1975]. Substantial effort has gone into studying designs for machines that can directly and efficiently execute data flow programs [Rumbaugh, 1975; Dennis, 1974; Dennis, 1977; Arvind, 1975; Plas, 1976]. On such a machine, there is no sequencing of instructions; an instruction may be executed any time after its operands become available. This is essentially the same principle as the one underlying the operation of modules within a packet system; in fact, the concepts of packet systems have been directly influenced by the research in developing architectures to implement data flow.

The conceptual compatibility between the ideas of data flow and packet communication architecture yields a natural connection between them. In a packet system, the activity that takes place within a module is initiated by the arrival of the appropriate data packets. There is no explicit sequencing of operations in data flow programs, and it should be practical to implement them on systems that do not require ordered sequences of instructions as their programs. This is one of the motivating factors behind the conception of packet communication architecture. Most of its concepts are far from new or original, but it is the combination that makes it suitable for realizing data flow computation in hardware. Conversely, data flow is a natural way to represent programs that will run on processors designed according to the principles of packet communication architecture. Thus, there is a commonality between data flow and packet systems that arises because they share similar goals and principles.

There is one more property of packet systems that should be noted here. The behavior of a packet system (or of any of its modules) is observable in terms of the packets it sends out in response to the packets it receives. In general, packet systems are *nondeterminate*, which means that given the packets received by a module, there may be several distinct but equally valid responses to the input. Nondeterminacy is one of the factors that make the behavior of packet systems difficult to understand and formalize. This will have a definite bearing on the approach taken here towards specification and verification.

This concludes the overview of the basic ideas of packet communication architecture. The principal reason why packet systems were chosen for this research is that their design is structured in a way that supports system specification and verification. The next section presents an overview of some of the major concepts and techniques that have been developed for formal specification of computer programs and systems.

1.3. Formal specifications

Much of the research concerned with formally describing the activity within computer systems has dealt with programming language specifications. There are essentially three basic approaches to describing the behavior specified by a piece of program text: *axiomatic*, *denotational* and *operational*. Each approach may be applied to verifying the correctness of program text as well as serving as a pure descriptive vehicle.

Axiomatic specifications capture the effect of executing a program by comparing properties of the system state before and after execution. The paradigm "if assertion *A* is true before program text *P* is executed, then assertion *B* is true after *P* is executed" describes the meaning of program text *P*. Special rules of inference are set up to describe the meanings of various combinations of program texts in terms of their components' meanings; these rules incorporate the basic semantic properties of constructs such as iteration and conditionals. This approach became known through the work of Floyd [Floyd, 1967] and Hoare [Hoare, 1969] in which it was used to prove correctness of simple flowchart-like programs that manipulated integers. The assertions they used related values of program variables. There has been a substantial amount of more recent research in axiomatic specifications. Dijkstra [Dijkstra, 1976] has built up an entire methodology of programming around the ideas of axiomatic specification. Owicki and Gries [Owicki, 1976] extended Hoare's techniques to parallel programs; their assertions made use of auxiliary state variables to keep track of interprocess coordination. Greif [Greif, 1975] took a different approach to parallel programs, using a partial time-ordering on events to express coordination properties.

Denotational specifications capture the effect of a program by viewing the objects they model as abstract mathematical entities. This approach provides a formal mathematical description of the computational notions being treated. An early denotational approach to specifications for programming languages was the application of a mathematical formalism known as lambda calculus towards describing the semantics of Algol 60 programs [Landin, 1965]. The best known work in denotational specifications has followed from the research of Scott and Strachey [Scott and Strachey, 1971]. Mathematical results from lattice theory are used in the construction of complex domains over which programs are represented as functions. Programs are proved equivalent by showing that their functions coincide. A tutorial presentation of the Scott-Strachey approach is given in [Tennent, 1976].

Operational specifications deal with the changing states within computer systems as computations are performed. This is done by means of a state-transition model in which a state represents information present in the system at a given moment in time. The action of a program is captured by the sequence of transitions of the model. The sequence of states the model passes through as a program is executed defines the action of an *interpreter* for the program. The idea of using such an interpreter to define the meaning of programs in some language originated with McCarthy [McCarthy, 1962]. A well-known approach to operational specifications is the Vienna Definition Language (VDL) described in [Wegner, 1972b], which uses an interpreter that manipulates tree-structured system states. Dennis' Common Base Language [Dennis, 1971] is similar, dealing with more general directed graphs in place of trees. Another approach to operational specifications is due to Parnas [Parnas, 1972]. This approach distinguishes two kinds of operations: those that yield state information, and those that alter the state of the system. Parnas applied his approach to operations on abstract data in programming languages; this was extended to the domain of systems in [Robinson, 1975]. Verification is achieved within an operational framework by proving that the behavior of the interpreter in question is equivalent to the behavior of one that is known to perform the desired function. The ideas underlying verification by interpreter equivalence were developed by Milner [Milner, 1971] and are also presented in [Wegner, 1972a].

Although hardware specification has not received as much attention as software specification, there has been a substantial amount of study of computer hardware description languages (CHDL's). The approaches taken towards hardware specification have been almost entirely operational. The language APL, before it was ever implemented as a programming language, was used as a hardware description language to *specify* the operation of IBM/360 computers [Falkoff, 1964]. Another CHDL, called ISP, was developed by Bell and Newell [Bell and Newell, 1971] to describe the operation of a large number of different computers. Both of these CHDL's describe their target systems at the instruction set level, treating machine words as a basic data type with operations for byte extraction and bitwise arithmetic and logical functions. On the other hand, the language PMS, which was also developed by Bell and Newell [Bell and Newell, 1971], describes the structure of computer systems in terms of their component processors, memories, controllers and I/O devices. This is an example of a CHDL describing systems from a higher-level conceptual point of view. DDL [Dietmeyer, 1974] is an example of a lower-level CHDL that defines the behavior of elements such as multipliers by *specifying* them as interconnections of basic logic gates.

Most of the CHDL's have been developed with two particular goals in mind: automated system design, and system testing by means of simulation. However, the microprogram certification project at IBM has developed an approach to hardware system specification that is directed towards formal verification of system design [Birman, 1974]. For both the instruction execution level and the microprogram level, a VDL-style interpreter is used to supply formal specifications. These two interpreters are then proved equivalent in exactly the same way that correctness is proved in operational specifications for programming languages as described above. The proof techniques for this approach are additionally described in [Leeman, 1975; Leeman, 1977]. Rumbaugh takes a similar approach to the IBM group in proving the correctness of a data flow processor [Rumbaugh, 1975]. He shows that an interpreter for his machine is equivalent to one that models the operations in a data flow language.

1.4. The approach to be presented

The research in specifications that has been reviewed here cannot be directly applied to the task of formally describing and verifying packet systems. The principal reason for this is that conventional techniques are not equipped to handle the asynchronous operation of packet systems. The concurrency in packet systems makes it difficult to verify their correctness: in order to establish some property of a packet system, it must be shown true *for all possible* sequencings of packet transmissions and receptions within the system. Most existing techniques for formal specifications do not lend themselves to this kind of task. Moreover, the notion of sequencing of actions, which is fundamental to nearly all the approaches that have been taken towards formal specifications, is not present in the context of packet systems.

There is a descriptive formalism, *Petri nets*, that has been developed specifically for specifying asynchronous behavior within systems. Petri nets [Peterson, 1977] are directed graphs in which markers called tokens pass along the arcs and through the vertices to model the occurrence of various events. Although they have received much attention in this regard [Patil, 1970; Hack, 1976], they cannot be directly applied to verifying packet systems. Petri nets convey only control information for use in coordinating concurrent activities; the nature of these activities is left uninterpreted. In particular, they do not treat data values that are passed within packet systems. Also, although many mathematical properties have been established for Petri nets, no methodology has been developed for applying them to system verification. Most of their practical applications have been in connection with simulating asynchronous behavior rather than proving properties of systems. For these reasons, Petri nets do not seem to meet the goals of specification and verification of packet systems.

Within a packet system, the modules receive and process input packets, generate new packets for output and send them out, all asynchronously and in parallel. The kind of approach that seems most suited to specifying this kind of behavior is basically operational in nature. The state of a packet system describes which packets have been passed between which modules (and may also convey any coordination information relevant to the correct operation of the system). However, unlike conventional operational models, the transitions between states need to be governed not by an externally supplied sequence of instructions to be processed by the system, but rather by the presence or absence of packets as needed for processing. This means that an operational model for a packet system must take into account the many possible sequences of execution that could arise from the flow of packets.

Describing the internal operation of packet systems is not sufficient by itself for verification purposes. There must also be a method for specifying the logical function a system is expected to perform. This function concerns the system's input/output behavior as seen by the outside world in terms of packets received and sent out. Of the three kinds of approaches to specifications as discussed in the previous section, a denotational approach seems best suited for our needs because it can be easily tailored to describe sequences of packets that have been passed between various modules. Because of this flexibility, a denotational approach will also interface nicely with the hierarchical structuring of packet systems. Thus, we shall be working with two kinds of specifications for packet systems: operational specifications to describe the internal operation, and denotational specifications to describe their behavior in relation to the outside world. Verification of correctness for a packet system will be demonstrated by proving that these two sets of specifications for the system agree with each other.

A recent research effort is specifically directed towards formally describing the structure and behavior of packet communication systems. The descriptions are expressed in a formalism called ADL (Architecture Description Language), which is introduced in [Leung, 1977]. There are two ways in which a system may be described in ADL: structurally and behaviorally. A structural description characterizes how the system is formed as an interconnection of modules. A behavioral description is an operational characterization of the system's interaction with the outside world, describing reception, processing and transmission of individual packets. The notation used here is similar to the programming language Pascal [Wirth, 1971], and the underlying semantics are also based on the principles of data flow. As a first approach to placing packet systems on a formal foundation, ADL is both helpful and illuminating. However, the concept of specifying the internal operation of a packet system has not been developed within the ADL framework. This idea, which has not been studied previously, is crucial for verifying the correctness of systems in a hierarchical and modular fashion. The development of this concept is the most significant contribution of our research. The denotational approach to be used in our treatment for specifying the function of systems turns out to be more convenient to use than the operational approach found in ADL.

The body of this thesis consists of four chapters. Chapter 2 describes the basic properties of packet systems in more detail. The notion of correctness is defined, and a formalism for describing the structural composition of packet systems is also presented. Chapter 3 presents the denotational part of the packet system specifications. The behavior of a packet system or module is formally defined as a relation between the packets it receives as input and the corresponding packets sent out in response.

Chapter 4 motivates and defines the central concepts of the research, giving an operational characterization of the actions that take place within a packet system. Chapter 5 shows how the specification model developed in the two preceding chapters may be applied to the task of verifying correctness of packet systems. Three sample systems are proven correct, and a theorem is presented to show how the model may be simplified in certain cases.

CHAPTER 2: PACKET SYSTEMS AND THEIR STRUCTURE

2.1. Overview

In this chapter the concepts of packet communication architecture will be elucidated in detail. We shall clarify the notion of a packet system and develop a means for formally describing the structural composition of such a system. We will also informally introduce the concept of correctness for packet systems. The machinery needed to formally define and prove correctness will be developed in Chapters 3 and 4.

Packet communication architecture is a discipline dealing with a special class of systems known as *packet systems*. Packet systems are composed of independently functioning units, known as *modules*, which interact only by passing information to each other. The information is passed in the form of units called *packets*. There is no centralized facility for coordinating the action of the modules. Data processing and communication within packet systems are asynchronous, and the various modules operate concurrently.

In a packet system, the various modules are interconnected through one-way data paths known as *channels*. A channel connects two modules in a specified direction and is used to pass data from the first module to the second. Channels leading into a module are called *input channels* for the module, and channels leading out are called *output channels*. A packet system has its own set of input and output channels connecting it to the outside world. The other ends of these channels are never explicitly designated.

The structure of a packet system is determined by the way it is composed from modules and channels, and always remains fixed for a particular system. Modules and channels within a system are uniquely named. Figure 2.1-1 depicts a packet system DAS composed from three modules D, A and S. There is one system input channel X and two system output channels Y and Z. The internal channel U connects module D to module A, and channel V connects module D to module S.

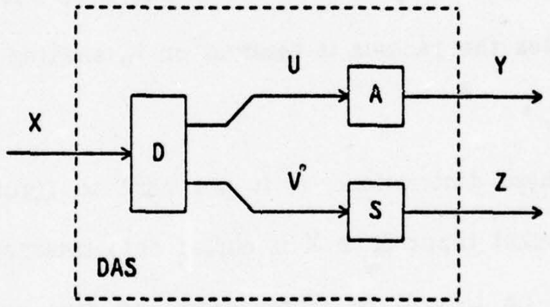


Figure 2.1-1: A sample packet system DAS.

All data treated by a packet system appear in the form of packets, which are passed along the various channels of the system. Each packet carries a value of some type. The modules in a packet system all have the same basic principle of operation: a module receives packets on its input channels, processes them internally and generates packets to be sent out on its output channels. This principle applies to entire packet systems just as it does to their individual component modules. Packet systems are data-driven in the sense that the progress of a computation in a packet system is determined by the passage of packets through the system.

There are two ingredients which together determine the behavior of a packet system: its structure and the behavior of its modules. Thus, for instance, in order to describe how the system DAS acts, one must first decide what the modules D, A and S do. We now describe the behavior of these three modules.

All three modules receive and pass integer-valued packets. Module A, upon receiving a packet from its input channel U, adds one to the value and sends out the incremented value as a packet on its output channel Y. Module S behaves identically except for subtracting one instead of adding. Module D duplicates the packets it receives on X, sending out identical copies on U and V.

Given these descriptions, it is not hard to figure out how system DAS acts. Any packet input from X is copied onto internal channels U and V. The packet passed on U will be incremented and sent out on Y; the packet passed on V will be decremented and sent out on Z. Thus each packet received by DAS causes two packets to be generated: a packet with value one greater on Y and a packet with value one less on Z.

It may occur to some readers here that these characterizations are incomplete. There is ambiguity in describing what happens when several packets are to be processed in sequence: in what order are resulting packets generated and passed? In our example we can resolve such questions by stipulating that the relative order of packets on a channel is always preserved. Precise methods for dealing with questions of this nature will be described in the next chapter.

2.2. A closer look at packet systems

In this section the workings of packet systems will be examined in greater detail. The first thing we discuss is one of the fundamental properties they satisfy: *the internal resources of a packet module or packet system may be allocated and utilized in any arbitrary manner as long as the specified operations will be performed correctly.* Consider, for example, the system DAS from the previous section when it is in a state depicted in figure 2.2-1. An input packet with value 2 has been received on the X channel and processed by the D module, leaving copies of the packet on channels U and V. Another packet with value 5 is still waiting on channel X to be processed by the system.

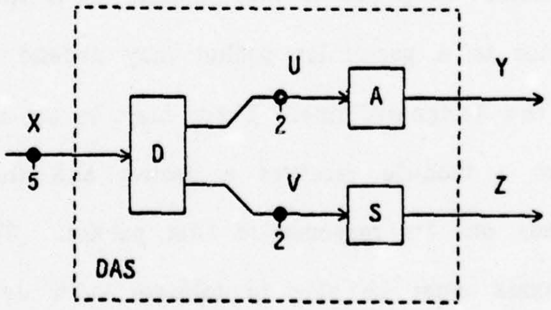


Figure 2.2-1: A sample state of system DAS.

There are three actions that should now be performed within the system: *(1)* module A absorbing and processing the packet on channel U; *(2)* module S processing the packet on V; and *(3)* system DAS accepting the packet from channel X and initiating its processing in module D. The crucial property of packet systems exhibited here is that these three actions may be performed in any order, serially or concurrently, and the correct operation of system DAS will be completely independent of whatever particular order is chosen. It is this property that makes the behavior of packet systems genuinely asynchronous.

We can gain a better understanding of the action of packet systems by taking a more detailed view of the operation of their component modules. When a module receives a packet from one of its input channels, it begins to process the packet internally. Sometimes the only effect of the packet's absorption is that the module's internal state may change. In general, though, the module's semantics may require that it generate one or more packets to be sent out on its output channels in reply to the packet received. The sequences of packets generated by a module in reply to a packet received are said to be the module's *response* to that packet. It is important to note that a module's response to a particular packet may depend on previous packets input as well as the designated one. There may be an arbitrary finite delay between the time a module receives a packet and the time the module generates and sends out its response to that packet. The fact that packet modules and systems must be able to tolerate such delays is an essential consequence of their asynchronous operation.

There is a special protocol that must be fulfilled in packet systems for the transmission and receipt of packets through the various modules and channels. Suppose a channel C connects module M1 to module M2, as illustrated here:

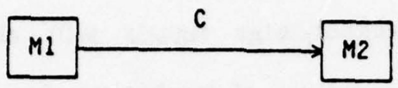


Figure 2.2-2: A channel in a packet system.

It is desirable for module M1 to have some way of knowing when it has successfully sent a packet out on channel C. The convention that has been adopted is that when a packet sent on C from M1 is received by module M2, M2 will send a signal to M1 on channel C in the *reverse* direction to indicate that it now has the packet safely in hand. Such a signal is known as an *acknowledge signal*. It is not until M1 receives an acknowledge signal for a particular packet that it knows it is done with the process of generating and sending that packet. Thus, from the point of view of module M1, there are three discrete steps in the transmission of a packet: generation, sending and receipt of acknowledgment. It should be noted that module M2 cannot generate output to packets it receives from channel C until it has sent back on C an acknowledge signal for those packets. There is a caveat with regard to acknowledge signals: although they are sent in response to every packet transmission in a packet system, we regard them as part of the hardware and not available to be manipulated by system designers.

The channels in a packet system are assumed to have certain special characteristics as transmission media. The first, and simplest, is that any time a packet is sent out on a channel, it will eventually be received at the other end. A packet generated to be sent out from some module in a packet system can never be called back. This means that whenever a module generates a packet to be sent out, it will receive an acknowledge signal for the packet within some finite span of time. It is assumed that the channels never "break" and that acknowledge signals will always be received by the appropriate modules. Failure of mechanism, for the purposes of verification, invalidates the entire system function. The issues of fault tolerance in systems are beyond the scope of this research. Thus, packet communication architecture requires that every packet generated by some module must actually be sent out and acknowledged within some finite time interval. It should be noted that this requirement is a consideration of correctness rather than performance, because packet systems are speed-independent.

A second important property of channels is that if a module receives a packet from one of its input channels, then that packet must have been sent out on that channel at some previous time. Thus, for example, module M2 may not receive a packet from channel C unless module M1 had already sent that packet out on C. Another way of stating this property is that no channel may generate spurious packets of its own.

A third characteristic of channels is that they act as FIFO queues, which means that if the module M1 above sends a packet x out on channel C and then sends another packet y out on C at a later time, then M2 must receive and acknowledge x before y. We make the further assumption that the channels have unbounded buffering capacity, which means that there is no limit to the number of packets that can be on the channel C at any given moment of time and be awaiting receipt by the "target" module M2.

Physically speaking, this assumption is not realizable in general, because no real device can have infinite capacity, let alone a high-speed transmission medium. However, if we assume the unbounded buffering, then we rule out the possibility of system deadlock caused by packets piling up in certain channels and inhibiting further packet output into those channels. Unbounded buffering is therefore a convenient assumption to make.

Finally, we shall assume that for each channel in a packet system there is a designated set (type) of packets that may be passed on the channel. For example, one channel may carry only integer packets, while another channel may accept only packets that consist of an employee name together with a corresponding identification number.

There is an extremely important property of packet systems which we will be treating, namely *nondeterminacy*. A module or system is said to be nondeterminate if its semantics allow two or more distinct possible responses to a given packet input. A simple example of a nondeterminate module is one that models the toss of a coin. It has one input channel and one output channel, and its response to any packet received will be a single packet with either the value "heads" or the value "tails." The choice is arbitrary and independent of the input packet. Nondeterminate modules and systems are very difficult to work with because the multiplicity of possible results is cumbersome to model mathematically. We will explicitly allow for nondeterminate modules and systems in our treatment.

A certain class of nondeterminate system behavior will be of particular interest because it arises frequently in the design of packet systems.

This kind of behavior concerns the relative order of packets sent out on a channel. Consider a system in which the task of generating and sending out packets in response to inputs taken from a specific channel is relatively complicated or time-consuming. One would naturally wish to allow the processing of distinct inputs to proceed concurrently if possible. But then it may turn out that responses to a recent input will be ready to be sent out before responses to inputs received earlier. Moreover, it cannot be determined in advance whether or not such "cutting ahead" behavior will actually occur. It is possible to impose a synchronization discipline that will force the outputs into a desired order, but in doing so all the advantages of asynchronous processing of different inputs are lost. Thus, if the system application and design can tolerate "cutting ahead," it is wise to allow it. In general, then, providing for nondeterminate behavior that involves different alternative orderings of generated output packets should often in practice become an attractive design goal for packet communication architecture.

2.3. Correctness

The notion of correctness for packet systems bears a close relationship to the ways the issues of system structuring and composition are treated within the framework of packet communication architecture. At a very intuitive level, a system is correct if it satisfies certain conditions laid out for it in advance. For packet systems, these conditions take the form of behavioral specifications. As we mentioned in the preceding chapter, a packet system's behavior is observable by the way it responds to its inputs. More precisely, the behavior is a relationship between inputs received and outputs generated in response to those inputs. A packet system, therefore, is correct if this relation satisfies a given set of specifications. The nature of such specifications will be discussed in detail in subsequent sections.

It is important to note that one cannot *prove* correctness of a system without some knowledge of its internal workings. If a system is viewed as a "black box" (figure 2.3-1),

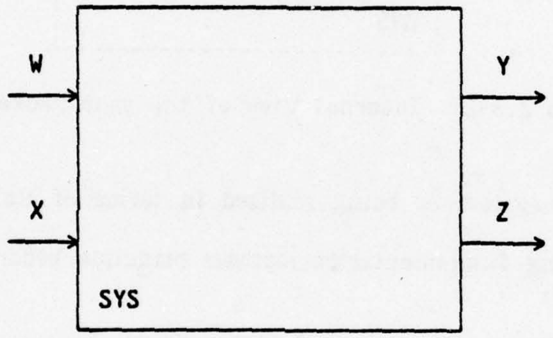


Figure 2.3-1: "Black box" view of a packet system.

then the only things that can be seen are packets entering and leaving. There is simply not enough information available to determine whether or not a system is behaving correctly. Since modules operate asynchronously and with arbitrary finite delays, one cannot tell if additional output packets are forthcoming. For example, suppose a system has already sent out all the packets it should transmit in response to some particular input. The module only *appears* to be behaving correctly, since there is no guarantee that an invalid packet will be unexpectedly sent out later. Even if this were determinable, observation alone could never suffice to decide whether the system would respond correctly in *all* situations. The only way to tie down the notion of correctness for a particular packet system, therefore, is to open the system up and look inside:

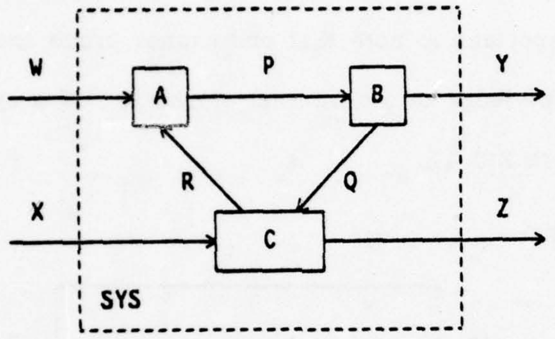


Figure 2.3-2: Internal view of the same packet system.

If we view the system as being realized in terms of its component modules, then the following fundamental correctness principle becomes evident:

> *A packet system is correct if its given structural decomposition satisfies the behavioral specifications for the system whenever the component modules satisfy their own respective behavioral specifications.*

The notion of a system's decomposition satisfying a set of specifications is not yet formally defined; it will be treated in detail in Chapter 4. The notion of a module satisfying specifications is simply that of a physical device acting as intended. The above correctness principle defines only a relative nature of system correctness. An obvious question that arises is how to establish the correctness of the modules in order to show the system correct. We already have the answer to this question: just as with the system itself, correctness of the component modules can be established only in terms of their own respective internal structures.

A significant ramification of this approach is that packet systems and modules are really two different views of the same thing: a module is revealed to be a system when one examines its internal structure, and ignoring the composition of a packet system is just the same as regarding it as a module. There is an underlying source for this conceptual unity, which is that packet communication architecture supports the hierarchical structuring and composition of systems. Packet systems can (and should) be designed so that there are distinct and well-structured levels of decomposition, each level consisting of systems built up from simpler modules. In this sense, our fundamental correctness principle for packet systems supports a top-down verification methodology in which correctness proofs are broken down level by level into their natural logical and conceptual constituents. Logically distinct lines of argument are isolated so that they cannot interfere with one another. Thus the notion of modular and hierarchical system structure is carried through in the approaches we take to correctness and verification.

It may seem for a moment that there is a potential infinite regress in working with smaller and smaller modules within modules, but this can never arise. There is always a well-defined bottom level to the hierarchy in which the modules are regarded as implementing primitive operations such as adding and gating. At this point, correctness has been reduced to the way the primitive functions are defined.

Our approach to correctness and verification of packet systems allows a system to be viewed in two different ways: *internally*, in terms of its structural composition from modules, and *externally*, by concealing the internal workings. The idea of distinguishing between internal and external views of systems is closely related to the notion of data abstractions in programming languages [Liskov and Zilles, 1974]. As we shall see in Chapter 3, it is fairly straightforward to construct behavioral specifications for a packet system viewed externally. However, in order to establish correctness of a system, we need to show that the external characterization agrees with the system's structure. It is a difficult task to formally describe the behavior of a system in terms of its internal composition. We shall address this task in Chapter 4.

2.4. Structural descriptions

The only means we have used so far to describe the structure of packet systems is through informal block diagrams. If any general assertions are to be made involving system composition, we will need a more precise vehicle for structural description. Such a technique is introduced in this section.

The structure of a packet system may be modeled in very straightforward fashion by a directed graph in which nodes representing modules are connected by directed arcs representing channels. Figure 2.4-1 shows a sample packet system together with the directed graph that models it. Note that the directed graph has an extra node labeled "*". This gives explicit representation to the system's "outside world," which serves as both the source of system input channel X and the target of system output channel Y. The graph may look like just another stylized drawing of the system, but

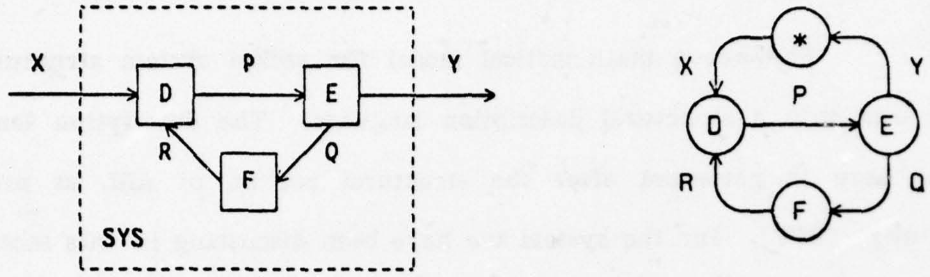


Figure 2.4-1: A packet system and its directed graph.

it is a mathematical object of specific characteristics. Formally speaking, a directed graph is an ordered pair of the form ⟨N, A⟩ in which N is the set of its nodes and A is the set of its arcs. Each arc in A is an ordered triple containing a *source* node, an arc name and a *target* node. An arc $a \in A$ has the form (a.source, a.name, a.target). For example, the graph in figure 2.4-1 is the ordered pair

⟨{*,D,E,F}, {(*,X,D), (D,P,E), (E,Q,F), (F,R,D), (E,Y,*)}⟩.

It is easy to see that for each node n in the directed graph we can define the sets of arcs leading into and out of n. These sets are given by

inputs(n) = {a∈A: a.target = n} and outputs(n) = {a∈A: a.source = n}.

The directed graph characterization thus mathematically specifies how the modules in a system are interconnected.

There are two additional properties of packet systems that can be incorporated into our formal structural descriptions. First, we can model the packet type restrictions for the channels by associating a type description with each channel. Second, we can specify packets initially present on the channels with an initial packet sequence for each channel. Both properties are handled easily in the directed graph model by adding extra fields to the arcs.

The above mathematical model for packet system structure may be sugared into a structural description language. The description language we use here is patterned after the structural portion of ADL as presented in [Leung, 1977]. For the system we have been discussing in this section, if we assume that all channels carry only integer valued packets and that there is one packet with value zero initially present on channel R, then the formal description of its structure may be represented as follows:

```
System SYS
    inputs X(integer)
    outputs Y(integer)
    internals P(integer), Q(integer), R(integer)
Submodules
    D inputs X, R; outputs P
    E inputs P; outputs Q, Y
    F inputs Q; outputs R
Initially R<0>
```

While descriptions of this form do not explicitly name the source and target modules for each channel, these are very easily determined since each internal channel in the system must appear exactly once in a submodule input list and exactly once in a submodule output list.

This section has presented structural specifications for packet systems. The next two chapters present a model for behavioral specifications.

CHAPTER 3: SPECIFICATIONS FOR PACKET MODULES

3.1. The slice relation approach

Because of the way a packet system is built up from component modules, the behavior of a system will be a function of its structure and the behavior of the modules in it. In this chapter we shall develop a method for formally specifying the behavior of packet modules. Specifications defined by this method will be called *external specifications* because they describe the behavior of packet modules without considering their internal structural composition.

A packet module has a fixed number of input channels on which it receives packets to be processed, and there are a fixed number of output channels on which it sends out packets in response to the inputs it has received. A formal behavioral specification for a module must be able to rigorously determine for each input exactly what is a valid output response. Because packet systems are in general nondeterminate, the potential multiplicity of valid output responses rules out a direct functional mapping. Instead, we shall supply external specifications for a module M in the form of a *relation* EXT_M that formally relates inputs to the semantically valid corresponding outputs. Such a relation will be called an *external characteristic relation* for the module M.

The most obvious approach is to use a relation from input packets to output packets, but this does not suffice in even the simplest case: consider a module ID that "does nothing," that is, sends out its input packets untouched.

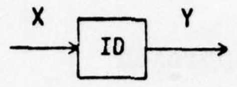


Figure 3.1-1: The identity module ID.

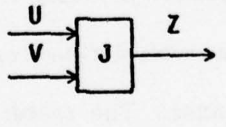
The identity relation EXT_{ID} on packets defined by the equation

$$(p,q) \in EXT_{ID} \text{ if and only if } p = q$$

does not completely describe the behavior of the module ID. If ID receives as input a packet with value 1 followed by a packet with value 2, there are two different possible responses: ID can send out the 1 followed by the 2, or it can send out the 2 first and the 1 later. Thus a specification for the module must describe the sequencing of packets in order to completely capture its behavior. For example, if we intend for the module ID to preserve the relative order of the packets it receives, then its behavior would be correctly specified by the identity relation EXT_{ID} taken over the domain of *sequences* of packets rather than individual packets. Such sequences are required in general to describe the behavior of a module when it depends on a memory of *previous* packets received in order to decide how to respond to a given packet. We therefore need to develop some mathematical machinery for manipulating sequences of packets. We will use the term *stream* to denote a sequence of packets. The mathematics of streams will be discussed in the next section.

In general, the behavior of a module is specified by a binary relation that relates presented inputs to valid output responses. For the module ID, we see that presented input may be correctly modeled by a stream of packets passed on the input channel X. For a module with an arbitrary number of input channels, in order to model presented input we need a separate packet stream for each input channel. We therefore define an *input slice* for a module M to be a collection of streams, one for each input channel of M. Similarly, an *output slice* has as its components one stream for each output channel. Thus the formal specifications for a module M will consist of a binary relation between input slices and output slices. This relation is called the *characteristic relation* for M. We reserve the notation EXT_M from now on to denote the characteristic relation for a module M. The slice relation approach to module specifications is not original, and a corresponding definition may be found in [Dennis, 1972a].

As an example, an input slice for the module J shown below is a pair (u,v) in which u and v are packet streams for channels U and V, respectively; an output slice for J has the form (z), where z is a packet stream over Z.



Thus the characteristic relation EXT_J for J will have elements of the form ((u,v), (z)).

Slices distinguish the time ordering between packets passed on each individual channel but not between packets on different channels. It may seem that crucial behavioral information is lost by not imposing a total ordering on all packet transmissions into and out of a module, but this turns out not to be the case. If a packet p1 is sent out on a channel C1 in some packet system before packet p2 is sent out on channel C2, there is no guarantee that p1 will arrive ahead of p2 in their race to their respective destinations. This is because asynchronous packet systems impose no constraints on transmission times along channels, allowing for different channels with different characteristics suited to their needs. Thus, the extra information obtained from interstream packet ordering is rendered useless by the properties of channels in a packet communication system. The use of slices in our model, then, provides exactly the information needed for proper behavioral specifications.

3.2. Streams and their operations

In this section the basic definitions, operations and mathematical properties of streams are laid out in detail. Because of the technical nature of the material, an index to the notations and technical terms is provided in an Appendix.

For any arbitrary packet module, we take as given for each of its input and output channels a well-defined space (set) of packet values that may be passed along that channel. The space, which we call a *channel space* for the channel, is identified with the channel and shares the same name. Similarly, elements of a channel space are identified with packets passed on the channel.

We will define a stream to be a sequence of packets passed on a particular channel. Individual packets in a stream z will be referred to by expressions of the form $z[i]$. A stream z will be denoted by an expression of the form $\langle z[1], z[2], \ldots \rangle$. Streams may be finite or (countably) infinite. The size of a stream z, written $\#z$, is the number of packets in it. Two streams are equal if they have the same size and corresponding packets in them are equal. This means that a stream is uniquely determined by its size and by its elements and their ordering. The space of streams for a channel Z is denoted by Z^*. Formally, we have:

__Definition__: A set S of natural numbers is said to be an *initial segment* of the natural numbers iff for any $i \in S$, $j \leq i$ implies $j \in S$.

__Definition__: A *stream* over a space Z is a function mapping some initial segment of the natural numbers into Z. The space of all streams over Z is denoted by Z^*.

__Definition__: The *empty stream* over a space Z, denoted by ϵ or by $\langle\rangle$, is the unique stream over Z having empty domain and no elements.

__Definition__: If i is in the domain of a stream z, we define the i-th element of z, denoted $z[i]$, to be the image of i under z.

Observe that $z[i]$ is undefined if i is not in the domain of z, and that if $z[i]$ is defined then $z[j]$ is defined for all $j \leq i$.

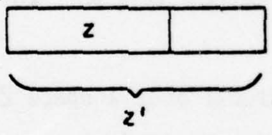
__Definition__: For any stream z, the *size* of z, denoted $\#z$, is the number of elements in the domain of z. If the domain of z is infinite, then we say $\#z = \infty$.

Note that $z[i]$ is defined if and only if $1 \leq i \leq \#z$. In particular, $z[i]$ is defined for all natural numbers if and only if $\#z = \infty$.

__Definition__: Two streams z and z' are said to be *equal*, written $z = z'$, iff $\#z = \#z'$ and $z[i] = z'[i]$ for all $i \leq \#z$.

In our treatment, we shall regard the token "∞" as a distinguished natural number whose arithmetic properties are defined in an obvious manner, such as $i \leq \infty$ and $\infty + i = \infty$ for all natural numbers *i*. The value ∞ may or may not be counted in the range of natural number quantifiers; this depends on context. Because all streams are countable, an expression such as $z[\infty]$ has no meaning, even when z is an infinite stream.

An important relation over streams is the prefix relation. Stream z is a prefix of stream z' whenever z "occurs" at the beginning of z', as shown below:



In such a case, the stream difference z' - z shall be the portion of z' occurring after the prefix z.

For any stream z, we use the special notation z[k:m] to denote the segment of z consisting of the *k*-th through *m*-th elements of z in order. z[k:m] is a stream of size m-k+1, and we allow the special case of an infinite stream when $m = \infty$. If $k > m$, then z[k:m] is the empty stream. As a special case, whenever $k \leq \#z$, z[1:k] is the unique prefix of z of length *k*. This means that $z[1:k][i] = z[i]$ for each $i \leq k$.

Given streams z_1 and z_2 we can form their concatenation $z_1 @ z_2$, which is a stream consisting of the packets in z_1 followed by the packets in z_2. The formal definitions now follow:

<u>*Definition:*</u> Given two streams z, z' over the space Z, we say z is a *prefix* of z', denoted z PREFIX z', if and only if

(1) $\#z \leq \#z'$ and

(2) $i \leq \#z \Rightarrow z'[i] = z[i]$.

<u>*Definition:*</u> For any stream z, if $k \leq m \leq \#z$, then $z[k:m]$ is the unique stream of size $m-k+1$ such that $z[k:m][i] = z[k+i-1]$ for each *i* in its domain.

<u>*Definition:*</u> Given streams z and z' for which z PREFIX z', we define the *difference* z' - z by $z' - z = z'[1+\#z:\#z']$.

<u>*Definition:*</u> For any two streams z_1 and z_2 over the same space Z, their *concatenation* z_1 @ z_2 is the unique stream z of size $\#z_1 + \#z_2$ satisfying

$z[i] = ($*if* $i \leq \#z_1$ *then* $z_1[i]$ *else* $z_2[i-\#z_1])$.

There are two stream operations we will use which count and find particular packets in a stream: count(p,z) is the number of packets in z equal to packet p, and index(p,z,j) is the position in z of the *j*-th occurrence of packet p. They are defined by:

<u>*Definition:*</u> $\text{count}(p,z) = \text{card}\{i \leq \#z: z[i] = p\}$.

<u>*Definition:*</u> $\text{index}(p,z,j) = ($*if* $\exists i \leq \#z: z[i] = p \ \& \ \text{count}(p,z[1:i-1]) = j-1$
then i *else* undefined).

This is well-defined since if such *i* exists, then it is uniquely determined.

Two more important relations over streams are the subsequence and merge relations. A stream z_1 is a subsequence of stream z_2 if the elements of z_1 occur in the same relative order within z_2. They do not have to occur contiguously. A stream z is a merge of streams z_1 and z_2 if and only if z_1 and z_2 occur in z as disjoint subsequences and together exhaust z. All merges of z_1 and z_2 are of length $\#z_1 + \#z_2$. The formal definitions are:

Definition: Given two streams z_1 and z_2 over the space Z, we say z_1 is a *subsequence* of z_2, denoted z_1 SUBSEQ z_2, if and only if there exists a function *f* that maps the domain of z_1 into the domain of z_2 such that

(1) $k_1 < k_2 \Rightarrow f(k_1) < f(k_2)$ and

(2) for each $k \leq \# z_1$, $z_1[k] = z_2[f(k)]$.

A function *f* satisfying properties *(1)* and *(2)* will be called an *insertion*. Any subset S of the domain of a stream z defines a unique subsequence of z which is formed simply by arranging the elements of z indexed by S in increasing order.

Definition: Given three streams z, z_1, z_2 over a common space Z, we say z is a *merge* of z_1 and z_2 if and only if the domain of z can be partitioned into two disjoint subsets, one defining z_1 as a subsequence of z and the other defining z_2 as a subsequence of z.

This concludes the presentation of the fundamentals of streams.

3.3. Examples

In this section we exhibit some elementary packet modules with their *specifications*. The *first* module we describe is the *distribute* module D (figure 3.3-1).

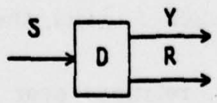


Figure 3.3-1: The distribute module D.

Input slices for D belong to S^* (streams over S) and output slices belong to $R^* \times Y^*$ (pairs of streams over R and Y, respectively). This gives us the space for the characteristic relation $EXT_D \subseteq ((S^*) \times (R^* \times Y^*))$. Within a packet system, module D has the general function of distributing packets through the system to places where they need to be routed. There are no restrictions on the type of packets that may be passed through D. The behavior of module D is to pass unchanged copies of input packets from S onto both output channels Y and R. The response of D to an input stream s is the generation of two output streams r and y identical to s. As with all the modules we describe here, this works for infinite streams as well as finite streams. Thus the behavior of D is defined by

$$((s), (r,y)) \in EXT_D \Leftrightarrow r = y = s.$$

We give a couple of examples of the behavior of D, showing input streams s together with valid responses r and y:

$s = \langle 8,1,6,4\rangle$, $r = \langle 8,1,6,4\rangle$, $y = \langle 8,1,6,4\rangle$;
$s = \langle 1,2,3, \ldots\rangle$, $r = \langle 1,2,3, \ldots\rangle$, $y = \langle 1,2,3, \ldots\rangle$.

The *negation* module N (figure 3.3-2) processes boolean-valued packets, sending out for each input value b a packet whose value is the logical negation not(b).

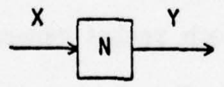


Figure 3.3-2: *The negation module* N.

An output stream y will be a termwise negation of the corresponding input stream x. Formally, $EXT_N \subseteq ((X^*) \times (Y^*))$ and

$$((x), (y)) \in EXT_N \Leftrightarrow \#y = \#x \text{ and } y[i] = not(x[i]) \ \forall i \leq \#y.$$

An example of the behavior of module N is:

x = ⟨true,false,true,true,false⟩, y = ⟨false,true,false,false,true⟩.

The *adder* module A (figure 3.3-3) pairs up integer-valued packets in corresponding positions in its input streams x and r, adds the pairs and sends the sums out as a stream on S.

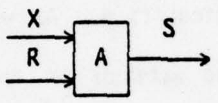


Figure 3.3-3: The adder module A.

If one input stream is longer than the other, the extra packets absorbed from the longer input stream are not reflected in the output response. This is specified by $EXT_A \subseteq ((X^* \times R^*) \times (S^*))$ and

$$((x,r), (s)) \in EXT_A \Longleftrightarrow \#s = \min(\#x, \#r) \text{ and } s[i] = x[i] + r[i] \ \forall i \leq \#s.$$

As examples, we have:

x = ⟨8,1,-6⟩, r = ⟨3,-5,6⟩, s = ⟨11,-4,0⟩;
x = ⟨4,-9,0,-18⟩, r = ⟨ ⟩, s = ⟨ ⟩;
x = ⟨1,3,5,...,2i-1,...⟩, r = ⟨2,4,6,...,2i,...⟩, s = ⟨3,7,11,...,4i-1,...⟩.

A slightly more complicated module is the *cumulative adder* module C (figure 3.3-4) for which each packet generated for output on Y is the sum of all packets received on X so far.

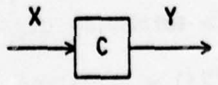


Figure 3.3-4: The cumulative adder module C.

We specify the behavior by $EXT_C \subseteq ((X^*) \times (Y^*))$ and

$$((x), (y)) \in EXT_C \iff \#y = \#x \text{ and } y[i] = \sum_{j=1}^{i} x[j] \;\forall i \leq \#y.$$

As examples of the action of C, we have:

$x = \langle 4,2,-1,0,-6,3\rangle$, $y = \langle 4,6,5,5,-1,2\rangle$;
$x = \langle\ \rangle$, $y = \langle\ \rangle$;
$x = \langle 1,3,5,7,\ldots,2i-1,\ldots\rangle$, $y = \langle 1,4,9,16,\ldots,i^2,\ldots\rangle$.

One of the modules we will be discussing later on is the *feedback modified first* module F (figure 3.3-5), which handles integer packets.

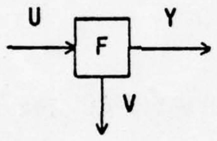


Figure 3.3-5: The feedback modified first module F.

Packets input from U are copied directly onto output channel Y. In addition, the value of the first packet input from U (if there is any) is suitably modified and the resulting value is output as a packet on V. For the purposes of this example, we shall say that the first packet value is modified by adding the number four to it. The behavior of F is specified by $EXT_F \subseteq ((U^*) \times (V^* \times Y^*))$ and

$$((u), (v,y)) \in EXT_F \iff y = u \text{ and } \#v = \min(1,\#u) \text{ and } v[i] = u[i]+4 \;\forall i \leq \#v.$$

As examples, we have:

$u = \epsilon$, $v = \epsilon$, $y = \epsilon$ (empty streams);
$u = \langle 1,2,3\rangle$, $v = \langle 5\rangle$, $y = \langle 1,2,3\rangle$.

A module with an interesting logical function is the *true gate* T (figure 3.3-6), which pairs up integer data inputs from channel X with boolean control inputs from channel C. If the control signal value is true, the corresponding data input from X is passed out on Z. If the control signal is false, the data packet is discarded. Thus the control signal stream C filters out *specified* elements of the data stream X. C must carry boolean packets, and X and Z may pass packets of any type as long as they agree.

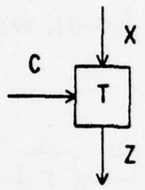


Figure 3.3-6: The true gate T.

The behavior of T is specified by $EXT_T \subseteq ((X^* \times C^*) \times (Z^*))$ and

$$((x,c), (z)) \in EXT_T \iff \#z = \text{count}(\text{true}, c[1:\#x]) \text{ and } z[i] = x[\text{index}(\text{true},c,i)] \ \forall i \leq \#z.$$

As examples, we have:

$x = \langle 1,2,3,4,5\rangle$, $c = \langle \text{true},\text{false},\text{true},\text{true},\text{false}\rangle$, $z = \langle 1,3,4\rangle$;
$x = \langle 6,7\rangle$, $c = \langle \text{false},\text{true},\text{true}\rangle$, $z = \langle 7\rangle$;
$x = \langle 8,9,10,11\rangle$, $c = \langle \text{false},\text{true},\text{true}\rangle$, $z = \langle 9,10\rangle$.

The above modules are all determinate, since for any input slice there is exactly one output slice that constitutes a valid response. Their behavior is therefore functional. Our specification technique may be applied to nondeterminate modules as well, as we now show.

The *nondeterminate merge* module J (fig. 3.3-7) sends out all the packets it receives from input channels U and V onto output channel Z. The relative ordering of packets on each of U and V is preserved, but the packets coming from these two channels are arbitrarily interleaved on output. There is no restriction on the type of packets that may be passed through J.

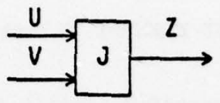


Figure 3.3-7: The nondeterminate merge module J.

We may specify the behavior of J by $EXT_J \subseteq ((U^* \times V^*) \times (Z^*))$ and

$$((u,v), (z)) \in EXT_J \Leftrightarrow z \text{ is a merge of } u \text{ and } v,$$

where the notion of a merge of two streams was defined in the previous section to be a stream containing the two given streams as disjoint subsequences. The size of an output stream z will always be the sum of the sizes of the corresponding input streams u and v.

As an example of the behavior of J, if it is given as inputs the two streams u = ⟨1,2⟩ and v = ⟨3,4⟩, then there are six possible valid output responses: ⟨1,2,3,4⟩, ⟨1,3,2,4⟩, ⟨1,3,4,2⟩, ⟨3,1,2,4⟩, ⟨3,1,4,2⟩ and ⟨3,4,1,2⟩. The output response ⟨1,4,2,3⟩, however, is *not* valid, since the relative ordering of 3 before 4 in the input stream v has not been preserved on output.

In practice, a wide variety of nondeterminate behavior can be realized by constructing systems formed by interconnecting various determinate modules with instances of the module J. In this sense, the nondeterminate merge module J is often viewed as a canonical "source" of nondeterminacy in packet systems.

3.4. Evaluation

We have seen how the slice-relation approach to module specifications works for some simple cases. In this section we address the question of applicability of our method to more complicated modules.

The examples we presented treated only packets of elementary types (integer and boolean). One of the areas of flexibility in packet communication architecture is that systems may be easily designed to process packets which are arbitrarily complex data structures, such as personnel records. Data items in the various fields of a structure-valued packet may be processed concurrently in different internal sections of a system. Direct support for handling packets with arbitrarily complex structure is equally easy in our specification model. All that needs to be added are stream and packet operators for building and decomposing structures, and this is well understood and straightforward: structures are essentially labeled cartesian products of their components, and basic operations on structures have been found in programming languages for a long time.

The basic question to be discussed here is how effectively our specification techniques can model the functional capabilities of modules that are to be physically realized in hardware within packet systems. We claim that the slice-relation descriptive formalism has sufficient power of expression to model the behavior of any realizable packet module. There are several factors that substantiate this claim. Our technique allows the use of arbitrary mathematically defined functions and predicates on packet values and streams. Basic operations on packet values may be composed through the use of conditional expressions and recursion on *streams*. This places at our disposal the functional capabilities of the textual language used to model data flow schemas in [Weng, 1975]. Thus, from the standpoint of Turing computability, the slice-relation approach can model behavior of any desired complexity. Moreover, a module's characteristic relation acts as a predicate that asks of an output *slice* "is this a correct response to the presented input?" Thus, external characteristic relations are the way our model *mathematically* determines correctness of modules in packet systems.

The above arguments say nothing about the complexity of behavioral descriptions in our model. It is an unfortunate fact that as processes one wishes to model increase in complexity, the effort required to formally specify them increases even more rapidly. Although this appears to be the case with packet modules as well as with computer programs, it is hoped that the hierarchical composition of packet systems can reduce the *structural* complexity to be handled if not the functional complexity. Behavioral specifications for the structural composition of packet modules into systems are treated in the following chapter.

CHAPTER 4: SPECIFICATIONS FOR PACKET SYSTEMS

4.1. Internal specifications

The external specifications described in the previous chapter constitute a formal way of defining how a packet system is to interact with its outside world. The most important conceptual property here is that a system is correct whenever is satisfies its external specifications. As we mentioned earlier, correctness of a system cannot be established by outside observation alone; it is necessary to analyze the internal operation of a system in order to prove correctness.

Structurally speaking, a packet system consists of a collection of component modules interconnected by channels. The behavior of a system is determined by two things: its structure and the behavior of its component modules. A formal description of a system's behavior which is based entirely on these two ingredients will be called a set of *internal specifications* for the system because it expresses the system's action in terms of its internal composition.

In order to show a system is correct, two steps must be taken. First, one must produce a set of internal specifications for the system. These internal specifications then must be proved equivalent to the system's external specifications. The logical reasoning involved here is that the component modules are assumed to be correct from the beginning; this assumption is then used throughout the system correctness proof. If one wishes to demonstrate the correctness of a component module, it is decomposed structurally into its own components; this module's correctness is verified in the exact same manner as the entire system. In this way the hierarchical system structuring provided in packet communication architecture supports hierarchical structuring of system verification.

To formally derive the internal specifications for a packet system, two pieces of information are needed: *(1)* a structural description of the system, and *(2)* the external specifications for each of its component modules. It is not necessary to examine the component modules internally, since they are assumed correct. The internal specifications will take the identical form as the external specifications, namely a binary relation between input slices and output slices.

At first glance, coming up with internal specifications for a packet system may appear to be a straightforward task. Consider, for example, the system S1 shown in figure 4.1-1.

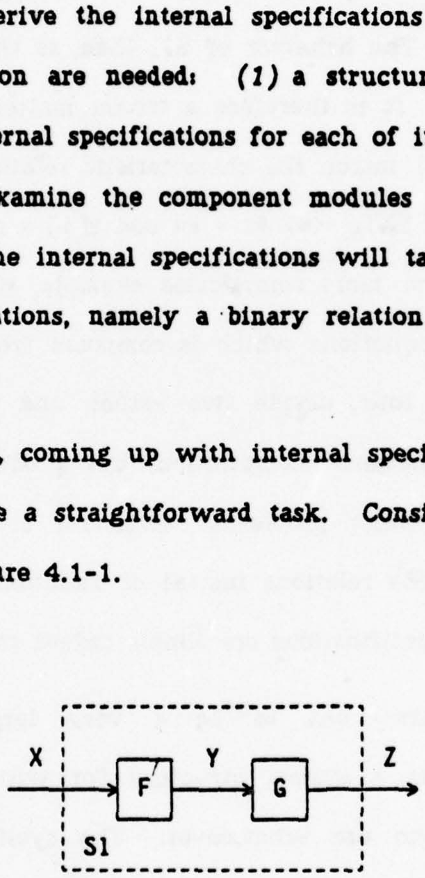


Figure 4.1-1: System S1 acts by functional composition

Suppose that module F applies a function f to each packet value x received on X, sending the resulting value f(x) out as a packet on Y. If F preserves packet ordering, its characteristic relation EXT_F would contain all ordered pairs ((x), (y)) for which y is the stream obtained from stream x by applying f to each packet of x in sequence. In other words,

$$((x), (y)) \in EXT_F \iff \#y = \#x \text{ and } y[i] = f(x[i]) \ \forall i \le \#y.$$

If module G applies a function g in the same manner, *i.e.*

$$((y), (z)) \in EXT_G \iff \#z = \#y \text{ and } z[i] = g(y[i]) \ \forall i \le \#z,$$

then it is easy to see that for each packet entering the system S1, first f and then g is applied. The behavior of S1, then, is the functional composition of modules F and G. It is therefore a trivial matter to show that the internal specifications for S1 match the characteristic relation

$$((x), (z)) \in EXT_{S1} \iff \#z = \#x \text{ and } z[i] = g(f(x[i])) \ \forall i \le \#z.$$

One could take a far more complicated example, such as a system to compute roots of quadratic equations which is composed from modules that take square roots, multiply by four, divide two values, and the like. There would be long chains of functional composition, but producing internal specifications would present no major problems. Even for a nondeterminate system, one could simply compose relations instead of functions. So it seems, at least so far, that internal specifications are simple indeed to determine.

There turns out to be a very large fly in the ointment. Figure 4.1-2 depicts a system structure for which functional or relational composition is of no use whatsoever. The cyclic interconnection structure imposes mutual data dependencies between channels Q and R. Packets passed on channel R from module B depend on the packets received by B from channel Q, while the packets passed on Q depend on earlier packets received by module A from channel R. It is a distinctly nontrivial task to express the stream R in terms of the remaining streams X, Q and Z, since packets passed on R will in general depend on packets previously passed on R. This kind of

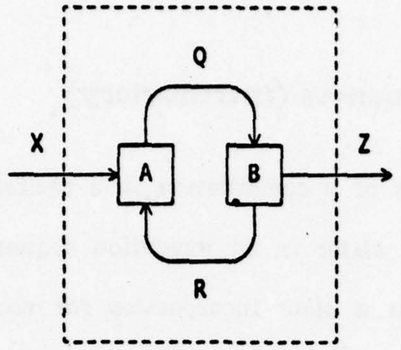


Figure 4.1-2: Cyclic data dependencies

dependency introduces mutually recursive systems of equations expressing the channel streams in terms of one another. Gilles Kahn [Kahn, 1974] has found a way to solve systems of this kind through the use of a mathematical theory of *fixpoints*. His technique, however, requires that the modules be determinate, and there is no straightforward way to apply his techniques to nondeterminate systems. The task of deriving internal specifications for a packet system is a challenging problem, and a new approach is required.

The approach we will be using is based on an *operational* view of systems. We model the operation of a system by recording the progress of a computation in a series of internal system states. The system's response to particular presented input is characterized by a time-ordered progression of internal states, which we call an *execution sequence*. In general, there are a large number of possible execution sequences that correspond to a particular system response to some presented input. A system property we would want to prove must be shown to hold over *all possible* execution sequences that may be taken by the system. The next section informally introduces some of the basic characteristics of execution sequences.

4.2. Execution sequences (introductory)

The progress of a computation in a packet system is modeled by the succession of internal states in an execution sequence. We will be defining internal states so that a state incorporates for each channel the cumulative stream of packets generated to be passed on that channel. This determines, in particular, for each state the input slice presented to the system and the output slice generated by the system so far.

A property we wish execution sequences to have is that one can construct a system state that represents the computation running to completion. For such a state, the output slice represents one of the system's possible ultimate responses to its presented input. Such an execution sequence will be said to *realize* that particular output response to the system's presented input. It will then be a straightforward task to produce the system's internal specifications, which are given by the relation between input slices and corresponding output slices realized by some execution sequence.

A particular kind of physical event we wish to model in an execution sequence is the transmission of a packet on some channel. The act of a module sending a packet out on a channel may occur at any moment between the time the packet is generated by the module and the time the module receives an acknowledge signal for the packet. For any given instant of time during such an interval, the packet may or may not have been sent out already, and we cannot determine which is the case. Thus, our execution sequences will capture two kinds of events: generation of a packet and receipt of the acknowledge signal. Because we do not know the actual moment of transmission, a packet will be regarded as only *potentially* present on the channel during the interval between these two events.

Each state in an execution sequence must reflect the relevant events that have occurred in the system. The events described above are associated with particular channels, so we may partition state information into components relating to the individual channels in the system. To model a state, we give for each channel the cumulative sequence of events of each kind (packet generation and acknowledgment) that have taken place. Packet generation events are handled by giving the stream of generated packets for each channel. Since the channels act as FIFO queues, the packets that have been acknowledged are always given by a prefix of the generated packet stream. We call this prefix the *acknowledged prefix* of the stream. Thus every state in an execution sequence consists of a generated packet stream for each channel together with its acknowledged prefix.

Another significant property of execution sequences is that they are to exhibit the behavior of the component modules of the system. At any state, for each module the generated packet streams on the module's output channels must constitute a valid response by that module to the input packets it has received (and acknowledged).

A transition from one state to the next in an execution sequence models the physical occurrence of a module receiving new input and generating new output packets in response. If there are no more packets generating new output packets in response. If there are no more packets waiting to be absorbed by modules in the system, the system state will remain constant.

We now give some examples of execution sequences for a particular system S shown in figure 4.2-1.

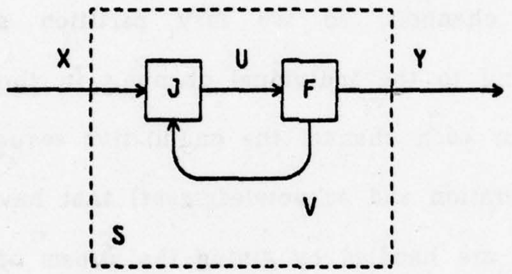


Figure 4.2-1: A sample packet system S

J is the nondeterminate merge module and F is the feedback modified first module; both of these modules were described in the previous chapter. Nondeterminate systems such as S may generate different output responses to a given presented input. This will be reflected in our examples.

An execution sequence is represented by a table in which the rows are the internal states and the columns correspond to channels. Each entry in the table is the appropriate stream of generated packets with a heavy dot marking the end of the acknowledged prefix.

Execution sequence *A*, shown in figure 4.2-2, models a particular response of system S to the input stream ⟨1,2⟩ presented on channel X. We also give a corresponding series of snapshots that illustrate the internal system states during the computation.

state	X	U	V	Y
0	•12	•	•	•
1	1•2	•1	•	•
2	1•2	1•	•5	•1
3	12•	1•2	•5	•1
4	12•	12•	•5	•12
5	12•	12•5	5•	•12
6	12•	125•	5•	•125
7	12•	125•	5•	125•

Figure 4.2-2: Sample execution sequence *A* for system S.

The snapshots, shown in figure 4.2-3, depict the first seven internal system states captured in execution sequence *A*. In state 0, the sequence ⟨1,2⟩ of input packets has not yet entered the system to be processed, and no packets have been acknowledged (all the heavy dots are at the left end of the channel streams). In state 1, the first packet (with value 1) has been received and acknowledged by module J, and a copy has been generated to be sent on channel U. This copy is, by the time of state 2, received and acknowledged by module F. F generates a copy for output on Y, and also a packet with value 5 (1+4) for output on V (since the packet 1 was the first packet received by F on U). In state 3, the input packet 2 will be passed by J onto U, and in state 4 it is generated as output on Y. Note that no further packets are generated for channel V. By state 5, the packet with value 5 has been

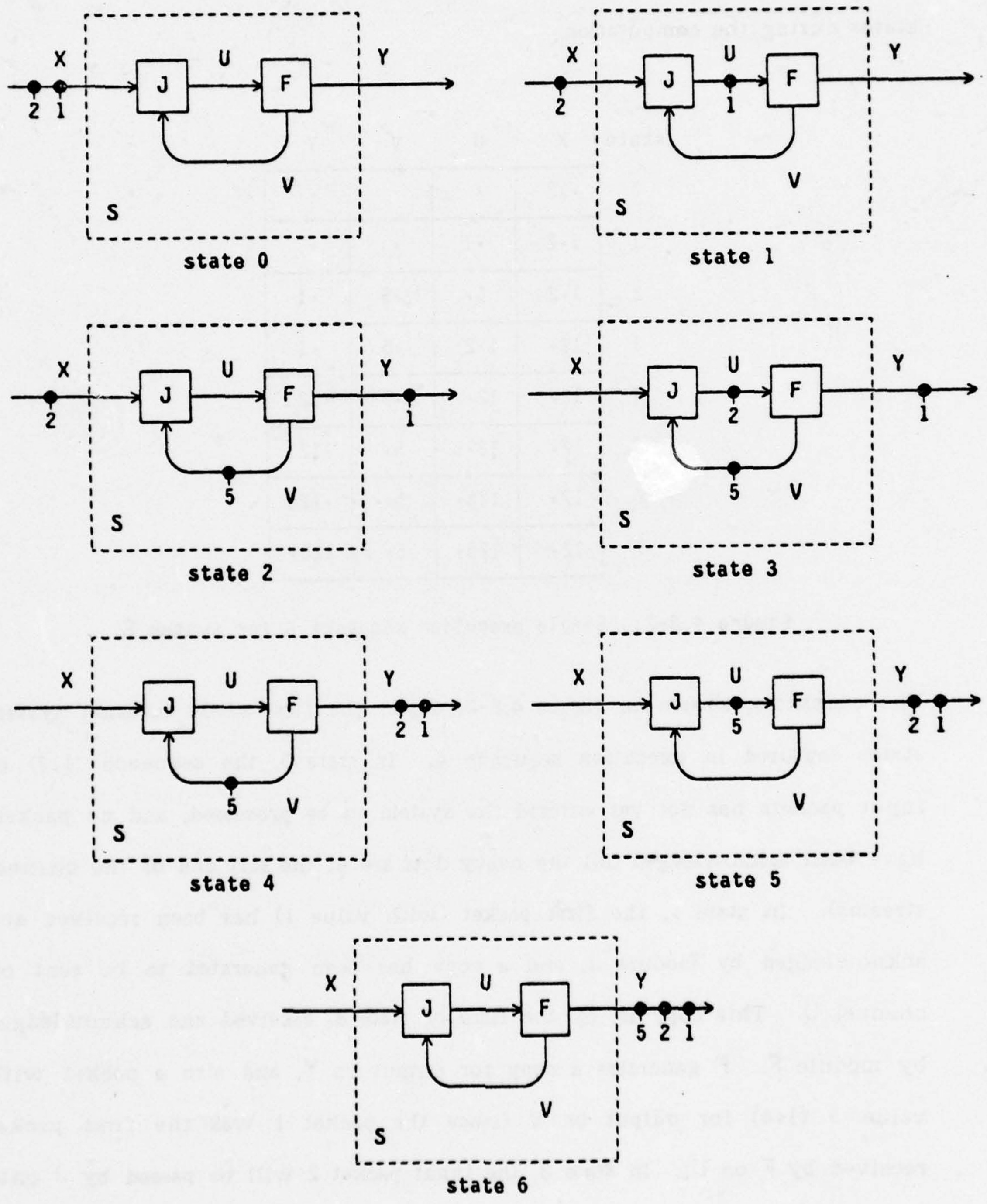


Figure 4.2-3: Snapshots for execution sequence *A*.

processed by J, and by state 6 it has been passed through F. State 6 shows that *system* S's response ⟨1,2,5⟩ to its input ⟨1,2⟩ has been completely generated for output. By state 7 (not shown), these packets have been sent out and acknowledged by their outside world recipient.

We now present another execution sequence that models the response of *system* S to the same presented input stream ⟨1,2⟩. Execution sequence *B*, shown in figure 4.2-4, is identical to execution sequence *A* except for states 2 and 4.

state	X	U	V	Y
0	•12	•	•	•
1	1•2	•1	•	•
2	12•	•12	•	•
3	12•	1•2	•5	•1
4	12•	1•25	5•	•1
5	12•	12•5	5•	•12
6	12•	125•	5•	•125
7	12•	125•	5•	125•

Figure 4.2-4: Sample execution sequence *B* for system S.

From state 1 to state 3, this execution sequence has module J receive and process the packet 2 before module F processes the packet 1, reversing the order of these two events from the way they were in execution sequence *A*. Similarly, from state 3 to state 5 here, J takes in the packet 5 before F processes the packet 2. The snapshots of states 2 and 4 for execution sequence *B* are shown in figure 4.2-5.

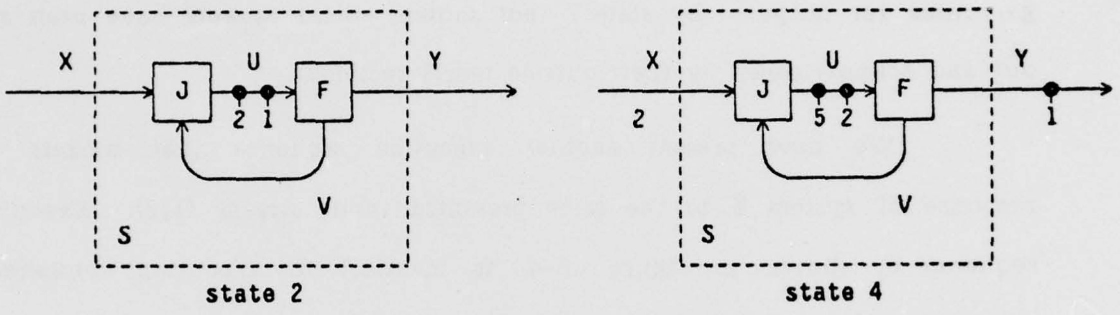


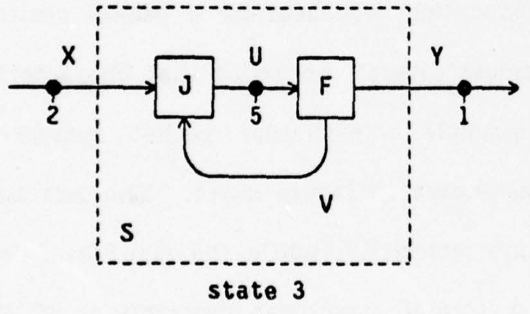
Figure 4.2-5: Snapshots for execution sequence *B*.

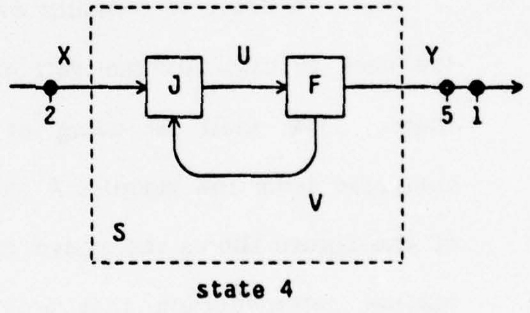
Observe that the two distinct execution sequences *A* and *B* model two distinct computations for the system S, both resulting in the same system response ⟨1,2,5⟩ to the presented input ⟨1,2⟩. On the other hand, execution sequence C, shown in figure 4.2-6, models a computation in which the system produces a different response ⟨1,5,2⟩ to the same input. This sequence is identical with execution sequence *A* through state 2, but now module J processes the packet 5 from channel V before it takes the packet 2 from channel X. This difference is what causes the change in system response. Snapshots for the resulting states 3 through 6 for execution sequence *C* are shown in figure 4.2-7.

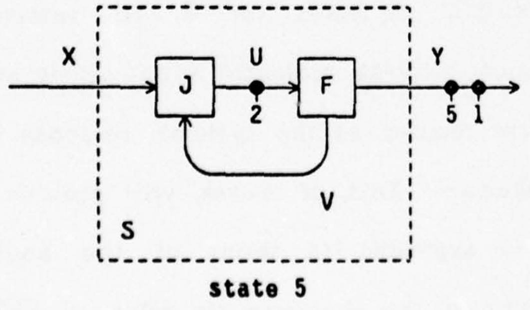
It is important to note that at any time during a computation in a packet system, a packet that has been generated to be sent out on some channel may or may not actually have been sent out already. After the packet is acknowledged we know it has been sent out, but before

state	X	U	V	Y
0	•12	•	•	•
1	1•2	•1	•	•
2	1•2	1•	•5	•1
3	1•2	1•5	5•	•1
4	1•2	15•	5•	•15
5	12•	15•2	5•	•15
6	12•	152•	5•	•152
7	12•	152•	5•	152•

Figure 4.2-6: Sample execution sequence C for system S.







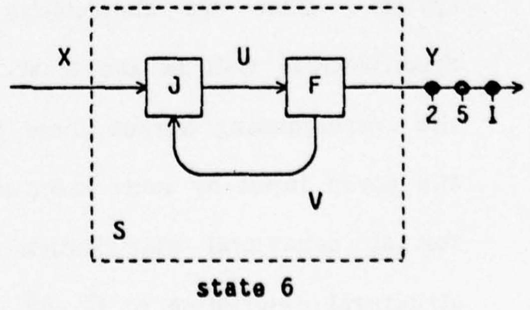


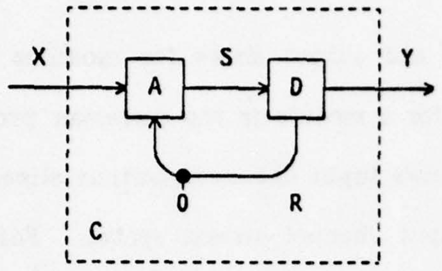
Figure 4.2-7: Snapshots for execution sequence C.

acknowledgment it is only *potentially* on the channel. "Potential" packets are guaranteed to have been by some future time *eventually* passed on the channel in the relative order given, but we can draw no stronger conclusions. This means that in all the snapshots we have depicted here, the packets shown on the various channels were at the indicated time only *potentially* present.

This concludes our informal introduction to execution sequences. In the next section we shall motivate and discuss the properties that will be used to characterize them formally.

4.3. Properties of execution sequences

In order to formally define execution sequences for a packet system, we need to carefully motivate and discuss several properties that characterize them. We shall be using as an example a particular packet system C composed from the modules A and D as shown in figure 4.3-1. The left half of the figure shows the system structure pictorially, while the right half is a textual representation that provides a formal structural description of the system. Once we characterize execution sequences for C, its internal specifications will be the binary relation between presented input slices and the corresponding output slices that are realized as the system's response to the given input by some execution sequence. This, of course, will provide a formal behavioral specification for C expressed in terms of the above structural description of C and in terms of the characteristic relations EXT_A and EXT_D for the component modules A and D. In the previous chapter, we



```
System C
    inputs X(integer)
    outputs Y(integer)
    internals S(integer), R(integer)
Submodules
    A inputs X, R; outputs S
    D inputs S; outputs R, Y
Initially R<0>
```

Figure 4.3-1: Realization of a sample packet system C

specifically defined the external specifications for A and D, but in our treatment here the characteristic relations shall be viewed abstractly.

An execution sequence is a time-ordered progression of internal states of a packet system, and a state gives particular information about each channel in the system. The state information for a channel Z at any given moment contains, as we mentioned earlier, both the stream of packets generated to be passed on Z and its acknowledged prefix. The space of streams of packets passed on Z is denoted by Z^* and includes infinite as well as finite streams. For any stream $z \in Z^*$, we denote its acknowledged prefix by z^a. A channel state for Z will then be an ordered pair of the form $\langle z,z^a \rangle$.

The state information for a system is simply the collection of state information on all of its channels. For our sample system C, define the space $CSYS^*$ to be the cartesian product of the channel packet stream spaces X^*, S^*, R^* and Y^*. Elements of $CSYS^*$, which are called *system slices*, are denoted \$ (the dollar sign is pronounced "slice" !) and are tuples of the form (x,s,r,y), where x, s, r and y are streams of integer packets. A system state will consequently be an ordered pair of the form $\langle \$,\$^a \rangle$, where the acknowledged prefix $\$^a$ of the slice \$ is the tuple whose components are the acknowledged prefixes of the respective components of $.

We have already defined input and output slices for modules in a packet system. The space of input slices for a module is the cartesian product of the channel stream spaces for the module's input channels; output slices are similarly built up from the module's output channel stream spaces. For the module A in our example, these two spaces are $AIN^* = (X^* \times R^*)$ and $AOUT^* = (S^*)$; for the module D they are $DIN^* = (S^*)$ and $DOUT^* = (R^* \times Y^*)$. The same thing can be done for the system C by viewing it as a module: $CIN^* = (X^*)$ and $COUT^* = (Y^*)$. Thus the characteristic relations for the system C and its two component modules A and D are given by $EXT_C \subseteq (X^*) \times (Y^*)$, $EXT_A \subseteq ((X^* \times R^*) \times (S^*))$ and $EXT_D \subseteq ((S^*) \times (R^* \times Y^*))$. We will have $((x), (y)) \in EXT_C$ if and only if the output stream y is a valid response to the input stream x under the semantic properties of the system C.

Execution sequences for a packet system will be of the form $\{\langle \$_i, \$_i^a \rangle\}$, where *i* takes on natural number values starting from zero. $\$_i^a$ will be the acknowledged prefix of the *i*-th system slice $\$_i$. There are a number of semantic properties which an execution sequence must satisfy in order to correctly model the action of a packet system. We describe them here in terms of the sample packet system C, noting that the generalization to arbitrary packet systems presents no difficulty. For the system C, the components of system slice $\$_i$ are denoted by $\$_i = (x_i, s_i, r_i, y_i)$.

The first condition an execution sequence must satisfy is that there be a valid initial system state. To express the property that no packets have been processed at the start, we require that the initial state $\langle \$_0, \$_0^a \rangle$ have an empty acknowledged prefix $\$_0{}^a$. The components of $\$_0$ corresponding to input channels must match the presented input slice. In our case, this means that x_0 must be equal to a given stream x of inputs. And it must also be the case that the other components of $\$_0$ agree with the initial configuration defined by the system structure. For system C, this requires that we have $s_0 = y_0 = \epsilon$ (empty streams) and $r_0 = \langle 0 \rangle$ (stream of one zero-valued packet).

An execution sequence is supposed to reflect a system's response to a particular presented input slice, and this input slice appears in its entirety within the initial system slice $\$_0$. In order for the execution sequence to realize a response to precisely this input and nothing more, we must have at each system state the identical input slice as at the beginning, which for the system C means that $x_i = x_0$ for all i. Physically, this requirement amounts to the outside world suspending additional input to the system until the system completes its response to the input already presented.

The third condition that must be fulfilled is agreement with the semantic properties of the component modules of the system. What this means is that for all states it must be true of each module that the packets that have been received and acknowledged by that module are related through the module's characteristic relation to the output packets generated by that module. In our system, the semantics for the A module impose the condition $((x_i{}^a, r_i{}^a), (s_i)) \in \mathrm{EXT}_A$, and the D module forces $((s_i{}^a), (r_i - r_0, y_i)) \in \mathrm{EXT}_D$. (The reason we specifically remove the stream r_0 is that it represents a packet stream that is initially present but is not generated as output by any module.) These conditions must hold for each i indexing some state in the execution sequence, starting from the initial state with $i=0$.

The fourth property that should hold within an execution sequence is rather complex. We wish to state precisely the requirement that state transitions within an execution sequence must agree with the system structure. Each state $\langle \$_{i+1}, \$_{i+1}^a \rangle$ must follow from its predecessor $\langle \$_i, \$_i^a \rangle$ in a manner consistent with the physical arrangement of the system's channels. Once a packet is sent out along a channel, it can never be "unsent" or called back. For each channel Z in the system, packets can only be added in going from one state to another. Moreover, since the channels act as FIFO queues, new packets cannot disturb the relative order of previous packets. Thus, for each channel Z, the channel stream z_i must be a subsequence of z_{i+1} for all *i*. This requirement also holds separately for the acknowledged prefixes on each channel, since acknowledged packets cannot become "unacknowledged," so we must also have z_i^a as a subsequence of z_{i+1}^a for all *i*.

It would greatly simplify the technical development in the following section if we could strengthen this fourth condition to require that z_i be a *prefix* of z_{i+1} rather than any subsequence. As it stands now, we are requiring that a module can only send out additional packets in response to new input packets received. Insisting on a prefix property would impose a time restriction on the intervals from packet generation to packet transmission, forcing packets to be sent out on channels in the exact same order in which their respective processes of generation were initiated. Unfortunately, this turns out to be too strong a stipulation. If a module such as M (figure 4.3-2) receives from its input channel X first a packet *p* and later a

packet *q*, it may very well take M longer to produce a packet *p'* in response to *p* than to produce a packet *q'* in response to *q*.

$$\begin{tikzcd} {} \arrow[r, "X"] & \boxed{M} \arrow[r, "Y"] & {} \end{tikzcd}$$

Figure 4.3-2: A module M.

This could occur naturally in applications such as a cache/bulk memory or an information retrieval system. In order for M to derive the benefits of asynchronous operation, its behavior should be specified nondeterminately so that either stream ⟨*p'*,*q'*⟩ or ⟨*q'*,*p'*⟩ will be a valid response to the input stream ⟨*p*,*q*⟩. *Figure 4.3-3 depicts the two corresponding execution sequences, which* should both be valid.

state	X	Y
0	•pq	•
1	p•q	•p'
2	pq•	•p'q'
3	pq•	p'q'•

(a)

state	X	Y
0	•pq	•
1	p•q	•p'
2	pq•	•q'p'
3	pq•	q'p'•

(b)

Figure 4.3-3: Two execution sequences for M.

In execution sequence *(a)*, channel stream y_1 = ⟨*p'*⟩ *is a prefix of* channel stream y_2 = ⟨*p'*,*q'*⟩. However, in sequence *(b)*, the packet *q'* has cut ahead of the packet *p'* by the time state 2 occurs. This is legal, since the *p'* packet is

only potentially present on Y during state 1. So for sequence *(b)*, $y_1 = \langle p' \rangle$ is a subsequence and not a prefix of $y_2 = \langle q', p' \rangle$. In fact, there is no way to realize the response described by execution sequence *(b)* if we insist that y_1 be a prefix of y_2. We need the generality of the subsequence relation to realize "cutting ahead" behavior of this nature in packet systems. Thus we cannot strengthen the requirement that each channel stream in an execution sequence be a subsequence of its successor.

We can, on the other hand, strengthen this subsequence property to use the prefix relation in the case of acknowledged prefixes of channel states. The "cutting ahead" behavior as described above cannot occur within the acknowledged prefix of a channel stream, since we know that all the packets here have already been passed. This means that in any execution sequence, the only way $z_{i+1}{}^a$ may differ from $z_i{}^a$ is through the appending of newly acknowledged packets to the end of the stream. Thus $z_i{}^a$ cannot be just any subsequence of $z_{i+1}{}^a$; it must be a prefix.

The fifth and final condition that must be satisfied by an execution sequence is that no channel may receive acknowledgment for a packet that was never generated as output to be sent on that channel. This is guaranteed by requiring that for each *i* the acknowledged prefix $z_{i+1}{}^a$ must be an initial segment of the previous stream z_i on all channels Z.

The notion of execution sequences that has been developed here models the progress of a computation within a packet system, but there is one final element that is missing: the idea of ultimate result of a computation. We must identify when a packet system finishes reacting to its input as well as handle the cases of infinite inputs and infinite responses to finite inputs. This will be done by developing the concepts of *limits* and *completeness* for execution sequences.

For any packet system, we may define a relation PRECEDES on system states by $(\$_i, \$_i^a)$ PRECEDES $(\$_j, \$_j^a)$ iff ($\$_i^a$ PREFIX $\$_j^a$ and $\$_i$ SUBSEQ $\$_j$). Intuitively, increasing values with respect to PRECEDES indicate forward progress of a computation within a packet system. In particular, *S1* PRECEDES *S2* must hold whenever system state *S2* is reachable from system state *S1* in some computation through the processing of additional packets. We may observe that PRECEDES is a transitive relation. Furthermore, by condition (4) above, an execution sequence is monotonically increasing with respect to PRECEDES. An upper bound of an execution sequence, then, corresponds to a computation that has progressed at least as far as all the states in the sequence, while a least upper bound indicates that no extraneous computation is taking place. We define a *limit* of an execution sequence to be a least upper bound with respect to the PRECEDES relation. Thus, a limit of an execution sequence corresponds to a system state in which all the computation specified by the sequence runs to completion. This notion applies to infinite as well as finite computations. We use the notation $\lim \{(\$_i, \$_i^a)\} = \sup_{\text{PRECEDES}} \{(\$_i, \$_i^a)\}$ to denote the limit (least upper bound) of an execution sequence when it is well-defined and unique.

It may be observed that the PREFIX relation is a partial order and that for any execution sequence $\{(\$_i, \$_i^a)\}$ the sequence $\{\$_i^a\}$ is monotonically increasing with respect to PREFIX and always has a uniquely defined least upper bound $\$_\infty^a = \sup_{\text{PREFIX}} \{\$_i^a\}$. These facts are proved in the next section. However, least upper bounds are not necessarily well defined with respect to PRECEDES. We therefore need some additional properties to be satisfied by an execution sequence in order to guarantee that limits exist and are well defined.

Consider a system state $\langle \$, \$^a \rangle$ in which $\$^a$ is a *proper* prefix of \$. The nonempty difference slice $\$ - \$^a$ would represent packets that have been generated but not yet acknowledged. Such a state can never represent a complete computation, since it specifies packets still awaiting processing by various internal modules. If the system is to fully respond to its inputs, all the packets that have been generated at any time during a computation must eventually be acknowledged. We thus define an execution sequence $\{\langle \$_i, \$_i^a \rangle\}$ to be *complete* if and only if for each *i* there exists a *j* such that $\$_i$ SUBSEQ $\$_j^a$. This *j* will be the state by which time all packets that have been generated by the time of state *i* will have been sent out and acknowledged. In general, in any state $\langle \$, \$^a \rangle$ for which $\$ = \$^a$, there are no generated packets waiting for processing and acknowledgment, so the system cannot perform any further actions. We prove in the next section that any complete execution sequence $\{\langle \$_i, \$_i^a \rangle\}$ has a unique and well defined limit $\langle \$_\infty, \$_\infty^a \rangle$ for which $\$_\infty = \$_\infty^a$. This result will be known as the *Limit Existence Theorem.* Thus the notion of a computation running to completion within a packet system is always well defined.

The limit of a complete execution sequence should always represent the state of the system upon completing its ultimate output response to the presented input. For a given input slice, we call such a state a *limit state*, and we say that the slice consisting of the streams for the system output channels in a limit state is an *ultimate output slice*. The presented input slice and the ultimate output slice may each be finite or infinite. If either is infinite, there will be infinitely many states in a complete execution sequence and the limit state will not be one of the states in the sequence. We shall adopt the convention that execution sequences will *always* be infinite. If both the presented input and ultimate output slices are finite, then the limit state will be an element of the execution sequence, and all succeeding elements will be identical to this state.

There is a class of pathological conditions under which the limit of a complete execution sequence fails to represent the system's ultimate output response to the presented input. Consider the case of a module M (figure 4.3-4),

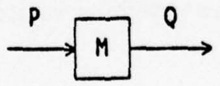


Figure 4.3-4: A discontinuous module.

which outputs the empty stream for finite input but which echoes any infinite input stream. The external characteristic relation EXT_M is given by $EXT_M \subseteq ((P^*) \times (Q^*))$ and

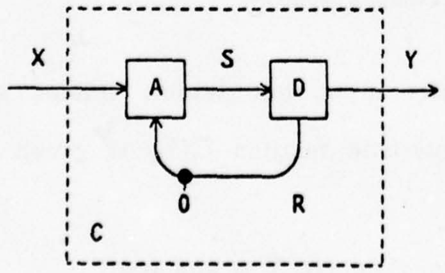
$$((p), (q)) \in EXT_M \iff (\#p < \infty \text{ and } q=\epsilon) \text{ or } (\#p = \infty \text{ and } q=p).$$

In response to input streams p_i of increasing finite length, M will not send out any packets at all, and the limit of a complete execution sequence modeling this behavior will exhibit an empty ultimate output stream q_∞. But this disagrees with M's specified nonempty response to infinite input. The problem lies in the way EXT_M is specified; we may avoid this by requiring that all modules in packet systems be *continuous*, which means that the responses to an increasing sequence of input streams must tend to an appropriate, well-defined limit. When this is the case, we are guaranteed that the limit of a complete execution sequence does in fact properly capture the system's ultimate output response.

We now have described all the relevant characteristics of execution sequences. The mathematical development follows in the next section.

4.4. Execution sequences (formally)

We now give the formal characterization for the notion of execution sequences that has been developed. First, we show an example; afterwards, we give the definition for the general case. Consider the sample system C, which was discussed in the previous section and is shown here again:



```
System C
    inputs X(integer)
    outputs Y(integer)
    internals S(integer), R(integer)
Submodules
    A inputs X, R; outputs S
    D inputs S; outputs R, Y
Initially R<0>
```

Figure 4.4-1: Realization of a sample packet system C

we have the following characterization:

An infinite sequence $\{\langle \$_i, \$_i^a \rangle\}$ in which for each natural number i $\$_i^a = (x_i^a, s_i^a, r_i^a, y_i^a)$ is an acknowledged prefix of $\$_i = (x_i, s_i, r_i, y_i)$ will be an execution sequence for C if and only if the following five conditions hold:

(1) *[initial state]* $\$_0^a = (\epsilon,\epsilon,\epsilon,\epsilon)$, $s_0 = y_0 = \epsilon$, $r_0 = \langle 0 \rangle$

(2) *[input suspension]* $x_i = x_0$ for all i

(3) *[consistency]* $\forall i:((x_i^a, r_i^a), (s_i)) \in EXT_A$ and $((s_i^a), (r_i - r_0, y_i)) \in EXT_D$

(4) *[FIFO]* $\$_i^a$ PREFIX $\$_{i+1}^a$ and $\$_i$ SUBSEQ $\$_{i+1}$ for all i

(5) *[connection]* $\$_{i+1}^a$ PREFIX $\$_i$ for all i

An execution sequence $\{\langle \$_i, \$_i^a \rangle\}$ for system C is complete if and only if $\forall i\ \exists j$: $\$_i$ SUBSEQ $\$_j^a$.

Note that although the PREFIX and SUBSEQ relations were defined over streams, they are being applied to system slices here. The intent is for these relations to be taken componentwise over all channel streams, which means that one slice is a prefix of a second if and only if each component channel stream in the first slice is a prefix of the matching channel stream in the second slice. Subsequences are treated in the same way.

The above formal characterization of execution sequences for the system C may be extended to arbitrary packet systems with no difficulty. The formal structural definition for a packet system is of the general form

```
System SYS
    inputs  W(---), ..., X(---)
    outputs Y(---), ..., Z(---)
    internals U(---), ..., V(---)
Submodules
    .
    .
    .
    M inputs P, ..., Q; outputs R, ..., S
    .
    .
    .
Initially U<u0>, ..., V<v0>,   Y<y0>, ..., Z<z0>.
```

The parenthesized items are channel packet types and may be arbitrary. (The use of consecutive letters in the alphabet separated by ellipses, such as "P, ..., Q" allows an arbitrary number of items in between, so that for example a submodule M of the system may have any number of input channels.)

The generalized definitions now become:

Definition: A sequence $\{(\$_i, \$_i^a)\}$ of system states for a system SYS whose structural description is as given above will be an execution sequence for SYS if and only if

(1) *[initial state]* $\$_0^a = (\epsilon,\ldots,\epsilon)$, $u_0 = u0$, ..., $v_0 = v0$, $y_0 = y0$, ..., $z_0 = z0$
(2) *[input suspension]* $\forall i$: $w_i = w_0$, ..., $x_i = x_0$
(3) *[consistency]* For each module M in SYS we have
$$\forall i\text{: } ((p_i^a,\ldots,q_i^a), (r_i - r_0,\ldots,s_i - s_0)) \in EXT_M$$
(4) *[FIFO]* $\$_i^a$ PREFIX $\$_{i+1}^a$ and $\$_i$ SUBSEQ $\$_{i+1}$ for all i
(5) *[connection]* $\$_{i+1}^a$ PREFIX $\$_i$ for all i

Definition: An execution sequence $\{(\$_i, \$_i^a)\}$ for a system SYS is complete if and only if $\forall i\ \exists j$: $\$_i$ SUBSEQ $\$_j^a$.

We will thus be able to give internal specifications for any packet system.

The relations PREFIX and SUBSEQ were defined in section 3.2. We now proceed to derive the basic mathematical properties for these two relations and the PRECEDES relation. This will lead up to a proof of the Limit Existence Theorem, which states that limits exist and are well-defined for complete execution sequences.

Lemma 1: For any space Z, the PREFIX relation is a partial ordering over Z^*.

Proof: The reflexive and transitive properties are clearly satisfied. Now if z PREFIX z' and z' PREFIX z, then $\#z \leq \#z'$ and $\#z' \leq \#z$, so $\#z = \#z' = N$, which means z and z' have the same domain. But then for $i \leq N$ we have $z[i] = z'[i]$, which means z and z' coincide over their common domain. This forces $z = z'$ and establishes the antisymmetry property, completing the proof.

Definition: A sequence $\{z_i\}$ of streams is said to be *monotone* if for each *i*, z_i PREFIX z_{i+1}.

Lemma 2: Any monotone sequence $\{z_i\}$ of streams has a unique and well defined least upper bound.

Proof: Each stream z_i is a function that may be regarded as a set of ordered pairs of the form $\langle k, z_i[k]\rangle$. Let z be the set-theoretic union of all the z_i. Then z will be a function, since any two ordered pairs $\langle k, z_i[k]\rangle$ and $\langle k, z_j[k]\rangle$ must coincide (by monotonicity). It is immediately apparent that $z \in Z^*$ and z is an upper bound for $\{z_i\}$ under PREFIX. Moreover, z will be a least upper bound, since any upper bound for $\{z_i\}$ must contain all the z_i set-theoretically and hence their union z. Finally, uniqueness follows from the antisymmetry property derived in Lemma 1.

Lemma 3: PREFIX is a subrelation of SUBSEQ.

Proof: The insertion function required by the formal definition of SUBSEQ is simply the identity function.

It is easy to see that the SUBSEQ relation is reflexive and transitive. However, it is not necessarily antisymmetric! Consider the two infinite streams $(0011)^\infty$ and $(0101)^\infty$, each consisting of infinitely many zeros and ones. These streams are distinct, but each is a subsequence of the other. Thus, SUBSEQ is not a partial ordering relation.

The relations PREFIX and SUBSEQ both apply to streams, but the PRECEDES relation will be taken over channel states, which we now define.

Definition: A *channel state* for a channel Z in a packet system is an ordered pair of the form $\langle z, z^a\rangle$ in which z and z^a are sequences of packets and z^a PREFIX z.

Definition: For two channel states $\langle z_i, z_i^a\rangle$ and $\langle z_{i+1}, z_{i+1}^a\rangle$ we say $\langle z_i, z_i^a\rangle$ PRECEDES $\langle z_{i+1}, z_{i+1}^a\rangle$ if and only if z_i SUBSEQ z_{i+1} and z_i^a PREFIX z_{i+1}^a.

Definition: A sequence $\{\langle z_i, z_i^a\rangle\}$ of channel states is said to be *monotone* if and only if $\langle z_i, z_i^a\rangle$ PRECEDES $\langle z_{i+1}, z_{i+1}^a\rangle$ for all *i*.

Definition: A sequence $\{\langle z_i, z_i^a\rangle\}$ of channel states is said to be *complete* if and only if $\forall i\ \exists j$ *s.t.* z_i SUBSEQ z_j^a.

It is extremely important to note that the relation PRECEDES fails to be a partial order because of its SUBSEQ component. This is easily seen in the case of the two channel states $\langle (0011)^\infty, \epsilon\rangle$ and $\langle (0101)^\infty, \epsilon\rangle$, each having an infinite stream and an empty acknowledged prefix. These states are distinct and each precedes the other, so the antisymmetry property fails here. Thus least upper bounds for a monotone sequence of channel states are not necessarily well defined. However, completeness is a sufficient condition to guarantee that the least upper bounds exist and are unique. The following theorem proves this fact.

Theorem: *If* $\{\langle z_i, z_i^a\rangle\}$ *is a monotone and complete sequence of channel states* and if $z_\infty = \sup_{\text{PREFIX}} \{z_i^a\}$, then $\sup_{\text{PRECEDES}} \{\langle z_i, z_i^a\rangle\}$ is well defined and unique and equal to $\langle z_\infty, z_\infty\rangle$.

Proof: Since z_∞ is by definition an upper bound for $\{z_i^a\}$, we have

(1) $\forall i$: z_i^a PREFIX z_∞.

Now given any *i*, by completeness we have

(2) $\exists j$: z_i SUBSEQ z_j^a.

But z_j^a PREFIX z_∞, which by Lemma 3 implies

(3) z_j^a SUBSEQ z_∞.

Since SUBSEQ is transitive, equations *(2)* and *(3)* yield

(4) z_i SUBSEQ z_∞.

The combination of equations *(1)* and *(4)* establishes $\langle z_\infty, z_\infty\rangle$ as an upper bound for $\{\langle z_i, z_i^a\rangle\}$ under the PRECEDES relation.

In order to show that this upper bound is in fact a least upper bound, we must establish that for any channel state $\langle z, z^a\rangle$ for which

(5) $\forall i$: $\langle z_i, z_i^a\rangle$ PRECEDES $\langle z, z^a\rangle$,

it must be the case that $\langle z_\infty, z_\infty\rangle$ PRECEDES $\langle z, z^a\rangle$. Now equation *(5)* implies z_i SUBSEQ z and z_i^a PREFIX z^a. Then the least upper bound z_∞ of $\{z_i^a\}$ must be a prefix of the upper bound z^a, *i.e.*

(6) $$z_\infty \text{ PREFIX } z^a.$$

But since $\langle z,z^a\rangle$ is a channel state, z^a PREFIX z, so z_∞ PREFIX z, which implies

(7) $$z_\infty \text{ SUBSEQ } z.$$

The combination of equations (6) and (7) yields the result $\langle z_\infty,z_\infty\rangle$ PRECEDES $\langle z,z^a\rangle$, so we now have established that $\langle z_\infty,z_\infty\rangle$ is a least upper bound for $\langle z_i,z_i^a\rangle$.

The proof is not yet complete, since the PRECEDES relation is not necessarily antisymmetric and we must therefore explicitly guarantee the well-definedness and uniqueness of the least upper bound we produced. This will follow directly if we show that for any channel state $\langle z,z^a\rangle$, whenever $\langle z_\infty,z_\infty\rangle$ PRECEDES $\langle z,z^a\rangle$ and $\langle z,z^a\rangle$ PRECEDES $\langle z_\infty,z_\infty\rangle$ then it must be true that $z = z^a = z_\infty$. Now z_∞ PREFIX z^a PREFIX z_∞ implies $z^a = z_\infty$. Also, the combination z^a PREFIX z SUBSEQ z_∞ implies $\#z^a \leq \#z \leq \#z_\infty$, and this "*squeeze*" condition forces $\#z^a = \#z$. But since z^a PREFIX z, we must have $z^a = z$. Thus $z = z^a = z_\infty$, which sets up the required antisymmetry condition and guarantees uniqueness of the least upper bound. This completes the proof.

All of the results established here have been stated for individual channels in a packet system. However, we may apply them to the internal behavior of an entire system in a rather straightforward manner. As an example, a system slice \$ is a prefix of a slice \$' if and only if each component stream in \$ is a prefix of the corresponding component stream in \$'. All properties of the PREFIX stream relation are just as valid for the PREFIX slice relation. Similarly, all properties of the stream relation SUBSEQ hold for slices. Moreover, all properties of the PRECEDES relation on channel states apply to system states. In particular, the following theorem, which we call the *Limit Existence Theorem*, holds:

<u>*Theorem:*</u> If $\{\langle \$_i, \$_i^a\rangle\}$ is a complete execution sequence for a packet system, and if $\$_\infty = \sup_{\text{PREFIX}} \{\$_i^a\}$, then $\sup_{\text{PRECEDES}} \{\langle \$_i, \$_i^a\rangle\}$ is well defined and unique and equal to $\langle \$_\infty, \$_\infty\rangle$.

We now give a formal definition for the notion of continuity, which was mentioned in the previous section. Continuity is a property of a module's external characteristic relation, so we define it for binary relations over *slices*:

Definition: A relation ~ on slices is *continuous* if whenever $\$ = \sup_{\text{PREFIX}} \{\$_i\}$, where the sequence $\{\$_i\}$ of slices satisfies $\$_i$ PREFIX $\$_{i+1}$ for all i, then

$(\$,\$') \in \sim \;\Leftrightarrow\; \exists$ a sequence $\{\$_i'\}$ of streams such that

(1) $(\$_i,\$_i') \in \sim$ for all i;

(2) $\$_i'$ SUBSEQ $\$_{i+1}'$ for all i; and

(3) $\$' = \sup_{\text{SUBSEQ}} \{\$_i'\}$ is uniquely defined.

4.5. Characterization of internal specifications

Now that we have defined execution sequences for any packet system, it is simple to produce a system's internal specifications. The internal specifications for a packet system SYS are a binary relation INT_{SYS} from system input slices to system output slices, which we call the system's *internal characteristic relation.* For the sample system C we have been discussing, we have $\text{INT}_C \subseteq ((X^*) \times (Y^*))$, and the internal specifications may be formally characterized by $((x), (y)) \in \text{INT}_C$ if and only if there is a complete execution sequence $\{\langle \$_i, \$_i^a \rangle\}$ for C such that $x_0 = x$ and $y_\infty = y$, where x_0 and y_∞ are defined by $\$_0 = (x_0, s_0, r_0, y_0)$ and $\$_\infty = \sup_{\text{PREFIX}} \{\$_i^a\} = (x_\infty, s_\infty, r_\infty, y_\infty)$. Note that x_0 represents the initial input presented to C and that y_∞ represents the ultimate output yielded by C. We can easily generalize this to an arbitrary system by quantifying the condition $x_0 = x$ over all input channels X and quantifying the condition $y_\infty = y$ over all output channels Y. Note that the definition of INT_{SYS} is in effect parameterized by the structural description of system SYS and by the characteristic relations of the component modules in SYS.

The development of internal specifications for packet systems is now complete. We have two ways of formally describing the behavior of a packet system: externally, in terms of its interaction with the outside world, and internally, in terms of its structure and composition. We can apply this to correctness proofs by observing that a system SYS is correctly realized by its internal structure if and only if its (external) characteristic relation EXT_{SYS} and its internal characteristic relation INT_{SYS} are identical. A correctness proof for a packet system will therefore consist of a demonstration that each of these two relations is contained in the other.

Aside from the obvious application to system verification, the formal specifications we have developed for packet systems are valuable in achieving a frequently overlooked objective: understanding the behavior of these systems. Our operational approach lets us model the activity within a system step by step. The "dot notation" tables for execution sequences are a useful pedagogical tool, aiding in a person's conceptualization of what goes on in packet systems. It is hoped that even without going through a process of formal verification, designers of asynchronous, nondeterminate systems will find the techniques developed here to be of assistance in building packet systems.

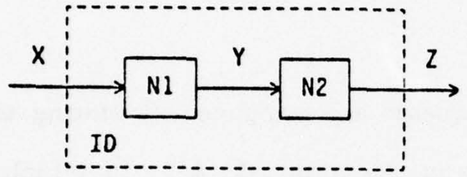
CHAPTER 5: PROVING PACKET SYSTEMS CORRECT

5.1. Setting up a correctness proof

In this chapter we discuss the application of our specification model to the problem of proving packet systems correct. A packet system is an interconnection of component modules, and its behavior is formally given by its internal specifications. Such a system will be correct if it also satisfies a given set of external specifications. Correctness of a packet system, therefore, is the agreement between a set of internal specifications and a set of external specifications.

To prove correctness of a particular system SYS, one must show that its external characteristic relation $\mathrm{EXT}_{\mathrm{SYS}}$ and its internal characteristic relation $\mathrm{INT}_{\mathrm{SYS}}$ coincide. The proof naturally separates into two parts, namely the inclusions $\mathrm{INT}_{\mathrm{SYS}} \subseteq \mathrm{EXT}_{\mathrm{SYS}}$ and $\mathrm{EXT}_{\mathrm{SYS}} \subseteq \mathrm{INT}_{\mathrm{SYS}}$. The first inclusion states that all system responses to given input satisfy the external specifications. This will be proved by showing that for *any* complete execution sequence for SYS, the initial input slice and the ultimate (limit state) output slice are related by $\mathrm{EXT}_{\mathrm{SYS}}$. We call this the *consistency* portion of the proof, since it verifies that complete execution sequences model the behavior intended for the system. The other inclusion states that all behavior allowed by the external specifications may be realized by *some* complete execution sequence. This is called the *synthesis* portion of the proof, since it involves construction of an appropriate execution sequence to realize each instance of system behavior.

The simplest example we can give of a correctness proof is for a system ID composed from two copies N1 and N2 of the negation module N described in section 3.3. The structure of this system is shown in figure 5.1-1.



```
System ID
    inputs X(boolean)
    outputs Z(boolean)
    internals Y(boolean)
Submodules
    N1 inputs X; outputs Y
    N2 inputs Y; outputs Z
Initially empty
```

Figure 5.1-1: A simple system ID to be proved correct

The behavior of ID is trivial: any boolean packet value coming in on channel X is twice negated, thus remaining unchanged. Since both N1 and N2 preserve stream ordering and since the channels are all FIFO, the system ID sends out on Z the identical stream received on X. So to demonstrate the correctness of ID, we will have to show that its internal characteristic relation INT_{ID} matches the external characteristic relation $EXT_{ID} \subseteq ((X^*) \times (Z^*))$ given by

$$((x), (z)) \in EXT_{ID} \iff z = x.$$

For the component modules N1 and N2, the external characteristic relations $EXT_{N1} \subseteq ((X^*) \times (Y^*))$ and $EXT_{N2} \subseteq ((Y^*) \times (Z^*))$ are given by

$$((x), (y)) \in EXT_{N1} \iff \#y = \#x \text{ and } y[i] = \text{not}(x[i]) \ \forall i \leq \#y$$

and $$((y), (z)) \in EXT_{N2} \iff \#z = \#y \text{ and } z[i] = \text{not}(y[i]) \ \forall i \leq \#z.$$

Note that all three channel spaces X, Y and Z are equal to the set {true, false} of boolean values.

We can formally state the correctness theorem for the given realization of system ID. The definition of the relation INT_{ID} is incorporated into the following statement.

Theorem: $((x), (z)) \in EXT_{ID} \Leftrightarrow ((x), (z)) \in INT_{ID}$
$\Leftrightarrow \exists$ a complete execution sequence $\{\langle \$_i, \$_i^a \rangle\}$ for ID such that $x_0 = x$ and $z_\infty = z$, where x_0 and z_∞ are defined by $\$_0 = (x_0, y_0, z_0)$ and $\$_\infty = \sup_{PREFIX} \{\$_i^a\} = (x_\infty, y_\infty, z_\infty)$.

We recall the definitions of execution sequence and completeness, stating them for our particular system ID: A sequence of the form $\{\langle \$_i, \$_i^a \rangle\}$ in which for each i $\$_i^a = (x_i^a, y_i^a, z_i^a)$ is a prefix of $\$_i = (x_i, y_i, z_i)$ will be an execution sequence for ID if and only if the following five conditions hold:

(1) *[initial state]* $\$_0^a = (\epsilon,\epsilon,\epsilon)$, $y_0 = z_0 = \epsilon$
(2) *[input suspension]* $x_i = x_0$ for all i
(3) *[consistency]* $((x_i^a), (y_i)) \in EXT_{N1}$ and $((y_i^a), (z_i)) \in EXT_{N2}$ for all i
(4) *[FIFO]* $\langle \$_i, \$_i^a \rangle$ PRECEDES $\langle \$_{i+1}, \$_{i+1}^a \rangle$ for all i
(5) *[connection]* $\$_{i+1}^a$ PREFIX $\$_i$ for all i

An execution sequence $\{\langle \$_i, \$_i^a \rangle\}$ for ID is complete if and only if $\forall i \exists j$ *s.t.* $\$_i$ SUBSEQ $\$_j^a$. Note that whenever this is true, the Limit Existence Theorem guarantees that we will also have $\sup_{PRECEDES} \{\langle \$_i, \$_i^a \rangle\} = \langle \$_\infty^a, \$_\infty^a \rangle$, where $\$_\infty = \sup_{PREFIX} \{\$_i^a\}$.

The statement of the correctness theorem for the system ID is now complete, and we are ready to begin developing a proof.

5.2. Proof for the system ID

We must show that for the system ID the external relation $\mathrm{EXT}_{\mathrm{ID}}$ and the internal relation $\mathrm{INT}_{\mathrm{ID}}$ coincide. The consistency portion of the proof involves showing that $\mathrm{INT}_{\mathrm{ID}} \subseteq \mathrm{EXT}_{\mathrm{ID}}$, which means that for any complete execution sequence for ID the initial input x_0 and the ultimate output z_∞ satisfy the characteristic relation $\mathrm{EXT}_{\mathrm{ID}}$. In proving this, we need to establish a particular property that will be an important ingredient in all our correctness proofs. This property, which we shall call the *Limit Size Lemma*, concerns the size of channel sequences in a limit state for a system. Essentially, it asserts that the size of each channel stream in the limit state of an execution sequence is the limit of the sizes of the streams for that channel as one proceeds through the states in the execution sequence. Note that this property is not limited to the particular system ID, but rather holds for any system we will wish to prove correct. The Limit Size Lemma is proved by using the least upper bound property of the limit state to establish the least upper bound property for the sequence sizes.

Lemma: In a complete monotone sequence $\{\langle z_i, z_i^a\rangle\}$ of channel states for a packet system, if $z_\infty = \sup_{\mathrm{PREFIX}} \{z_i^a\}$, then $\#z_\infty = \sup \{\#z_i\} = \sup \{\#z_i^a\}$.

Proof: The sequence $\{\#z_i^a\}$ is a nondecreasing sequence of natural numbers and must either be eventually constant or else increase without bound. In the first case, there exists a j such that $\forall k > j$: $\#z_k^a = \#z_j^a$, which implies $\#z_j^a = \sup \{\#z_i^a\}$. Now for any $k > j$, the combination $\#z_k^a = \#z_j^a$ and z_j^a PREFIX z_k^a forces $z_j^a = z_k^a$. Thus $z_\infty = \sup_{\mathrm{PREFIX}} \{z_i^a\} = z_j^a$ and $\#z_\infty = \#z_j^a = \sup \{\#z_i^a\}$.

In the second case, $\sup \{\#z_i^a\} = \infty$. We claim $\#z_\infty = \infty$. If this is false, then $\exists N$: $\#z_\infty = N$. But then ($\forall i$: z_i^a PREFIX z_∞) implies ($\forall i$: $\#z_i^a \leq \#z_\infty = N$), which would make N an upper bound for $\{\#z_i^a\}$, contradicting $\sup\{\#z_i^a\} = \infty$. Thus $\#z_\infty = \infty = \sup\{\#z_i^a\}$.

Now by the Limit Existence Theorem, we have $\langle z_\omega, z_\omega\rangle = \sup_{\text{PRECEDES}}\{\langle z_i, z_i^a\rangle\}$, which implies $\forall i$: $\langle z_i, z_i^a\rangle$ PRECEDES $\langle z_\infty, z_\infty\rangle$. In particular, $\forall i$: z_i SUBSEQ z_∞, so $\forall i$: $\#z_i \le \#z_\infty$, which makes $\#z_\infty$ an upper bound for $\{\#z_i\}$. But $\forall i$: z_i^a PREFIX z_i implies $\forall i$: $\#z_i^a \le \#z_i$, so any upper bound for $\{\#z_i\}$ must be an upper bound for $\{\#z_i^a\}$ and must therefore be no less than the *least* upper bound $\#z_\infty$. This makes $\#z_\infty$ a least upper bound for $\{\#z_i\}$ as well as for $\{\#z_i^a\}$, which completes the proof.

Corollary: If $k<\infty$ and $k \le \#z_\infty$, then there exists an *i* such that $\#z_i^a \ge k$.

Proof: Suppose that for all *i* we had $\#z_i^a < k$. This would imply $\forall i$: $\#z_i^a \le k-1$, which makes $k-1$ an upper bound for $\{\#z_i^a\}$. But by the Limit Size Lemma, $\#z_\infty$ is the least upper bound, so we must have $\#z_\infty \le k-1 < k$, which contradicts the hypothesis for finite *k*.

Now that we have proved this lemma, the consistency proof for system ID is easy. We shall use the abbreviation *LSL* in this proof and all others to denote use of the Limit Size Lemma.

Consistency proof: If we are given a complete execution sequence as in the statement of the correctness theorem for ID, we must show that if $x = x_0$ and $z = z_\infty$, then $((x), (z)) \in \text{EXT}_{\text{ID}}$. This is true if and only if $z = x$, *i.e.*

$$\#z = \#x \text{ and } z[i] = x[i]\ \forall i \le \#z,$$

so we must verify both a size property and an element property of z_∞.

We first note that by the input suspension property of an execution sequence, $x_i = x_0 = x$ for all *i*, so we must also have $x_\infty = x$. In particular, $\#x_\infty = \#x$. But then we have

$$\begin{aligned}
\#z_\infty &= \sup\{\#z_i\} && (\textit{LSL})\\
&= \sup\{\#y_i^a\} && (\text{by } \text{EXT}_{\text{N2}})\\
&= \#y_\infty && (\textit{LSL})\\
&= \sup\{\#y_i\} && (\textit{LSL})\\
&= \sup\{\#x_i^a\} && (\text{by } \text{EXT}_{\text{N1}})\\
&= \#x_\infty && (\textit{LSL})\\
&= \#x,
\end{aligned}$$

which establishes the desired size property.

The element condition is equally easy. For any natural number $k \leq \#z$, by the corollary to *LSL* we have $\exists i: \#z_i^a \geq k$. Now we have

$$\begin{aligned}
z[k] &= z_\infty[k] \\
&= z_i^a[k] \quad \text{(since } z_i^a \text{ PREFIX } z_\infty) \\
&= z_i[k] \quad \text{(since } z_i^a \text{ PREFIX } z_i) \\
&= \text{not}(y_i^a[k]) \quad \text{(by } EXT_{N2}) \\
&= \text{not}(y_i[k]) \quad \text{(since } y_i^a \text{ PREFIX } y_i) \\
&= \text{not}(\text{not}(x_i^a[k])) \quad \text{(by } EXT_{N1}) \\
&= \text{not}(\text{not}(x[k])) \quad \text{(since } x_i^a \text{ PREFIX } x) \\
&= x[k].
\end{aligned}$$

This is the required element condition, and the consistency portion of the proof is now complete.

The above consistency proof may appear to be relatively intricate for such a trivial system as ID, but it really isn't. All we really had to do was set up two simple chains of equality that traced the internal data paths and applied the behavioral properties of the component modules. For noncyclic systems, this presents no real difficulties.

The synthesis portion of the correctness proof for ID involves showing that $EXT_{ID} \subseteq INT_{ID}$. For each given input stream x and each corresponding output stream z, we need to construct an execution sequence for ID to realize the appropriate system behavior. Thus, given streams x and z for which $((x), (z)) \in EXT_{ID}$, we must realize the internal behavior of ID by a matching execution sequence $\$_0,\ldots,\$_j,\ldots$ in which each system state $\$_j$ is a 3-tuple (x_j, y_j, z_j) of dotted channel states. (The dot, as we mentioned earlier, separates the acknowledged prefix from the rest of a channel stream.)

Our strategy is to produce a general order in which the component modules absorb and process packets. The order we choose for these actions in the system ID is as follows: *(1)* Module N1 receives a packet p from channel X and generates its negation not(p) for output on Y; *(2)* Module N2 receives the not(p) packet from Y and generates a packet with value not(not(p)) = p for output on Z; *(3)* The outside world receives and acknowledges the p packet from Z. This sequence of actions is repeated once for each packet in the presented input stream X. Thus the execution sequence we shall generate for the given streams x and z will be cyclic of period three.

Synthesis proof: Given streams x and z for which $((x), (z)) \in EXT_{ID}$, we note that this means z = x. Let $k=\#x$ (note that *k* may be infinite), [illegible] let y be the unique stream of size *k* for which each element is given by $y[i] = not(x[i])$.

For each natural number *i* starting from zero, define

(0) $\$_{3i} = (x[1:i] \cdot x[i+1:k],\ y[1:i] \cdot,\ z[1:i] \cdot)$.

This formula gives every third state in the execution sequence. For i=0, it reduces to the case of the initial system state

$$\$_0 = (\cdot x,\ \cdot,\ \cdot),$$

since the stream segments indexed by the expression [1:i] = [1:0] are all empty.

For each natural number *i* starting from one, define

(1) $\$_{3i-2} = (x[1:i] \cdot x[i+1:k],\ y[1:i-1] \cdot y[i],\ z[1:i-1] \cdot)$ and

(2) $\$_{3i-1} = (x[1:i] \cdot x[i+1:k],\ y[1:i] \cdot,\ z[1:i-1] \cdot z[i])$.

These two formulas give all the system states whose indices are respectively one more and two more than the multiples of three.

Together, the formulas *(0)*, *(1) and (2)* define an infinite sequence of system states $\$_0, \ldots, \$_j, \ldots$ which may be verified in an extremely tedious and extremely straightforward manner to in fact be a complete execution sequence for the system ID. We will not go into the details here, since the remainder of the proof is neither interesting nor illuminating. We shall, however, make some comments about the execution sequence we just constructed.

First, we make some observations about the states. In the *i*-th state given by formula *(1)*, the *i*-th packet x[i] in the input stream x has just been absorbed by module N1, and its negation is seen as a newly-generated (but not yet acknowledged) packet on channel Y, denoted by the "•y[i]". In the corresponding (*i*-th) state given by *(2)*, this packet has been received and acknowledged by N2, and N2 has generated a new packet with value z[i]. This state is followed by the *i*-th state given by *(0)*, which reflects the acknowledgment of the z[i] packet by the outside world.

If the size *k* of the input stream x is finite, then the above sequence of system stated will repeat endlessly after $\$_{3k}$. All states from this point on will be identical, namely

(*) (x•, y•, z•).

In this terminal state, all the input packets have been processed and a complete response has been passed to the outside world. Since the sequence of states is eventually constant in this case, the limit is precisely this repeating terminal state. In the case of an infinite input stream x, the states in the infinite sequence are all distinct, and the terminal state given by *(*)* above is the limit even though it does not actually occur within the sequence. In either case, we note that the output stream z will be identical to the input stream x by the hypothesis ((x), (z)) $\in$ $\mathrm{EXT}_{\mathrm{ID}}$.

The execution sequences produced by the synthesis proof do not exhaust *all* possible sequences for the system ID; however, they are sufficient to realize all legal behaviors for ID given by $\mathrm{EXT}_{\mathrm{ID}}$.