1982

A general purpose State Architecture Simulator for discrete systems with application in data communication protocols

Lap-Kin Ip
Iowa State University

INFORMATION TO USERS

This reproduction was made from a copy of a document sent to us for microfilming. While the most advanced technology has been used to photograph and reproduce this document, the quality of the reproduction is heavily dependent upon the quality of the material submitted.

The following explanation of techniques is provided to help clarify markings or notations which may appear on this reproduction.

1. The sign or "target" for pages apparently lacking from the document photographed is "Missing Page(s)". If it was possible to obtain the missing page(s) or section, they are spliced into the film along with adjacent pages. This may have necessitated cutting through an image and duplicating adjacent pages to assure complete continuity.

2. When an image on the film is obliterated with a round black mark, it is an indication of either blurred copy because of movement during exposure, duplicate copy, or copyrighted materials that should not have been filmed. For blurred pages, a good image of the page can be found in the adjacent frame. If copyrighted materials were deleted, a target note will appear listing the pages in the adjacent frame.

3. When a map, drawing or chart, etc., is part of the material being photographed, a definite method of "sectioning" the material has been followed. It is customary to begin filming at the upper left hand corner of a large sheet and to continue from left to right in equal sections with small overlaps. If necessary, sectioning is continued again—beginning below the first row and continuing on until complete.

4. For illustrations that cannot be satisfactorily reproduced by xerographic means, photographic prints can be purchased at additional cost and inserted into your xerographic copy. These prints are available upon request from the Dissertations Customer Services Department.

5. Some pages in any document may have indistinct print. In all cases the best available copy has been filmed.

University
Microfilms
International
300 N. Zeeb Road
Ann Arbor, MI 48106

Ip, Lap-Kin

A GENERAL PURPOSE STATE ARCHITECTURE SIMULATOR FOR DISCRETE SYSTEMS WITH APPLICATION IN DATA COMMUNICATION PROTOCOLS

Iowa State University PH.D. 1982

PLEASE NOTE:

In all cases this material has been filmed in the best possible way from the available copy. Problems encountered with this document have been identified here with a check mark __√__.

1. Glossy photographs or pages _____
2. Colored illustrations, paper or print _____
3. Photographs with dark background _____
4. Illustrations are poor copy _____
5. Pages with black marks, not original copy _____
6. Print shows through as there is text on both sides of page _____
7. Indistinct, broken or small print on several pages __✓__
8. Print exceeds margin requirements _____
9. Tightly bound copy with print lost in spine _____
10. Computer printout pages with indistinct print _____
11. Page(s) _____ lacking when material received, and not available from school or author.
12. Page(s) _____ seem to be missing in numbering only as text follows.
13. Two pages numbered _____. Text follows.
14. Curling and wrinkled pages _____
15. Other _____

A general purpose State Architecture Simulator for discrete systems with application in data communication protocols

by

Lap-Kin Ip

A Dissertation Submitted to the
Graduate Faculty in Partial Fulfillment of the
Requirements for the Degree of
DOCTOR OF PHILOSOPHY

Department: Electrical Engineering
Major: Electrical Engineering (Computer Engineering)

Approved:

Signature was redacted for privacy.

In Charge of Major Work

Signature was redacted for privacy.

For the Major Department

Signature was redacted for privacy.

For the Graduate College

Iowa State University
Ames, Iowa
1982

TABLE OF CONTENTS

		Page
I.	INTRODUCTION	1
	A. Formal Methods for Protocol Specification	3
	B. Simulation of Protocols	10
	C. Outline of the Dissertation	16
II.	SPECIFICATION OF DISCRETE SYSTEMS USING STATE ARCHITECTURE NOTATION	18
	A. General Features	20
	B. Finite-State Machine	24
	C. Pulsed Combinational Function	32
	D. Static Combinational Function	32
	E. Pulsed Delay	35
	F. Static Delay	35
	G. Queue	36
	H. Derivative	37
	I. Clock	38
	J. Environment	39
III.	STATE ARCHITECTURE SIMULATOR	49
	A. Overview of the State Architecture Simulator	49
	B. Data Structure	57
	C. Transformation	69
	D. Data Input	92

E. Initialization 119

F. System Executive 128

IV. APPLICATION OF THE STATE ARCHITECTURE NOTATION AND STATE ARCHITECTURE SIMULATOR IN SIMULATING DATA COMMUNICATION PROTOCOLS 188

A. Operational Steps in Running SAS 188

B. A Simple Discrete System Simulation 198

C. Start-Stop Link Simulation 230

D. Advanced Data Communication Control Procedures Simulation 257

V. DISCUSSION AND CONCLUSIONS 265

A. SAN and SAS Model Size 265

B. Experience with SAN and SAS 266

C. Reliability of SAS 267

D. Limitations of SAN and SAS 269

E. Conclusions 272

VI. BIBLIOGRAPHY 273

VII. ACKNOWLEDGMENTS 277

VIII. APPENDIX: STATE ARCHITECTURE NOTATION SYNTAX DIAGRAMS 278

I. INTRODUCTION

A data communication protocol is a set of regulations governing the sending and receiving of data between two or more data processing systems such as computers, terminals, and telephone systems. A set of computers, terminals, and other data processing systems which are interconnected to each other is called a computer network.

The SAGE (Semi-Automatic Ground Environment) Air Defense system and SABRE, an on-line airline reservation system, were two of the earliest computer networks. These early networks were one-of-a-kind systems that were developed by experts using unique ad hoc methods to attain the required function.

The data communication protocols (regulations) involved in governing the communication between computer systems in the '60s and early '70s were defined predominantly in natural languages, e.g., Binary Synchronous Communication (BSC), CCITT's X.21 and X.25, and the currently popular HDLC. Validation of the protocol specification (defined in natural languages) was done mainly by hand checking and human intuition. As a result, ambiguities existed in the protocol specification. Errors or differences also existed in implementation. For example, the implementation of the well-known Binary Synchronous Communication protocol in different systems by different implementors ended up with incompatible systems which could not communicate with each other [Sussenguth 1976].

Generally, the natural language form of protocol specification creates ambiguities in the interpretation of the protocol's exact meaning.

Thus, the protocol design cannot be validated and analyzed automatically. Furthermore, it is not possible for the simulation and implementation of the protocol system to be derived directly from the specification.

The cost/performance of both computing systems and data communication facilities improved significantly in the '70s. As a result, a large number of businesses have begun using computer networks to improve their business transactions (banking, electronic mail, air reservation, etc.). The commercial opportunities and the advances in computer technology have led the computer and communication industry to market standard network hardware and software components. This is done so that the user might build a wide variety of application networks from a small number of standard components. Therefore, a number of function and interface standards were developed for these types of components. These standards include IBM's Systems Network Architecture [SNA 1978], Hewlett-Packard's Interface Bus [Hewlett-Packard 1975], Digital Equipment Corporation's DECNET [Digital Equipment 1978], CCITT's X.25 [CCITT 1979] and the IEEE Standard 488 [IEEE 1978]. However, the support tools needed in the design, specification, simulation, validation, implementation, and testing of these kinds of protocols have still been developed largely on an informal basis. During the last ten years, a significant amount of research has been focusing on formal methods to support the design, specification, validation, simulation, and implementation of data communication protocol systems. In the next section, we will examine current formal specification and validation techniques applied in the protocol design process.

A. Formal Methods for Protocol Specification

As understood by people in the data communication protocol area, protocol specifications must ultimately serve several purposes including definition, verification, simulation, analysis, implementation, and documentation of the algorithms involved. Definition means a complete and unambiguous written specification using a reference model. Verification means the analysis and reliable prediction of the behavior of the system design during the design stage (from the protocol specification) and prior to implementation. The verification may be manual or automated. Simulation means that a direct simulation of the protocol system can be derived from the protocol specification. The simulation should totally reflect the function of the design as denoted by the specification. Analysis means that the specification should provide an analytic basis for predicting implementation performance characteristics such as throughputs, queue sizes, response times, and delays. Implementation means that the specification is a basis for direct implementation, e.g., if the protocol is specified in a programming language it can be directly compiled into a running implementation.

In the past decade, many different formal methods have been proposed for protocol specification. A good collection of the papers related to this area can be found in the books (1) Communication Protocol Modelling edited by Sunshine [Sunshine 1981], (2) Computer Network Architectures and Protocols edited by Green [Green 1982] and (3) the Proceedings from both the First and Second International Workshop on

Protocol Specification, Testing, and Verification [Proceeding of the First International Workshop 1981] and [Proceeding of the Second International Workshop 1982].

In general, there are two main paths in the modelling (specification) and verification of protocols [Sunshine 1978]. The first path is based on algorithmic description of a protocol system followed by assertion proofs and temporal logic. This path mainly uses some kind of programming language to describe protocol systems [Bochmann 1975], [Stenning 1976] and [Hailpern and Owicki 1980]. The second path is based on some form of state and transition description of a protocol system followed by a reachability test. This category includes the UCLA graph [Postel and Farber 1976], Petri nets [Merlin and Farber 1976] and [Symons 1980a], formal language [Teng and Liu 1978] and [Harangozo 1977], and finite-state machines [Piatkowski 1975], [Sundstrom 1977], [Bochmann 1978] and [Piatkowski 1979].

1. Programming language model

The general approach of programming languages is to describe protocol behavior by writing a program or algorithm in some programming language. Programming languages are well-known and there is little in the specifications themselves to arouse interest.

This approach is straightforward. It is easily understood by those who have a programming language background. Assertion methods are usually used to test certain invariants existing in the protocol.

Recently temporal logic has been employed to validate protocols specified in programming languages [Hailpern and Owicki 1980].

On the other hand, programming language approaches tend to include implementation details which are overly constraining. The sequential nature of programming languages hides the concurrent nature of the protocol. Furthermore, the state of the protocol system is hidden in the value of the program variables and the stopping places within each program, which makes the validation process more difficult.

2. State transition model

The general approach of state transition models is to describe the input and output sequence of a protocol system together with an internal system state variable, which identifies the status of the current input history. Even though many of the concepts underlying the various state transition models (UCLA graph, Petri net and a finite-state machine approaches) are similar, the notations used for each method is quite different.

UCLA graphs and Petri nets notation are similar. Petri net models are represented by a directed graph with nodes and transition bars. The holding of a condition of a protocol system is represented by placing a token in the node. If all the nodes (conditions) input to a transition bar (event) have tokens, then a transition bar (event) can fire (occur). In this approach, the condition represents both the state and the input of a protocol system, and the event represents the state transition and the generation of new outputs of a protocol system. Diaz [Diaz 1982]

gives a summary of the evolution of a simple Petri net to different sophisticated Petri nets such as Place-colored net, Predicate-transition net, Predicate-action net, Numerical Petri net and Timed net and their applications in modelling different protocols.

Finite state automata theory was one of the earliest techniques used to model protocols. A finite state automaton or finite-state machine is a system consisting of five well-defined entities S, X, Z, FNS, FOUT

where S is a finite set of states;

X is a finite set of inputs;

Z is a finite set of outputs;

FNS is the next stat function;

FOUTS is the output function.

As mentioned by Sunshine [Sunshine 1981], the notion of a machine which reacts to inputs and produces appropriate outputs is a natural and intuitive way to view the functioning of a protocol machine. The inputs to a protocol machine represent commands from a local user, messages from a remote protocol machine, or internally generated events such as timeouts. The protocol machine changes its state and generates outputs appropriately after each input so that subsequent inputs will be processed correctly. The outputs of a protocol machine represent messages sent to the local users and remote protocol machines.

Early works in using the finite-state machine transition diagram to model protocol can be found in papers by Bjorner [Bjorner 1970] and Kawashima et al. [Kawashima et al. 1971]. In these early works, a single

finite-state machine transition diagram is used to represent the global states of the protocol system. However, for a protocol of any complexity, the number of states and transitions becomes unworkably large. To solve this problem, the idea of coupled-machines was introduced, in which the protocol system is subsequently decomposed into an interconnection of smaller independent finite-state machines. Models of this category can be found in papers by Piatkowski [Piatkowski 1975], Bochmann [Bochmann 1978], Gouda and Manning [Gouda and Manning 1976] and IBM's SNA Format and Protocol Reference Manual [SNA 1976].

When a protocol using a sequence number is modelled, such as the data transfer phase of the CCITT X.25, there must be different states and transitions to handle each possible sequence number. In this case, the states of the finite-state machine can become very large depending on the range of the sequence number to be modelled.

To solve this problem, researchers attempt to combine the advantages of state transition models and program languages to develop a hybrid model for protocol representation. The hybrid model usually employs a state transition model to capture the control state of a protocol such as the call establishment phase of X.25 and uses the program variables and algorithms in each major state to process incoming messages with numerical value. A typical hybrid model is the HDLC model developed by Bochmann and Chung [Bochmann and Chung 1977]. Other hybrid models can be found in papers by Danthine and Bremer [Danthine and Bremer 1978], Schultz et al. [Schultz et al. 1980] and Alfonzetti et al. [Alfonzetti et al. 1979].

3. Limitations

As for the above mentioned state transition techniques, Petri nets and finite-state machines are both widely used by different researchers. However, most of the models developed today are geared toward validation, especially the various Petri net approaches. Petri nets can be used to validate the correctness of certain kinds of communication protocol through some algorithms and assertions [Symons 1980b]. The models so far developed using Petri nets seem very large and hard to understand [Diaz 1982]; this seems to indicate that Petri nets are not well-suited as a specification tool. One reason is that a Petri net does not reveal explicitly the global state of the system, rather its markings do.

Even when finite-state machines are used, most of the models do not have detailed information concerning interconnections and valid input/output values to provide a guideline (documentation) for direct simulation and implementation.

Programming languages do meet the need as a document for direct simulation and implementation. However, the sequential nature of program execution, which is over constraining, is undesirable. Concurrent programming languages and programming languages with monitor capability may eliminate the sequential constraint, but such kinds of models are not in the current literature.

In the next section, we will propose a formal specification language for protocol systems. The language will preserve the

concurrent nature of a protocol system and at the same time can be used as a specification for direct simulation and implementation.

4. Overview of the SAN specification

As was mentioned, the objectives of formal methods in protocol specification include definition, verification, simulation, analysis, implementation and documentation. The SAN specification described in this thesis meets all of the above objectives as discussed below.

The SAN specification approach was begun by Piatkowski [Piatkowski 1975] while he was at work in IBM where he employed the state architecture oriented method to specify the function and architecture of IBM's Systems Network Architecture (SNA). The SAN mentioned in this thesis is an extension of the original SAN.

The SAN approach can be regarded as a hybrid method. It employs classical finite-state machines and combinational functions to capture the global states of the protocol system. Since some of the aspects of a protocol system's behavior cannot be modelled by finite-state machines, a number of additional primitive component types such as the delay, FIFO queue, derivative and clock were introduced in SAN. Also a special component type, environment, was introduced in SAN to allow an arbitrary PASCAL procedure to define the function of a component. These primitive component types have been shown to be useful in building protocol system models.

As far as the six protocol specification objectives are concerned, the precise definition of all components' input and output values, their

interconnections, and their transitions provide a complete and unambiguous written specification for the protocol. Even though computer tools for validation and analysis of SAN specifications have not been developed, we conjecture that the common verification methods such as reachability testing, assertion and temporal logic proofs can be used on SAN specifications. All of the above methods were related to the testing of the system state variables either by checking the invariants of the state variables or exhaustly driving the system to land in all possible values of the state variables.

A SAN model can also be used as a direct simulation model. In other words, a direct simulation can be automatically derived from the SAN model. Additionally, a partial or maybe fully automatic implementor could be built to generate different implementations of the SAN specification. Finally, the SAN specification augmented with natural language descriptions can be used as a document for a protocol system. The SAN specification is used for accurate and precise communication among many groups of people (designers, implementors and maintainers, etc.). The natural language description helps to explain the difficult concepts of the protocol, which may be difficult to understand in reading the SAN specification.

B. Simulation of Protocols

One of the objectives of protocol specification is to derive a simulation of the protocol system directly from the protocol

specification. This dissertation describes an investigation of the State Architecture Notation (SAN) and the State Architecture Simulator (SAS), a simulator driven by the SAN specification.

Though simulation cannot be used to test out a design completely, it is a good tool for a designer to check out the behavior of a design. This informal method allows a designer to observe the exchange of input and output sequences of the simulated system, which is very valuable in informal design validation. The ability of the simulator to simulate different levels of abstraction of protocol systems allows designers to check out the design of subsystems. A simulation model can also be used to predict the performance of protocol systems in different environments. If a simulation model exactly reflects the design of a protocol system written in some formal specification (like SAN), the simulation model can help the user to interpret the meaning of the formal specification by exercising the simulation model. In other words, the formal specification together with the simulation model can be used as a reference model for the protocol system.

The general approach to building a protocol simulation model in a computer is first to analyze and understand the concept and design of the protocol system, second to formulate a conceptual model according to the level of abstraction desired and third to translate the conceptual model to some computer executable language representation. Two different types of computer languages have been widely used by researchers.

First is the use of general purpose programming languages such as FORTRAN, PASCAL, PL/1, ALGOL, etc. For example, IBM uses PL/1 as a base

language to represent the SNA protocol [SNA 1978] and also uses the PL/1 language representation as a simulation tool to verify protocol designs. Yeh [Yeh 1979] demonstrates the "systematic model construction /validation and incremental modelling" approach by simulating HyperNet via a process oriented language called ASPOL.

Second is the use of general purpose simulation languages such as SIMSCRIPT II, SIMULA, GPSS, GASPIV, etc. For example, Remes [Remes 1978] developed a GPSS model for simulation of multidrop lines. Also, many business institutions have developed their own network simulation packages, for instance, NCR Comten's Link Evaluation Model and Discrete Simulation Model which is intended for simulation studies of networks of Comten's products (implemented in FORTRAN).

In general, using general purpose programming languages or simulation languages to simulate protocol systems suffers from two weak points. Firstly, the translation from prose specification or even formal specification to the specified language may introduce errors. Secondly, it takes a lot of time and effort to code from the specification into the target language. As for the network simulation packages developed in business institutions, most of them are too restricted to their specific applications. Further, most of the simulation packages are not available to the public.

In order to relieve the designer's burden to translate the protocol model (specification) to a specific language to form a simulated system and at the same time to enjoy the benefits of using simulation to facilitate the design process, we propose a formal language for the

representation of protocols, namely the State Architecture Notation (SAN). We have developed a general purpose simulator, namely the State Architecture Simulator (SAS), to execute system models represented in SAN.

Different formal methods for protocol specification and their pros and cons have been discussed in the previous section. Some of the formal specifications have a simulator to execute the system models represented in those specifications. The SARA GMB simulator (System Architect's Apprentice Graph Model of Behavior simulator) [Gardner 1977] and [Razouk and Estrin 1977] developed at UCLA and IBM's SNA written in the IBM SNA Format and Protocol Language (FAPL) are examples. Furthermore, the CCITT Specification and Description Language (SDL), which is still under study, is also aimed to provide a formal executable specification language for communication protocols [Rockstrom and Saracco 1982].

The SARA GMB simulator together with GMB translator and PL/1 preprocessor (PLIP) provides the user the capability to execute UCLA graph models and to examine or modify the state of the control and data graphs of the model during a simulation. From the machine readable specification of the control and data graph models, the GMB translator creates internal data representation of the control and data graphs. The PL/1 preprocessor takes the PL/1-like code in the processor and generates the corresponding PL/1 code properly interfaced to the simulator. The simulator executes the control graph, which models the different synchronization and mutual exclusion of processes. The

simulator provides the designer the capability to conduct interactive experiments on behavioral models during the design process.

The SARA GMB simulator together with some analytical tools provides a good base to use in designing and testing out a protocol model. However, as noted by Postel and Farber [Postel and Farber 1976], the graph becomes fairly complicated for any practical protocol.

As for IBM's SNA, it was first proposed by Piatkowski [Piatkowski 1975] to represent the SNA protocol modules by interconnections of finite-state machines and combinational functions. The function of each finite-state machine and combinational function is realized in a combination of graphical, matrix and tabular representation as appeared in the first edition of the IBM's SNA Format and Protocol Reference Manual [SNA 1976]. In the Second edition of the IBM's SNA Format and Protocol Reference Manual [SNA 1978], most of the matrices and tables were translated into PL/1 like procedures and some other FAPL statements. A translator was built within IBM to translate the FAPL statements into PL/1 statements, so that the SNA specification can be executed. The executable representation of the SNA data flow control layer has been tested by an automated protocol validation technique [Schultz et al. 1980].

The State Architecture Simulator (SAS), discussed in this dissertation, is built to execute system models represented in the SAN specification language. The reason for choosing, SAN language was explained in the previous section.

One of the purposes of building SAS is to provide a teaching tool in formal protocol design at Iowa State University. Students will be

able to conduct interactive experiments on protocol designs based on the SAN language. The SAS, will have the following properties:

1) It is an interactive system. Users can assign inputs and observe outputs at their terminals. Users can insert break points in the simulated system so that they can examine the system status and assign new inputs at the break point.
2) Programmable system input modules, both algorithmic and random, can be defined in the simulator to allow users to do both algorithmic and random testing for a long period of time.
3) It has the capability of data collection by tracing the system variables during simulation.
4) It provides some system logic checking during simulation; for instance, it checks 1) if there is any input change of a component, while the component is busy processing a previous input; 2) if the input and output values of a component are elements of the predefined input and output sets, etc.
5) It has a restart initialization process to allow users to continue a simulated system execution from the state where it stopped in a previous run.

The above properties will make SAS a useful tool in demonstrating and validating protocol systems during their design stages.

C. Outline of the Dissertation

This dissertation is divided into five main chapters. The first chapter is the introduction.

The second chapter presents the syntax and semantics of the machine readable form of the State Architecture Notation (SAN), a language for specifying models of protocol systems. There is a compatible graphical version [Piatkowski 1981]. Protocol systems are modelled by specifying an interconnection of the nine basic components defined in SAN: finite-state machine (FSM), pulsed combinational function (CFP), static combinational function (CFS), pulsed delay (DELP), static delay (DELS), queue (QUE), derivative (DER), clock (CLK), and environment (ENV).

The SAN language discussed in this dissertation was first proposed by Professor Piatkowski and was refined by us in the process of developing a State Architecture Simulator to execute the protocol models specified in SAN language.

The third chapter presents a detailed description of the design and the implementation of the State Architecture Simulator (SAS) which has been developed over the last three years by the author. SAS contains two PASCAL programs and a VAX command procedure that compiles, executes and reports on simulations of user supplied SAN specification of protocol systems. It was developed and is running on a VAX 11/780 at Iowa State University using the VAX VMS V2.0 operating system. SAS accepts SAN specifications of protocol systems as inputs and executes the simulated system. SAS is an interactive system allowing users to

examine the system status, to assign inputs and save system status at a terminal.

The fourth chapter presents the application of SAN and SAS to simulate some specific data communication protocol examples. The operational steps involved in creating and running an SAS executable simulated system from the SAN model are first described and then followed with a simple example to demonstrate the above steps. A second example demonstrates the use of all the different kinds of basic components and the trace utilities via a simulation of a Start-Stop link. A third example illustrates the use of SAN and SAS in simulating a fairly complicated data communication protocol, namely Advanced Data Communication Control Procedures (ADCCP). The ADCCP SAN model and the SAS runs were developed over the past year at Iowa State University by Mr. Dayun He, Visiting Scholar from the Research Institute of Posts and Telecommunications of the People's Republic of China.

The fifth chapter evaluates the pros and cons of SAN and SAS and discusses some of the possible ways that SAN and SAS can be improved to facilitate the simulation process. Finally, the contributions of the SAN and the SAS in the protocol design process are presented.

II. SPECIFICATION OF DISCRETE SYSTEMS USING STATE ARCHITECTURE NOTATION

In this chapter, we present the machine readable form of the State Architecture Notation (SAN). Users interested in a more detailed treatment of the conceptual basis for SAN or in the details of the graphical version may refer to papers by Piatkowski [Piatkowski 1975] and [Piatkowski 1981]. Part of the materials appearing in this chapter have been presented in a paper by Piatkowski, Ip and He [Piatkowski et al. 1982].

The purpose of the SAN is to allow users to represent discrete systems in a formal and machine readable specification. The basic approach in defining a complicated discrete system is to decompose the system, through a series of successive refinements, into a well-defined interconnection of smaller systems called components. After the desired level of system decomposition has been structurally described, the behaviors of the components are defined independently.

Each component in a SAN system model has a unique name and a set of pulsed and static input and output streams

XP.1,XP.2,...;XS.1,XS.2,...;ZP.1,ZP.2,,...ZS.1,ZS.2,...

as shown in Figure 2.1.

We can reference a particular input or output of any specific component using PASCAL-like qualification (e.g., name.cxp.3).

Each component operates in asynchronous discrete time with the pulsed variables being defined only at discrete times and the static variables changing only at discrete times.

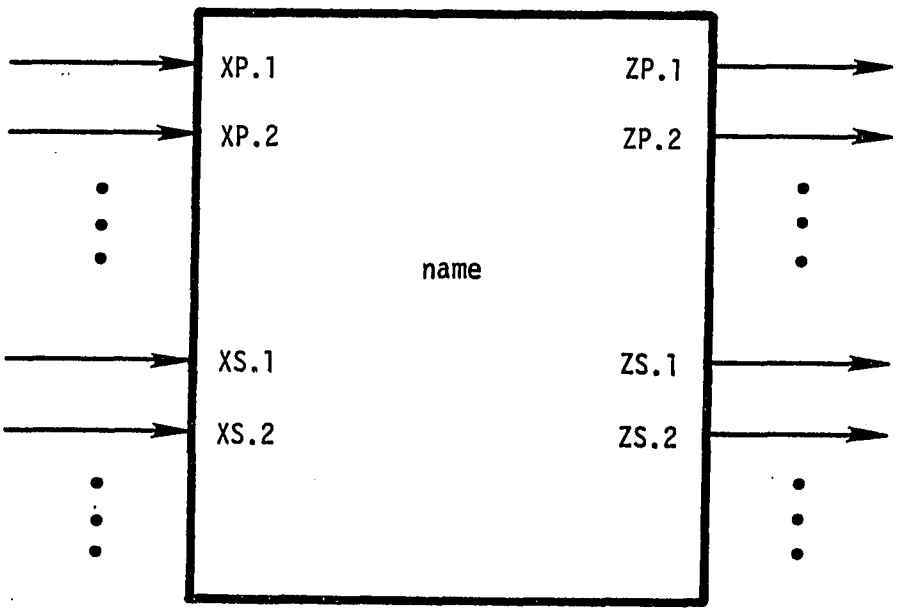


Figure 2.1. Overview of the input/output structure of a general component

A general SAN system model is built up as an interconnection of specific instances of nine types of basic components, namely:

finite-state machine (FSM)

pulsed combinational function (CFP)

static combinational function (CFS)

pulsed delay (DELP)

static delay (DELS)

queue (QUE)

derivative (DER)

clock (CLK)

environment (ENV).

A. General Features

Before we describe the detailed specification of each kind of component, we want to describe the special symbols, the variables, and the general features used in the SAN specification.

As an aid to clear exposition, all occurrences of special symbols in SAN will be underlined in this chapter. In the actual use of SAN, the underlining is not used. Also, the special symbols are confined to either upper case or lower case as they appear. A list of the special symbols used in SAN is shown in Figure 2.2. Readers may find it helpful when reading this section to glance ahead to the examples in Chapter II.B and Chapter IV.

In the SAN specification, the following variable values are treated as arbitrary alphanumeric strings: name, state, input, output, cs, nts,

Symbols	Meanings
CFP	pulsed combinational function
CFS	static combinational function
CHECKOPT	denotes when the traces of variables are to be recorded
CLK	clock
closedempty	a control state of a queue
closednotempty	a control state of a queue
CONDITION	a PASCAL boolean expression denoting when the traces of variables will be recorded
cs	current state of a local component
cxp.i	i th current pulsed input of a local component
cxs.i	i th current static input of a local component
defsinit	default initial state
deftclk	default timeout period of a clock
deftdel	default delay time
deftdeq	default execution time to dequeue data from a queue
deftenq	default execution time to store data in a queue
deftexec	default execution time
defzsinit	default initial values of static outputs
DELP	pulsed delay
DELS	static delay
deq	dequeue control pulse
DER	derivative
DTHISTORY	time interval at which traces of variables are recorded
ENV	environment
everyevent	the STARTEXP will be evaluated after every simulated event
everytimechange	the STARTEXP will be evaluated whenever the current event set in the event file is empty
EXPHISTORY	instance for tracing the value of a boolean expression
expired	a control state of a clock

Figure 2.2. Lists of special symbols in SAN

EXPRESSION	a boolean expression
FNS	next state function
FOUTP	pulsed output function
FOUTS	static output function
FROM	precedes source name list for an input stream
FSM	finate-state machine
FUNCTION	delimits the beginning of the specification of the function of an ENV
INIT	indicate the beginning of an initialization instance
interlaced	denotes that sources and destinations of all inputs and outputs are given within the component specification
list	list option
mulpulsecheck	indicates whether multiple simultaneous pulsed inputs to single components are trapped or not
never	the STARTEXP will never be evaluated
open	a control state of a queue
procedure	procedure option
regular	attribute of a trace; denotes that variables are recorded at regular time intervals
reset	a control state of a clock; a control input of a clock
running	a control state of a clock
S	state set
start	a control state of a clock
STARTEXP	a PASCAL boolean expression denoting when the ENV will be scheduled
STARTEXPCHECK	denotes when the STARTEXP of an ENV is to be evaluated
tbeg	simulation beginning time
tend	simulation ending time
terminal	terminal option
timeout	a control output from a clock
TO	precedes destination name list for an output stream
VARHISTORY	instance for tracing variables
VARIABLES	introduces set of variables to be traced

Figure 2.2. (continued)

XP	pulsed input set
XS	static input set
ZP	pulsed output set
ZS	static output set
#	don'tcare
-	the null or empty pulsed signal
=>	an implicator separating cs, xp's, and xs's from nts, ntzp's and ntzs's
/	a right slash separating cs, xp's and xs's
.	period, a SAN delimiter
:	colon, a SAN delimiter
;	semi-colon, a SAN delimiter
,	comma, a SAN delimiter

Figure 2.2. (continued)

xp, xs, zs, ntzp, and ntzs. Upper and lower case characters can be used for the strings; however, they are interpreted as different characters. In any case, the strings cannot be longer than forty characters.

The following dummy variables used in this chapter must be integers: i, j, k, l, m, n, q, s, u.

The following variables must be a non-negative real or integer: Texec, Tdel, Tenq, Tdeq, Tclk.

As far as the order of specifying each kind of component is concerned, SAN allows any sequence in placing the components. However, the specification of the connections and functions and other parameters within each component are confined to a certain sequence and format. The detailed syntax diagram of each component in SAN is shown in the Appendix. Throughout the use of SAN, any syntax question should be answered by referring to the Appendix.

B. Finite-State Machine

Each instance of a basic component type in a SAN system model is described in structured machine readable form. Figure 2.3 presents the general format for specifying a finite-state machine in SAN; we will look at it first since it is the most complicated and in most systems the most important type of system component to be used.

In the first line of the specification, the FSM component type is declared and the unique component name given. All components in the current version of SAN are interlaced, meaning the sources and

```
FSM name : interlaced

   S : state, state,... ;

   defsinit : state;

   XP.1 FROM name.ZP.i : input, input,... ;
   XP.2 FROM name.ZP.j ; input, input,... ;
    .
    .

   XS.1 FROM name.ZS.k : input, input,...;
   XS.2 FROM name.ZS.l : input, input,...;
    .
    .

   ZP.1 TO name.XP.m, name.XP.n,... : output, output,... ;
   ZP.2 TO name.XP.p, name.XP.q,... : output, output,... ;
    .
    .

   ZS.1 TO name.XS.r, name.XS.s,... : output, output,... ;
   ZS.2 TO name.XS.t, name.XS.u,... : output, output,... ;
    .
    .

   FNS : list

      cs / xp.1, xp.2,... / xs.1, xs.2,...  => nts ;
      cs / xp.1, xp.2,... / xs.1, xs.2,...  => nts ;
       .
       .
   END;

   FOUTP : list

      cs / xp.1, xp.2,... / xs.1, xs.2,...  => ntzp.1, ntzp.2,... ;
      cs / xp.1, xp.2,... / xs.1, xs.2,...  => ntzp.1, ntzp.2,... ;
       .
       .
   END;

   FOUTS : list

      cs  =>  ntzs.1, ntzs.2,... ;
      cs  =>  ntzs.1, ntzs.2,... ;
       .
       .
       .

   END;

   deftexec : Texec;

END;
```

Figure 2.3. Finite-state machine specification format

destinations of all input and output variables are given within the component specification.

Next we enumerate the finite state set, S, and specify the default initial state, defsinit.

Following this, we list in order for each pulsed and static input and output stream the respective source and destinations and an enumeration of stream variable values.

For both pulsed and static input, users first specify the local input (XP.n or XS.n), then follow with FROM, the external source name (name.ZP.m or name.ZS.m), and the valid set of input values. Their formats are as follows:

XP.n FROM name.ZP.m : input,---,input;

XS.n FROM name.ZS.m : input,---,input;

For both pulsed and static output, users first specify the local output (ZP.m or ZS.m), then follow with TO, the external destination names (name.XP.n or name.XS.n), and the valid set of output values. The brackets appearing in the following format mean that multiple destinations, with a maximum of twenty, are allowed. Their formats are as follows:

ZP.m TO name.XP.n [,----] : output,---,output;

ZS.m TO name.XS.n [,----] : output,---,output;

If the input or output of a component is not connected to any component, the unconnected input or output is specified as follows:

XP.n UNCONNECTED : input,---,input;

XS.n UNCONNECTED : input,---,input;

ZP.m UNCONNECTED : output,---,output,

ZS.m UNCONNECTED : output,---,output;

Having defined the scope of the state set and the I/O sets, we now specify the next state function (FNS), the pulsed output function (FOUTP) and the static output function (FOUTS). Figure 2.3 illustrates these functions defined using the list format; i.e., the functions are defined by listing specific combinations of current state (cs), pulsed inputs (xp's) and static inputs (xs's) followed by the associated next state (nts), pulsed outputs (ntzp's) or static outputs (ntzs's). The cs, xp's, and xs's are separated by a right slash, '/'. Within each set of inputs, each input element (xp or xs) is separated by a comma and the input elements are arranged in an ascending order with xp.1 or xs.1 appearing first. There is an implicator, '= ', separating cs, xp's and xs's from nts, ntzp's and ntzs's. The number of xp, xs, ntzp and ntzs elements are equal to the number of pulsed inputs, static inputs, pulsed outputs and static outputs of the component.

The general format of a FNS list entry is as follows:

cs/xp.1, xp.2,---,xp.n/xs.1,xs.2,---,xs.m = nts;

The general format of a FOUTP list entry is as follows:

cs/xp.1,xp.2,--- xp.n/xs.1,xs.2,---xs.m = ntzp.1,ntzp.2,---ntzp.i;

The general format of a FOUTS list entry is as follows:

cs = ntzs.1, ntzs.2,----, ntzs.j;

Beside using regular alphanumeric strings in the line entries of FNS, FOUTP, and FOUTS, the following options can be used:

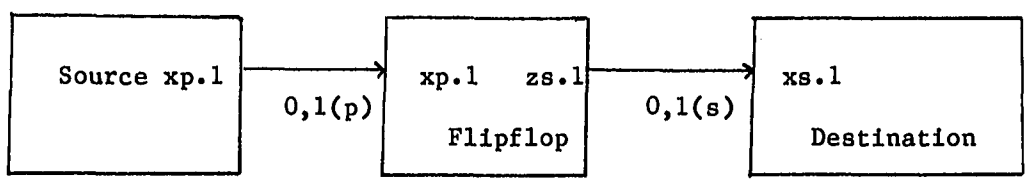
* (don't care), in any cs, xp, xs position

\- (null or empty), in any xp or ntzp position

cs, cxp.i or cxs.i in any nts, ntzp or ntzs position.

In the case where the list format for specifying FNS, FOUTP, or FOUTS is cumbersome, the procedure option may be used. The procedure option when used to define FNS, FOUTP, or FOUTS of a component, except in ENV component, is a restricted PASCAL procedure. The procedure may refer only to the current state and inputs (cs, cxp.n, cxs.m) local to the components, and assign values to the next state and outputs (nts, ntzp.i, ntzs.j) local to the component. The local component name is not used in the prefix of references to these variables. To define a procedure, the word procedure has to be written following the name of a function (FNS, FOUTP or FOUTS) and a colon. In the next lines, a set of statements can be written to represent the procedure functions. The details of the statements° syntax can be seen in the syntax diagrams in the Appendix. An END and a semicolon are appended after the last statement to indicate the end of the procedure.

The following is an example SAN specification of a flip-flop, named Flipflop, using a FSM component. The inputs and outputs of the flip-flop are connected to two fictitious components, namely Source and Destination, as shown in the following block diagram.



The flip-flop has two states, 0 and 1. It has a binary pulsed input and a binary static output. The current state of the flip-flop remembers the last pulsed input, and the current static output is a direct mapping of the current state. The SAN model of Flipflop is shown in Figure 2.4. Notice in the SAN model of Flipflfop, the static input, XS.n, the pulsed output, ZP.m, and the pulsed output function, FOUTP, are missing. Actually, the SAN supports partial specification of different types of components. The detailed syntax for representing each type of component can be seen in the Appendix.

As far as the options used in the line entries of FNS, FOUTP and FOUTS are concerned, the FNS line entries in Figure 2.4 can be improved by using the don't care symbol, '*' and the current state and input variables. The improved FNS is as follows:

```
FNS  :  list
*/* =  cxp.1;
END;
```

The next state (nts) will be equal to the current pulsed input (cxp.1) regardless of the current state and current pulsed input value.

On the other hand, the FNS can be specified via a procedure such as follows:

```
FNS : procedure
    If cxp.1='0' then nts:='0'
    else
    if cxp.1='1' then nts:='1';
END;
```

```
FSM Flipflop : interlaced
  S : 0,1;
  defsinit : 0;
  XP.1 FROM Source.ZP.1 : 0,1;
  ZS.1 TO Destination.XS.1 : 0,1;
  FNS : list
    0/0 => 0;
    0/1 => 1;
    1/0 => 0;
    1/1 => 1;
  END;
  FOUTS : list
    * => cs;
  END;
  deftexec : 0.0;
END;
```

Figure 2.4. SAN model of Flipflop

or an improved procedure specification

```
FNS : procedure
      nts:=cxp.1,
END;
```

The above four different FNS specifications are equivalent to each other. They demonstrate the use of the don't care '*' and current input variables in the list option, and also the use of the procedure option.

In general, a FSM operates as follows:

1) In the absence of a pulsed input, the FSM stays in its current state, producing a constant static output according to FOUTS(cs).
2) When one (or several simultaneous) pulsed input(s) arrive at the FSM, the machine becomes busy for the time interval Texec, prescribed in deftexec, the default time of execution. When the busy interval ends the machine assumes the next state, nts, specified by FNS (cs, xp, xs) and pulses the outputs according to FOUTP (cs, xp, xs). As a consequence of changing the current state, the static outputs of the machine may also change.
3) In interpreting the list representation of a FNS, FOUTP or FOUTS function, the list is scanned from the top to the bottom line entries. The first line entry that matches with the current state and inputs (cs, xp, xs) is used to map into the correspondinig next state and next onputs (nts, ntzp, ntzs); and, the rest of the line entries are ignored. In case none of the line entries matches, there is no pulsed output nor any change in the state or the static outputs of the component.

4) The arrival of a pulsed input at a busy FSM is interpreted as a system execution error.

C. Pulsed Combinational Function

The format for specifying a pulsed combinational function (CFP) in SAN is presented in Figure 2.5. Conceptually, a pulsed combinational function is a finite-state machine with one-state and no static outputs; the state variable is conveniently suppressed in the SAN representation of a CFP. The operation of a CFP is similar to that of a FSM.

D. Static Combinational Function

The format for specifying a static combinational function (CFS) in SAN is presented in Figure 2.6.

A static combinational function accepts only static inputs, as a function of which it produces only static outputs. In response to a changing input, the CFS becomes busy for the time interval specified in deftexec. When the busy interval ends the function changes, its outputs as specified by FOUTS(xs).

The default (initial) values of all static outputs are given in defzs.

The changing of an input at a busy CFS is interpretted as a system execution error.

```
CFP name : interlaced

   XP.1 FROM name.ZP.i : input, input,... ;
   XP.2 FROM name.ZP.j ; input, input,... ;
    .
    .
    .

   XS.1 FROM name.ZS.k : input, input,...;
   XS.2 FROM name.ZS.1 : input, input,...;
    .
    .
    .

   ZP.1 TO name.XP.m, name.XP.n,... : output, output,... ;
   ZP.2 TO name.XP.p, name.XP.q,... : output, output,... ;
    .
    .
    .

   FOUTP : list

       xp.1, xp.2,... / xs.1, xs.2,...  => ntzp.1, ntzp.2,... ;
       xp.1, xp.2,... / xs.1, xs.2,...  => ntzp.1, ntzp.2,... ;
       .
       .
       .
   END;

   deftexec : Texec;

END;
```

Figure 2.5. Pulsed combinational function specification format

```
CFS name : interlaced

   XS.1 FROM name.ZS.k : input, input,...;
   XS.2 FROM name.ZS.1 : input, input,...;
    .
    .
    .

   ZS.1 TO name.XS.r, name.XS.s,... : output, output,... ;
   ZS.2 TO name.XS.t, name.XS.u,... : output, output,... ;
    .
    .
    .

   FOUTS : list

      xs.1, xs.2,...  =>  ntzs.1, ntzs.2,... ;
      xs.1, xs.2,...  =>  ntzs.1, ntzs.2,... ;

   END;

   defzsinit : zs.1, zs.2,... ;

   deftexec : Texec;

END;
```

Figure 2.6. Static combinational function specification format

```
DELP name : interlaced

   XP.1 FROM name.ZP.i : input, input,... ;

   ZP.1 TO name.XP.m, name.XP.n,... : output, output,... ;

   deftdel : Tdel;

END;
```

Figure 2.7. Pulsed delay specification format

E. Pulsed Delay

The format for specifying a pulsed delay (DELP) is presented in Figure 2.7.

A pulsed delay has a single pulsed input and a single pulsed output and a constant delay time, Tdel, specified in deftdel. A DELP operates to make

zp.1(t) = xp.1(t - Tdel)

A DELP is an idealized function; it is never busy and can store an arbitrarily large number of inputs in a finite time interval. The initial state of a DELP is that of being empty.

F. Static Delay

The format for specifying a static delay (DELS) is presented in Figure 2.8.

A static delay has a single static input and a single static output and a constant delay time, Tdel, specified in deftdel. A DELS operates to make

zs.1(t) = xs.1(t - Tdel)

A DELS is an idealized function; it is never busy and can store an input that changes an arbitrary number of times in a finite time interval. The initial state of a DELS is that of a constant equal to the initial input.

```
DELS name : interlaced

   XS.1 FROM name.ZS.k : input, input,...;

   ZS.1 TO name.XS.r, name.XS.s,... : output, output,... ;

   defzsinit : zs.1 ;

   deftdel : Tdel;

END;
```

Figure 2.8. Static delay specification format

G. Queue

The format for specifying a queue (QUE) is presented in Figure 2.9.

```
QUE name : interlaced

   XP.1 FROM name.ZP.i : input, input,... ;
   XP.2 FROM name.ZP.j : deq;

   ZP.1 TO name.XP.k, name.XP.l,... : output, output,... ;

   ZS.1 TO name.XS.m, name.XS.n,... : open, closedempty, closednotempty;

   defteng : Tenq;

   deftdeq : Tdeq;

END;
```

Figure 2.9. Queue specification format

A queue has a single data input and output streams (XP.1 and ZP.1) for which the user enumerates identical sets of signal values. Additionally, a queue has single control input stream (XP.2) and output

stream (ZS.1). The static control output indicates in which of the three control states the queue resides. A queue operates as follows:

1) If a pulsed data input arrives at an open queue, it pulses the data out and goes to the closedempty control state.
2) If a pulsed data input arrives at a closed queue, it stores the data in FIFO order and assumes the closednotempty control state.
3) If the deq control input arrives at a closednotempty queue, the oldest stored data is dequeued and pulsed out. If no more data items remain in the queue, the control state becomes closedempty.
4) If the deq control input arrives at a closedempty queue, the queue assumes the open control state; no pulsed output is produced.
5) There are two independent busy time intervals, Tenq and Tdeq, associated with queue operation. These limit rates at which successive data inputs and control deq's can be validly imposed on the queue.

H. Derivative

The format for specifying a derivative (DER) is presented in Figure 2.10.

A derivative has a single static binary input and a single pulsed output. A DER detects rising and falling edges in the static input and pulses out r and f as appropriate. A DER is an idealized function; it

is never busy and can respond to an arbitrarily large number of edges in a finite time interval.

```
DER name : interlaced
   XS.1 FROM name.ZS.i : 0, 1 (or h, l);
   ZP.1 TO name.ZP.j, name.ZP.k,... : r, f;
END;
```

Figure 2.10. Derivative specification format

I. Clock

The format for specifying a clock (CLK) is presented in Figure 2.11.

```
CLK name : interlaced
   XP.1 FROM name.ZP.i : reset, start;
   ZP.1 TO name.XP.j, name.XP.k,... : timeout;
   ZS.1 TO name.XS.1, name.XS.m,... : reset, running, expired;
   deftclk : Tclk;
END;
```

Figure 2.11. Clock specification format

A clock is a controlled timer with a fixed timeout period, Tclk, specified in deftclk.

A clock has a single pulsed control input and a pulsed and a static control outputs. The static control output indicates the control state of the clock; it has two stable states, reset and expired. The reset input always drives the clock into the reset control state. The start input always initializes the timer function of the clock and puts it in the running control state. If allowed to run to completion, (i.e., not started or reset in the Tclk time interval since the last start) the clock pulses out a timeout and assumes the expired control state.

A CLK is an idealized function; it is never busy and reacts instantly to control inputs.

J. Environment

The environment component type gives the system designer or simulation user the freedom to name and connect-in a component whose behavior is not easily (if at all) specifiable as an instance of one of the eight basic component types already described. Two versions of ENV are specifiable in SAN:

terminal, which allows the simulation operator to connect into the system as a component during simulation execution through his terminal; when an ENV of this type is executing, the operator uses the on-line Terminal Mode Command Language (described in the next section) to interactively query the status of the global system, change the values of ENV parameters, store profiles of system

variables, set ENV output variables and schedule future ENV executions.

procedure, which allows the user to define the component operation via a general PASCAL procedure; in such a procedure the user may access all the system variables by global name references and may introduce new variables local to the ENV; the procedure can also invoke the functions of the on-line Terminal Mode Command Language by appropriate calls to built in monitor procedures.

The format for specifying a terminal ENV is presented in Figure 2.12. The ENV is named; the ranges and connections of all input and output variables are enumerated; the ENV type (terminal) is declared; the default (initial) values of all static outputs are given in defszinit. The busy time interval length, Texec, is specified in deftexec. Whether multiple simultaneous pulsed inputs to this single ENV component should be trapped is specified by assigning true or false to mulpulsecheck.

The ENV will be scheduled for execution during a simulation run when one of the following conditions occurs:

--a pulsed input arrives at the ENV.

--the boolean expression given in STARTEXP evaluates to true. The boolean expression is provided by the user and can involve some of the global system variables as described in the syntax diagram in the Appendix; the boolean expression is evaluated and tested by the simulator during execution at the times specified in

```
ENV name : interlaced

   XP.1 FROM name.ZP.i : input, input,... ;
   XP.2 FROM name.ZP.j ; input, input,... ;
    .
    .
    .

   XS.1 FROM name.ZS.k : input, input,...;
   XS.2 FROM name.ZS.1 : input, input,...;
    .
    .
    .

   ZP.1 TO name.XP.m, name.XP.n,... : output, output,... ;
   ZP.2 TO name.XP.p, name.XP.q,... : output, output,... ;
    .
    .
    .

   ZS.1 TO name.XS.r, name.XS.s,... : output, output,... ;
   ZS.2 TO name.XS.t, name.XS.u,... : output, output,... ;
    .
    .
    .

   FUNCTION : terminal;

   { alternatively, FUNCTION : procedure

                    ... PASCAL procedure ...

                    END ; }

   defzsinit : zs.1, zs.2,... ;

   deftexec : Texec;

   mulpulsecheck : boolean ;

   STARTEXP : boolean expression ;

   STARTEXPCHECK : never (or everytimechange or everyevent);

END;
```

Figure 2.12. Terminal and procedure environment specification formats

STARTEXPCHECK i.e., never, or at the completion of every simulated time interval (everytimechange), or after every simulated event (everyevent).

--the time has arrived for a start event previously filed for this ENV by the user with the Terminal Mode Command Language.

The format for specifying a procedure ENV is presented in Figure 2.12.

The use of general PASCAL procedures in the ENV component requires knowledge of the data structures and the global system variables used in the State Architecture Simulator (SAS) of the SAN model being simulated. The remaining paragraphs of this section which explains the use of general PASCAL procedures in ENV may be skipped until the SAS implementation has been discussed.

Within an ENV procedure, the user basically can access all the global system variables; however, only a subset of global system variables, which are useful to the users, are discussed in this dissertation. Users interested in all possibilities should study the SAS source listings. One group of these system variables are those related to the current state, input/output, and execution state of individual components. The detailed system variable names are shown in Figure 2.13. Notice that to reference a user component, the special prefix 'U$' is appended to the beginning of the component name. The reason is to distinguish the user component names from the PASCAL reserved words and the SAS variables used in the procedure.

Another group of system variables, which may be set to control simulator execution, is shown in Figure 2.14. The type of value assigned to each system variable will be discussed in the next chapter. The user can define new variables local to each ENV procedure. Besides the use of global system variables, the user can access some of the system procedures. Only those typically useful system procedures are shown in Figure 2.15. Examples on using system procedures will be provided in Chapter IV.

Variables	Meaning
fsm[U$name].cs .cxp[i] .cxs[i] .czp[i] .czs[i] .execstatus	The current state, the i^{th} current pulsed input, static input, pulsed output, static output and the execution state of the named FSM.
cfp[U$name].cxp[i] .cxs[i] .czp[i] .execstatus	the i^{th} current pulsed input, static input, pulsed output and the execution state of the named CFP.
cfs[U$name].cxs[i] .czs[i] .execstatus	the i^{th} current static input, static output and the execution state of the named CFS.
delp[U$name].cxp[1] .czp[1] .execstatus	the first current pulsed input, pulsed output and the execution state of the named DELP.
dels[U$name].cxs[1] .czs[1] .execstatus	the first current static input, static output and the execution state of the named DELS.
der[U$name].cs .cxs[1] .czp[1] .execstatus	the current state, the first current static input, pulsed output and the execution state of the named DER.

clk[U$name].cs.state .cs.time .cxp[1] .czp[1] .czs[1] .execstatus	the current control state, the clock start time, the first current pulsed input, pulsed output, static output and the execution state of the named CLK.
que[U$name].cxp[i] .czp[1] .czs[1] .cs.state .cs.size .enqstatus .deqstatus	the i^{th} current pulsed input, the first current pulsed output, static output, control state queue size, the execution state with respect to loading data, and the execution state with respect to dequeueing data from the named QUE.
env[U$name].cxp[i] .cxs[i] .czp[i] .czs[i] .execstatus .tstart .updatestate.task .updatestate.ntzp[i] .updatestate.ntz[i] .updatestate.time	the i^{th} current pulsed input, static input, pulsed output, static output, the execution state, the prescheduled time of a future start event, the indication of a future start event, the i^{th} next pulsed output, static output and the next update time of the named ENV.

Figure 2.13. Global system variables related to the status of each component

Variables	Meanings
file_event	the event type of the next event to be scheduled in the event file.
file_kind	the component kind of the next event to be scheduled in the event file.
file_name	the component name of the next event to be scheduled in the event file.
file_time	the event executon time of the next event to be scheduled in the event file.
seed	the seed value for the random number generator.
syshalt	system half variables, true means simulation execution is stopped.
tbeg	beginning of the simulation time.
tend	ending of the simulation time.
tnow	current simulation time.

Figure 2.14. Global system variables which can be set to control simulator execution

Procedure names	Description
pname(U$name)	Print the status of the named component.
pall	Print the status of all the system components
peventfile(output)	Print the contents of the event file
sname(U$name)	Save the current status of the named component in the system data file
sall	Save the current status of all the system components in the system data file
azp(n,zp)	Assign the value zp to the indexed pulsed output of the currently executing environment
azs(n,zs)	Assign the value zs to the indexed static output of the currently executing environment
astartcheck(option)	Assign the value option (never, everytimechange or everyevent) to the currently executing environment
fabs(T)	Schedule a future start event for the currently executing environment at time T, T must be greater than or equal to tnow

finc(T)	Schedule a future start event for the currently executing environment at time tnow + T, T must be greater than or equal to zero
uabs(T)	Schedule an update event for the currently executing environment at time T, T must be greater than or equal to tnow, this command makes the currently executing environment busy until T
uinc(T)	Schedule an update event for the currently executing environment at time tnow + T, T must be greater than or equal to zero, this command makes the environment busy until tnow + T
randint(max)	A function that returns a random integer between 1 and max.

Figure 2.15. Definition of user accessible SAS procedures

III. STATE ARCHITECTURE SIMULATOR

A. Overview of the State Architecture Simulator

The State Architecture Simulator (SAS) is made up of a VAX command language program, named SAS.COM, and two PASCAL programs namely TRANSFORM.PAS and SAS.PAS. The SAS compiles, executes, and reports on simulations of user supplied SAN models. SAS was developed and is running on a VAX 11/780 at Iowa State University using the VAX VMS V2.0 operating system.

The control of the SAS environment is governed by the VAX command language program, SAS.COM. Here, we will present an overview of the control structure of the SAS environment as shown in Figure 3.1.

The SAS begins by performing Transformation (2) on a user supplied SAN System Specification (1). In this step, the SAS scans the SAN Specification and compiles the non-PASCAL portions of the specification into equivalent PASCAL source code modules (3). Control is then passed to the PASCAL Compiler (4) which compiles the source modules into object modules (5). The object modules are then linked (7) to a set of standard pre-compiled SAS modules (6) creating a robust executable file (8) corresponding to the original system specified by the user. Control then passes to the executable system simulation file which proceeds through three major steps:

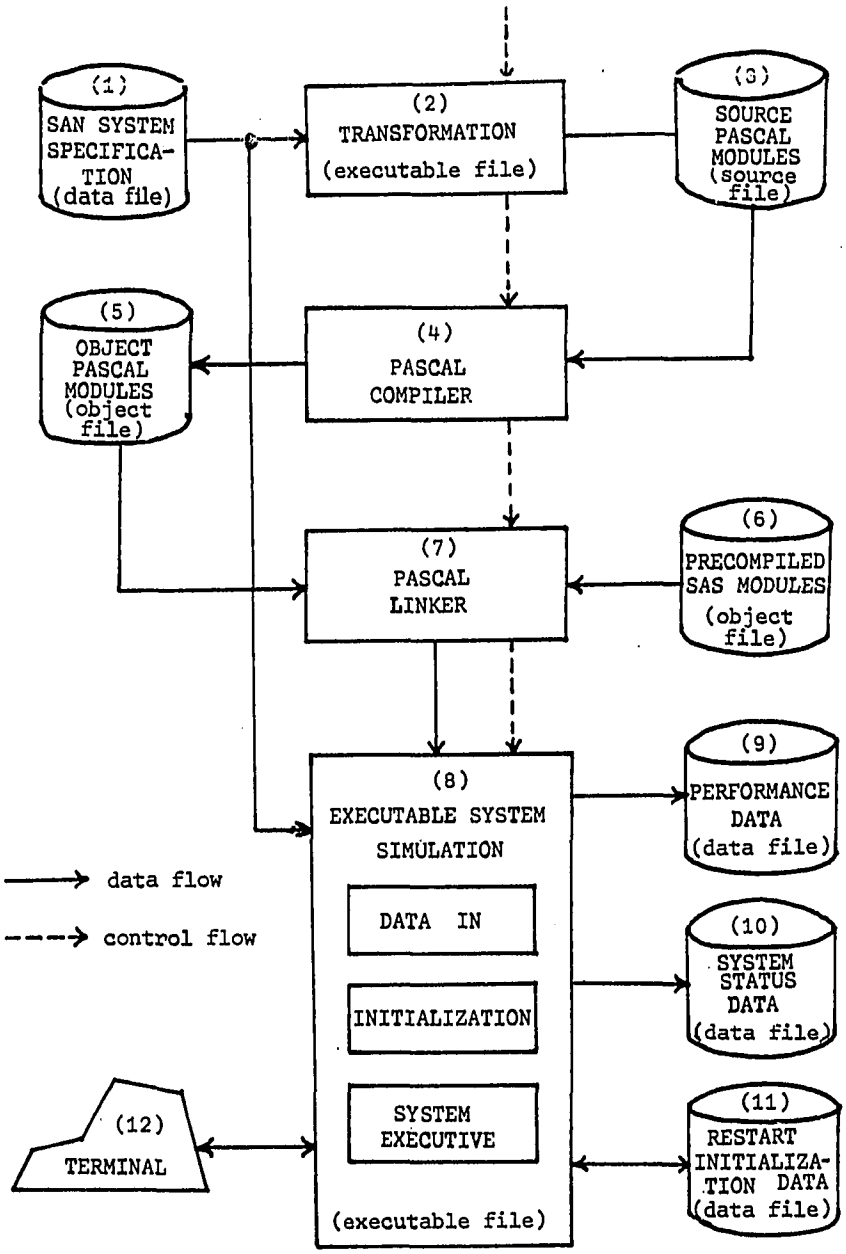


Figure 3.1. Control structure of the SAS environment

1) Data Input: scans the original SAN system specification, extracting those model parameters that are stored as table entries in the simulation (e.g., component input/output sets, execution times).

2) Initialization: can be either new or old (restart). For a new initialization, the SAS computes initial static outputs and inputs for all components based on their initial or default values. The SAS then examines that all initial component outputs are functionally and stably consistent with the given initial component inputs. If the system is unstable, simulation execution terminates at this point. For an old initialization, the SAS reloads the simulated system status from the restart initialization data file (11), which had stored the system status at the end of a previous system execution.

3) System Executive: carries out the simulated execution of the user's system. Performance trace reports are entered as appropriate in a performance data file (9); system status reports are entered in the system data file (10); user interaction with the model through the execution of terminal environment components is carried out through the user's terminal. SAS error reports are also made to the user via the terminal.

The System Executive synchronizes most of the simulation model activities using an event file, the structure of which is shown in Figure 3.2. Events associated with the same simulation time are grouped in the same event set. The entry for each event specifies the kind and name of basic component and the type of event involved.

Events are removed from the event file and acted upon in simulation time in chronological order, events within an event set being selected randomly.

Events are added to the event file (scheduled) as a result of component response to the currently executing event (e.g., pulsed output to a new component, future timeout for a starting clock).

Simulation time is advanced every time the current event set becomes empty. SAS steps through a series of values to pick up any scheduled environments or trace procedures prior to taking the value corresponding to that of the next event set in the event file.

The simulation halts when the preset simulation halt-time, Tend, is exceeded, or when the user invokes a halt through the Terminal Mode Command Language.

Whenever a terminal environment executes or a system error occurs, the system executive makes the on-line command monitor available to the user through the user terminal.

SAS offers several trace functions. In particular, the user may specify that SAS sample and save the value of any specified set of SAN system variables or the values of specified boolean expressions of the

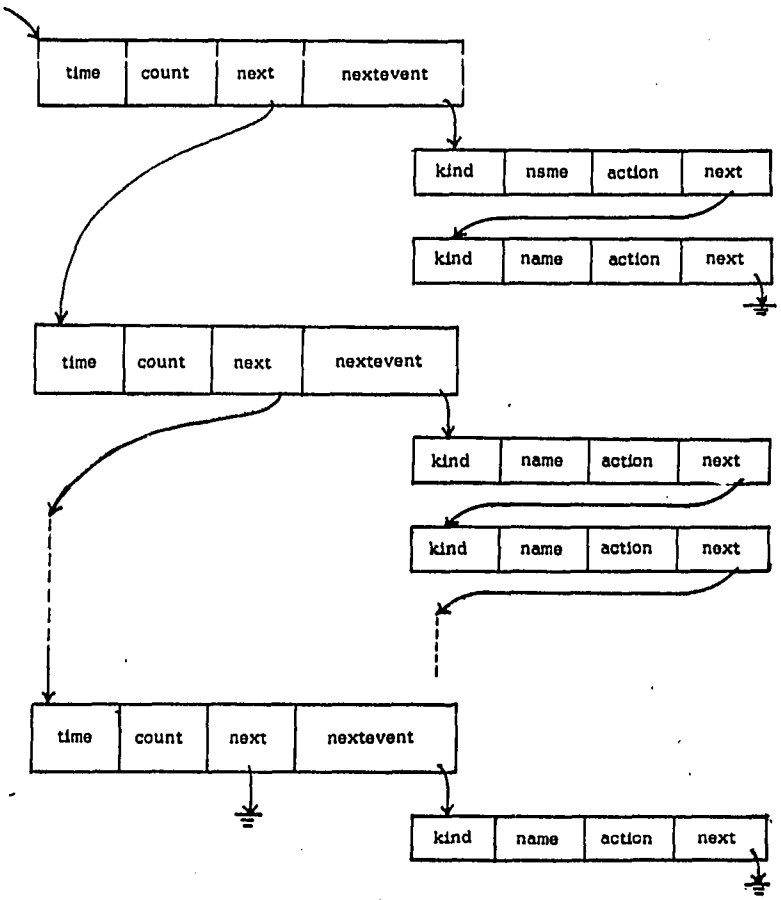


Figure 3.2. Event file structure

SAN system variables. The trace functions are invoked by augmenting the initial SAN system model with one or more instances of the three kinds of trace specifications as described in Chapter III.F.8.

At the termination of a simulation run, and at the user's option, the final status of the total simulation system is saved in the restart initialization file. The final status file of the run can be used at a later time to initialize a new simulation run that will continue from the state in which the initial run stopped.

The user may optionally include an initialization specification (INIT) in the SAN system specification in which is indicated the values of simulation time at which the run should begin and end and whether SAS should trap multiple simultaneous pulsed inputs to single components.

SAS consists of three programs, which are made up of modules of procedures. The first program, TRANSFORM.PAS, implements the transformation process and is made up of four modules of PASCAL procedures; namely: TRANPARAM, TRANSFORM, TRANSM, and TRANSUP. The second program, SAS.PAS, implements the Data Input, Initialization and System Executive processes. This program is made up of fourteen predefined modules of PASCAL procedures; namely: DATASTRU, SAS, DATAIN, HIST, INIT, ERROR, INILOAD, RELOAD, INTERNAL, SM, SCHEDULE, TERM, SMPS, STATSAVE, and a user defined module of PASCAL procedures called USER. USER is generated by the transformation process in compiling the procedure portions of the SAN specification into equivalent PASCAL source code

modules. The third program is a VAX command language program called SAS.COM.

SAS.COM governs the control of the SAS environment. SAS.COM first invokes the Transformation process; then the PASCAL compiler and linker are invoked to compile the PASCAL module, USER, and to link USER.obj, the USER object file, with the precompiled SAS.obj to produce a system executable image. SAS.COM then executes the system executable image.

In the SAS discussion below, the SAS source listings are frequently referenced. The listings[1] accompany this dissertation in a separate binding. To help locate the source lines, the source listings are separated into twenty modules. To reference a line in a particular module, the module number and the line number within the module (module number, line number) are indicated. The module names and their corresponding module numbers are shown in Table 3.1. For example, maxstrlength = 40; appears in line fifteen of the module DATASTRU, which is the first module. This line is referenced by (1, 15).

The following sections will present a detailed overview of the SAS implementation. First, we will describe the data structures of the SAS program; how the SAS keeps track of the necessary data of each component, the event file, and the performance trace.

---

[1]The listings can be obtained from the author.

Table 3.1. SAS program modules reference table

Function	Module Name	Module Number
Data Structure	DATASTRU.PAS	1
Data Input	SAS.PAS	2
	DATAIN.PAS	3
	HIST.PAS	4
	INIT.PAS	5
	ERROR.PAS	6
Initialization	INILOAD.PAS	7
	RELOAD.PAS	8
System Executive	INTERNAL.PAS	9
	SM.PAS	10
	SCHEDULE.PAS	11
	TERM.PAS	12
	SMPS.PAS	13
	STATSAVE.PAS	14
Transformation	TRANPARAM.PAS	15
	TRANSFORM.PAS	16
	TRANSM.PAS	17
	TRANSUP.PAS	18
Operation Steps	SAS.COM	19
User	USER.PAS	20

Second, the function of the Transformation process will be described: how the Transformation process scans through the user defined SAN model to generate a set of PASCAL compatible procedures. Third, we will describe the Data Input process: how the Data Input process scans through the user defined SAN models to check the SAN syntax and to establish the simulated system in the PASCAL data structure environment.

Fourth, we will describe the Initialization process. Fifth, we will describe the System Executive, which is the heart of the simulated system execution process.

B. Data Structure

This section presents an overview of the major data structures used in the SAS implementation. The serious reader is advised to have a general idea of the data structures of SAS before studying the SAS implementation. In the course of studying the SAS implementation, readers should not hesitate to reference the data structures file, DATASTRU.PAS, in order to understand the SAS implementation. As an aid to clear exposition, all occurrences of constant, type, variable and procedure names in the SAS.PAS program will be underlined in this section.

The following paragraphs will describe the philosophy of string usage, the constant and type declaration of the DATASTRU.PAS file, and highlight the data structures for components, event file, and performance trace.

1. Philosophy of string usage

The user can use arbitrary alphanumeric strings to name different kinds of components and to assign state, input and output variables. In this way, the user can name the component, state, input and output values according to their generic names, which provides a better feeling and

understanding of the SAN model. However, string type may not apply to each kind of PASCAL variable. In SAS, each component is referenced via an array indexed by the component enumerated name, e.g. fsm: array [fsmname] of single fsm (1, 601). All FSM components are referenced in the array fsm indexed by their component enumerated names. In PASCAL, the array index cannot be a string. To get around the problem, all the component names (alphanumeric strings) are mapped in one-to-one correspondence to the SAS predefined enumerated names. In this case, the user can still specify and refer to the components via their generic names while the SAS translates the component names into enumerated names used by SAS. The details on the translation (transformation) will be discussed in the Transformation process, Chapter III.C.

2. Constant and type declaration

In the DATASTRU.PAS file, there is a set of predefined constants which are not changed throughout the SAS execution. Some of the values for the constants are arbitrarily assigned and some are intentionally assigned to set a limit for the size of the SAN models. In the present implementation, the SAN models are allowed to have a maximum of 162 components (1, 7); the maximum length of a string is 40 characters (1, 15). The maximum number of the pulsed or static input/output streams of a component is 20 each (1, 18); the maximum number of fanouts from an output is 10 (1, 20). Besides that, a set of integer constants (1, 25) are defined for error messages in the SAN syntax and a set of integer constants (1, 40) for error messages during the SAS execution. There are

constants defined for the SAN delimeters (1, 54) and also a set of string constants (1, 63) for the SAN and SAS special symbols. All of them are listed in the DATASTRU.PAS file.

The SAS defines smname (1, 157) as a set of enumerated names for components. The enumerated names, S$UNCONNECTED, S$SYSTEM-MONITOR, fsm0, cfp0, cfs0, delp0, dels0, que0, clk0, der0 and env0 are reserved for the SAS system. S$UNCONNECTED are regarded as unconnected. S$SYSTEM-MONITOR is a terminal ENV, which allows the user to examine the system status whenever a system execution error occurs. fsm0, cfp0, cfs0, delp0, dels0, que0, clk0, der0, and env0 are used by SAS to indicate that there is no user defined FSM, CFP, CFS, DELP, DELS, QUE, CLK, DER and ENV components, respectively. The rest of the enumerated names are used to map with user defined component names, so that there is a one-to-one correspondence between the user defined component names and the enumerated names. The SAS separates all the enumerated names into nine groups according to nine kinds of components. There are fifty-two enumerated names assigned for FSM components ranging from S$UNCONNECTED to fsm50 (1, 174). S$UNCONNECTED and fsm0 are reserved for the SAS system and the other fifty enumerated names, fsm1 to fsm50, are used for user defined FSM components. In the same manner, CFP has twenty enumerated names ranging from cfp1 to cfp20, for user defined CFP components; CFS has twenty, ranging from cfs1 to cfs20; DELP has ten, ranging from delp1 to delp10; DELS has ten, ranging from dels1 to dels10; QUE has ten, ranging from que1 to que10; CLK has ten, ranging from clk1 to clk10; DER has ten, ranging from der1 to der10; ENV has twenty, ranging from env1 to env20.

3. Components

All component characteristics are kept in an array indexed by an enumerated name. Each kind of component has its own array name. The FSMs are represented by the array, fsm[fsmname] (1, 601), and indexed by a member of the FSM enumerated name, fsmname. Each FSM array has a record of elements, singlefsm (1, 261), holding all the information on the FSM component. The singlefsm holds the following information:

S	a linked list for the state set
XP	a linked list for the pulsed input set
nXP	number of pulsed input streams
XS	a linked list for the static input set
nXS	number of static input streams
ZP	a linked list for the pulsed output set
nZP	number of pulsed output streams
ZS	a linked list for the static output set
nZS	number of static output streams
fnsfirst	a linked list for the list of FNS vectors
fnstype	type of FNS function (procedure or list)
fnsproc	a number corresponding to the case index of the FNS procedure in the procedure fsmfunction, which contains all the procedures defined in the FSM components
foutpfirst	a linked list for the list of FOUTP vectors
foutptype	type of FOUTS function (procedure or list)

foutpproc	a number corresponding to the case index of the FOUTP procedure in the procedure fsmfunction
foutsfirst	a linked list for the list of FOUTS vectors
foutstype	type of FOUTS function (procedure or list)
foutsproc	a number corresponding to the case index of the FOUTS procedure in the procedure fsmfunction
destzp	a record storing the destinations for each pulsed output stream
destzs	a record storing the destinations for each static output stream
texec	time required to execute the component
execstatus	execution status of the component; it can be Idle, Pend, or Busy
cs	current state value
cxp	the value of the current pulsed inputs
cxs	the value of the current static inputs
czp	the value of the current pulsed outputs
czs	the value of the current static outputs
cxsload	a boolean value representing whether the current static inputs have been changed at the current simulation time, tnow
updatestate	an update record storing the next update time, next state, next pulsed outputs, next static outputs and the task of the update process

The updatestate record is a general purpose update record (1,251). Not all the elements in the update record are applicable to each kind of component. For example, the task entry with valid values (notask, delete, add) is used by a queue to indicate whether data is to be added into or deleted from the queue in the update operation.

The above discussion of a single FSM data structure can be generally applied to all the other kinds of components. The details of each kind of component will not be discussed here, but they are listed in the DATASTRU.PAS file.

4. Event file

The event file is organized in a multi-linked list structure as shown in Figure 3.2. The multi-linked list groups all the events with the same execution time into one event set and those events with different execution times into different event sets. Each eventsetentry (1, 493) has a record of elements which consists of the following:

time — the execution time of the event set

count — the total number of event entries in the event set

nextevent — a pointer to the first event entry of the event set

next — a pointer to the next event set

Each evententry (1, 485) within an event set also has a record of elements which consists of the following:

kind — the kind of the component

name — the name of the component

action — the type of event (start or update)

next a pointer to the next event entry within the event set

Event sets are organized in chronological order. The event set with the lowest execution time is stored on the top of the list. Within an event set, the event entries with ENV kind are stored on the top of the list; the event entries with the other kinds of components are stored below the ENV kind. The ENV kind events are given higher priority to be executed within an event set because when an ENV component is in terminal mode, we want the user to always have first chance to examine the system.

5. Performance trace

SAS offers three types of trace functions. They are the regular variable history, conditional variable history and regular expression history. Each of the trace functions is organized in a multi-linked list structure.

The data structure for a set of regular variable histories is shown in Figure 3.3. Each instance of a regular variable history is represented by a varhistregentry (1, 541) record. All the varhistregentry records are linked together in a linear linked list. The first instance of the varhistregentry is pointed to by the regular variable history header, varhistregpt (1, 633). The varhistregentry record (1, 541) consists of the following information:

histname the name of the regular variable history instance

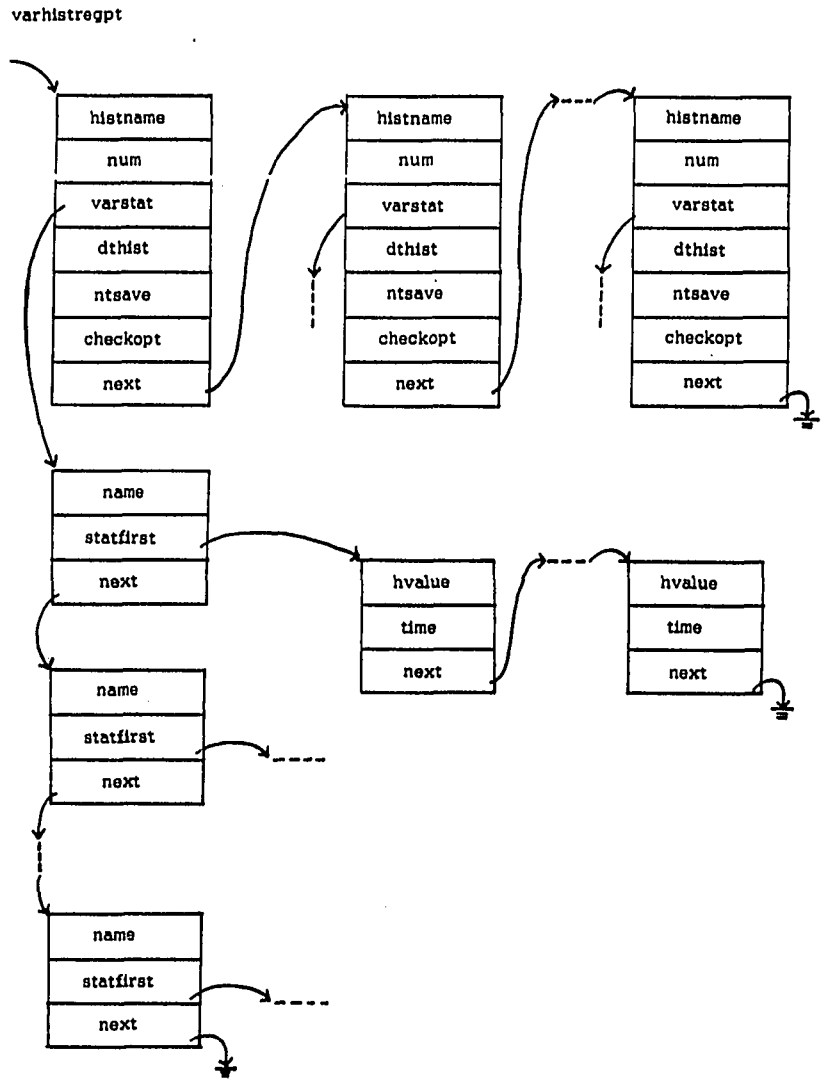


Figure 3.3. Data structure of a set of regular variable histories

num	the order of the regular variable history instance's occurrence, e.g., 1 implies the first regular variable history instance
varstat	a header pointing to the first record of a variable history
dthist	the simulation time intervals at which the set of variables value are to be recorded
ntsave	the next simulation time for the set of variable values to be recorded
checkopt	the frequency for which the set of variables are to be recorded at the recording time (everyevent means a sample is recorded after every event executed at the recording time; everytimechange means precisely one sample is recorded just before advancing simulation time forward from the current recording time; never means the trace is disabled)
next	a pointer pointing to the next regular variable history record

For each variable, there is a variable history record called histentry (1, 533) to save the trace of the variable. The histentry consists of the following:

name	the name of the variable
statfirst	a pointer pointing to the first data element of the variable

next — a pointer pointing to the next variable history record

Each data element is represented by a statentry (1, 504), which consists of the following:

hvalue — the value of the data element

time — the time at which the data is recorded

next — a pointer pointing to the next data element

The data structure for a set of conditional variable histories is shown in Figure 3.4. Each instance of a conditional variable history is represented by a varhistconentry (1, 354) record. All the varhistconentry records are linked together in a linear linked list like the varhistregentry records. The first instance of the varhistconentry is pointed to by the conditional variable history header, varhistconpt (1, 634). The varhistconentry record is similar to the varhistregentry record except that the varhistconentry record does not have the dthist and ntsave entry to indicate the next simulation time the samples are recorded. Instead, each instance of a conditional variable history has a boolean expression. The boolean expression is evaluated at the times specified in the checkopt entry. The variables are recorded at those times when the boolean expression evaluates to true.

The data structure for a set of regular expression histories is shown in Figure 3.5. Each instance of a regular expression history is represented by an exphistentry (1, 564) record. All the exphistentry records are linked together in a linear linked list like the varhistregentry records. The first instance of the exphistentry is

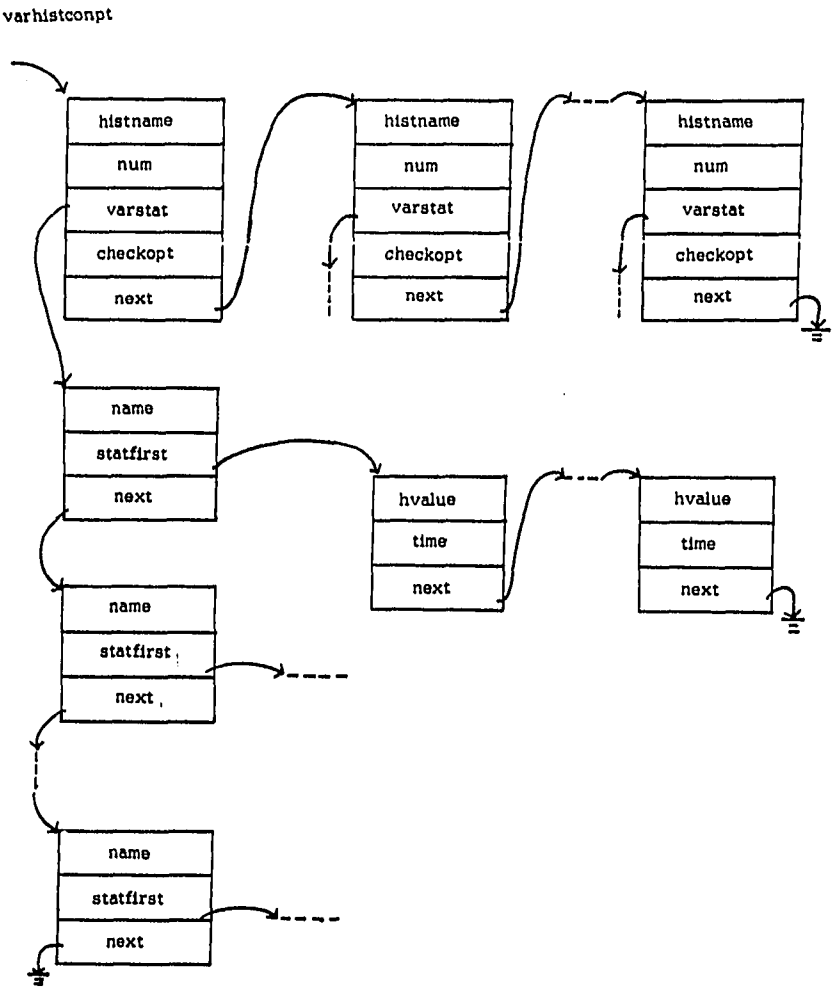


Figure 3.4. Data structure of a set of conditional variable histories

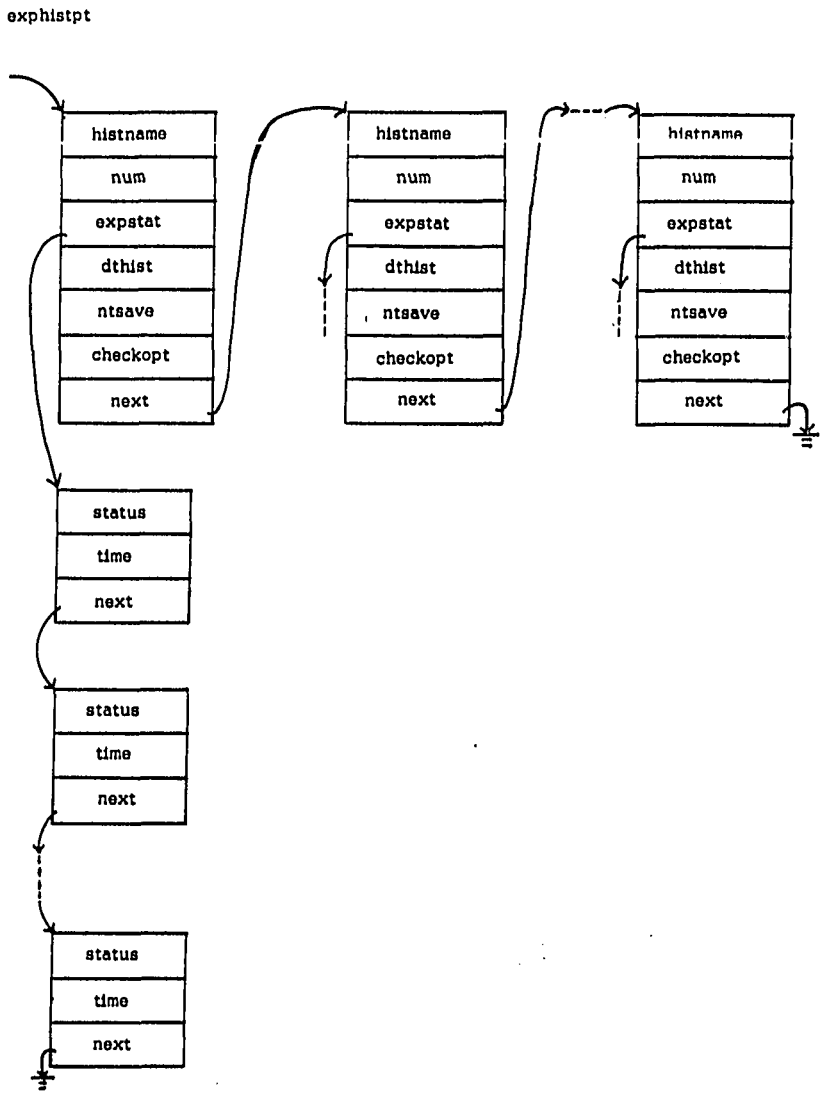


Figure 3.5. Data structure of a set of regular expression histories

pointed to by the expression history header exphistpt (1, 635). The exphistentry record is similar to the varhistregentry record except that the exphistentry record has a expstat record pointer (1, 567) instead of a varstat record pointer (1, 544). The expstat record pointer points to a boolean status data element called statusentry (1, 514). Each statusentry record consists of the following:

status	the boolean value of the expression
time	the time at which the expression was evaluated
next	a pointer pointing to the next data element

C. Transformation

The function of the Transformation process is to compile the local procedures and the boolean expressions in the SAN specification into equivalent globally compatible procedures. All the global procedures generated by the Transformation process are stored in USER.PAS. As an aid to clear exposition, the Transformation process will be discussed in reference to a simple SAN model as shown in Figure 3.6. This SAN model has two ENV components and a FSM component. Three kinds of performance traces and an initialization are also included in the simple SAN model.

1. Transformation of local procedures

In SAN, the Next State Function (FNS), Pulsed Output Function (FOUTP), and Static Output Function (FOUTS) of a FSM, the FOUTP of a CFP and the FOUTS of a CFS can be specified via the procedure option as discussed in Chapter II. These local procedures have to be extracted

```
ENV Terminal : interlaced
        ZP.1 TO Pulsegen.XP.1 : start;
        FUNCTION : terminal;
        deftexec : 0.0;
        STARTEXP : true;
        STARTEXPCHECK : everytimechange;
END;

ENV Pulsegen : interlaced
        XP.1 FROM Terminal.ZP.1 : start;
        ZP.1 TO Counter.XP.1 : reset,inc;
        FUNCTION : procedure
          procedure Pulsegenproc;
          var i:integer;
              a:array[1..2] of string;
          begin
              a[1]:='reset                                                    ';
              a[2]:='inc                                                      ';
              i:=randint(5);
              with env[USPulsegen] , updatestate do
              begin
                  if(i=3) then ntzp[1]:=a[1]
                          else ntzp[1]:=a[2];
              end; {end with}
          end;
        END;
        deftexec:1.0;
        STARTEXP : false;
        STARTEXPCHECK : never;
END;

FSM Counter : interlaced
        S : 0,1,2,3,4,5;
        defsinit : 0;
        XP.1 FROM Pulsegen.ZP.1 : reset,inc;
        ZS.1 UNCONNECTED : 0,1,2,3,4,5;
        FNS : procedure
          if oxp.1='reset' then nts:='0'
          else
          begin
              if cs='0' then nts:='1'
              else if cs='1' then nts:='2'
              else if cs='2' then nts:='3'
              else if cs='3' then nts:='4'
              else if cs='4' then nts:='5'
              else nts:='5';
          end;
        END;
        FOUTS : procedure
          ntzs.1:=nts;
        END;
        deftexec : 0.0;
END;

VARHISTORY TraceCount : regular
        VARIABLES : Counter.czs.1;
        DTHISTORY : 1.0;
        CHECKOPT : everytimechange;
END;.

VARHISTORY TracePulse : conditional
        VARIABLES : Pulsegen.czp.1;
        CONDITION : Pulsegen.czp.1='reset';
        CHECKOPT : everytimechange;
END;

EXPHISTORY Tracestatus : regular
        EXPRESSION : (Pulsegen.czp.1='inc')and(Counter.czs.1='1');
        DTHISTORY : 1.0;
        CHECKOPT : everytimechange;
END;

INIT
        tbeg : 1.0;
        tend : 10.0;
END;
```

Figure 3.6. A simple SAN model

and transformed into globally compatible procedures in order to be used by the SAS.PAS program. All the local procedures declared in the FSM, CFP and CFS instances will be transformed and placed together in procedures fsmfunction (20, 755), cfpfunction (20, 787) and cfsfunction (20, 798), respectively. Each instance name is mapped into its corresponding enumerated name according to its order of appearance in the SAN specification. For example, the first FSM component, Counter, in the SAN model is given the enumerated name, fsm1 (20, 910).

In the Transformation process of the local procedures in FSM components, the following operations are done:

1) All the local state, input and output variable names are transformed into their corresponding global PASCAL variables. For example, the cxp.1 in the FNS procedure of Counter is transformed into fsm[fsm1].cxp[1] (20, 762).

2) All the character strings declared in the local procedures are mapped into the SAS defined string array: constval[i], according to the order of the string appeared in the SAN model. The character string is mapped into an array variable, constval[i], so that the forty characters long string can be shrunk to a twelve character variable. It will make the indentation of the transformed procedure look better. For example, 'reset' in the Counter's FNS procedure is mapped into constval[1] (20, 762), '0' in the Counter°s FNS procedure is mapped into constval[2] (20, 762).

3) All the PASCAL reserved names remain unchanged, e.g., else if (20, 766).

4) Each local procedure mapped into the procedure fsmfunction is referenced by passing an integer to the procedure fsmfunction. The number by which the local procedure is referenced is equal to the order of the place the local procedure appeared in the SAN model. For example, the FNS procedure of the Counter is the first procedure to appear in the FSM component. This procedure is referenced by passing an integer value 1 to the procedure fsmfunction (20, 758).

The transformation of local procedures in CFP and CFS components is the same as in the FSM components. In case there is no local procedure declared in a FSM, CFP or CFS component, its corresponding procedures fsmfunction, cfpfunction and cfsfunction consist only some dummy statements. For example, there is no CFP local procedure declared in the example SAN model. The procedure cfpfunction, which holds all the transformed CFP local procedures, has only some dummy statements. The dummy statements (20, 790-796) are a convenience to the designer and also allow the dummy procedure to return an error message in case the dummy procedure is called.

2. Transformation of ENV procedure

The procedures defined in the ENV components are PASCAL compatible procedures except that the component names used in referencing a component should be their enumerated names. In the Transformation

process, all these user defined component names are transformed into their corresponding enumerated names. For example, U$Pulsegen in the ENV procedure of the SAN model is transformed into its enumerated name, env2 (20, 817). Each ENV procedure is named by appending a suffix 'proc' to the end of the ENV component name. For example, the ENV procedure in the Pulsegen is named as Pulsegenproc. In order to reference these ENV procedures, the procedure envfunction (20, 824) is generated by the Transformation process. The procedure envfunction has a case statement. The case statement branches off to call the proper ENV procedure according to the integer passed to the procedure envfunction. The order in which the ENV procedure names appear in the procedure envfunction is the same as they appear in the SAN model.

3. Transformation of boolean expression in ENV

The boolean expression of the STARTEXP line of each ENV instance in the SAN model is put under the boolean function envexpst (20, 834). Each of those boolean expressions are assigned to the boolean function variable envexpst. The order in which the boolean expressions appear under the case statement of the boolean function envexpst is the same order their corresponding ENV instances appeared in the SAN model. In transporting the boolean expression from the SAN model to the boolean function envexpst in the USER module, the following transformations are done:

1) All the SAN global variables are transformed into PASCAL compatible global variables.

2) All the character strings declared in the boolean expression are mapped into the SAS defined string array, constval[i], as was done in local procedure transformations.

3) All PASCAL reserved words remain the same.

In the example SAN model, there are two ENV instances. The STARTEXP of one ENV instance has a simple boolean expression with a constant value true and the other ENV instance has a constant value false. These two simple boolean expressions are transported to the boolean function envexpst (20, 834). The assignment statement (20, 839) under the branch number 1 of the case statement is transported from the first ENV instance. The assignment statement (20, 842) under the branch number 2 of the case statement is transported from the second ENV instance.

4. Transformation of regular variable history instances

The regular variable history trace is recorded via the procedure historynum (20, 849), which is generated by the Transformation process. The procedure historynum consists of a sequence of procedure calls to procedure history (20, 854). The procedure history records the value of the PASCAL global variables passed to the procedure. These variables are extracted from the SAN global variables in the line VARIABLES of the regular variable history instance. If there are N variables needed to be traced in one regular history instance, N calls of the procedure history are made to record the traces. In the example SAN model, a static output variable, Counter.czs.1, is specified to be traced under the regular

variable history instance, Tracecount. Counter.czs.1 is transformed into its PASCAL global variable, fsm[fsm1].czs.[1]; this variable is used as a parameter passed to the procedure history (20, 854). There are also two integer parameters to be passed to procedure history. The first integer identifies the order of the regular variable history instance appearing in the SAN model, the second integer identifies the order of the variables appearing in the line VARIABLES of a regular variable history instance. In the above example, the variable Counter.czs.1 appears in the first regular variables history instance and the variable is also the first one appearing in the line VARIABLES. Therefore, the two integers passed to the procedure history are both 1. All the history procedures for one set of variables of a regular variable history instance are put together under the same branch number of the case statement in procedure historynum (20, 853-855). The set of history procedures for the first regular variable history instance are put under the branch number 1, the set of history procedures for the second regular variable history instance are put under the branch number 2 and so on.

5. Transformation of the conditional variable history

The conditional variable history traces are recorded via the boolean function conexp (20, 861) and the procedure savehistcon (20, 874). They are both generated by the Transformation process. The function conexp consists of the boolean expressions specified in the CONDITION line of the conditional variable history instances. The boolean expressions are assigned to the function variable conexp (20, 866). In transporting the

boolean expression from the SAN model to the function conexp, the SAN global variables are transformed into PASCAL compatible global variables; the character strings are mapped into the SAS defined string array constval[i] similar to the transformation of the boolean expression in the ENV instance. If there is more than one conditional variable history instance, the multiple boolean expressions are placed under the case statement of the function conexp in the same order as the conditional variable history instances appeared in the SAN model.

The procedure savehistcon checks if the trace of the conditional variable history is to be recorded by comparing the check option and evaluating the boolean expression of the conditional variable history instance (20, 882). If they evaluate true, the procedure stathist (20, 884) is called to record the trace of the variables, as specified in the VARIABLES line of the conditional variable history instance. The SAN global variables named in the VARIABLES line are transformed into PASCAL campatible global variable as a parameter passed to the procedure stathist. The number of times the procedure stathist is called is equal to the number of listed variables in the trace needed to be recorded.

6. Transformation of the regular expression history

The boolean expressions of the regular expression histories are put together in the boolean function cexpst (20, 890). The boolean expressions of the regular expression histories are transported to the boolean function cexpst in the same manner as the boolean expressions of the conditional variable history are transported to the boolean function

conexp (20, 861). In the example SAN model, there is a regular expression history instance. The boolean expression of the regular expression history instance is transformed and put into the boolean function cexpst (20, 890). The boolean expression is assigned to the boolean function variable cexpst (20, 895) and put under branch number 1 of the case statement (20, 893-897). The boolean expression is put under branch number 1 because the regular expression history in which the boolean expression belonged to is the first one appearing in the SAN model.

7. Generation of procedure iniset

One other procedure generated by the Transformation process is the procedure iniset (20, 903). The procedure iniset assigns user defined component names to the array smword[i] and assigns the corresponding component enumerated name to the array smsym[i]. smword[1], smsym[1], smword[2] and smsym[2] are reserved for the SAS system components UNCONNECTED and SYSTEM-MONITOR (20, 905-908). smword[3], smsym[3], smword[4], smsym[4],---and so on are for user defined component names and their corresponding enumerated names. For example, the user defined component name, Counter, is assigned to smword[3]; the counter°s enumerated name, fsm1, is then assigned to smsym[3]. The enumerated names are assigned to the component names in the following order: the first FSM component appearing in the SAN model is assigned the enumerated name fsm1; the second appearing FSM component is assigned fsm2, and so on. The first CFP component appearing in the SAN model is assigned the enumerated name

cfp1; the second CFP component is assigned cfp2 and so on. As for CFS, DELP, DELS, QUE, CLK, DER, and ENV components, they are also assigned in the same manner. The details of the enumerated names for each kind of component had been explained in the previous section (Data Structure).

The procedure iniset also assigns enumerated names to the SAS global variables, which identify the first and last component enumerated name of each kind of components. For example, there is only one user defined FSM component; therefore, the first and last FSM component enumerated name variables, fsmf and fsml are both assigned fsml (20, 916-917). There is no user defined CFP component; the first and last CFP component enumerated name variables, cfpf and cfpl, are both assigned cfp0 (20, 918-919).

The SAS global variables which identify the set of enumerated names for each kind of component (20, 934-942) are also initialized in the procedure iniset. Finally, the procedure iniset establishes an array of character strings, which are defined by the user in the local procedures of the FNS, FOUTP and FOUTS and those in the boolean expressions of the SAN model. The character strings which appeared in the local procedures and the boolean expressions are extracted and assigned to the character string array, constval[i], in the same order the character strings appeared in the SAN model. For example, 'reset' the first user defined character string appeared in the local procedure FNS is assigned to constval[1], '0' the second appeared character string is assigned to constval[2] (20, 943-950).

8. An overview of program TRANSFORM

The program TRANSFORM implements the Transformation process as described in the above sub-sections. The program TRANSFORM consists of four modules of PASCAL source code namely: TRANPARAM.PAS, TRANSFORM.PAS, TRANSM.PAS and TRANSUP.PUS. TRANPARAM.PAS (15, 1-148) is a declaration file. This file consists of all the global constant (const), type, and variables (var) declarations used by the procedures of the program TRANSFORM. The module TRANSFORM.PAS consists of the main program TRANSFORM and some other utility procedures. TRANSM.PAS and TRANSUP.PAS consist of utility procedures called by the main program. The names and the functions of the utility procedures used by the main program will be described in the structure of the main program. The program TRANSFORM reads in the SAN model from the logical file named sanfile (15, 115). The program TRANSFORM generates a set of PASCAL compatible procedures and functions, namely: fsmfunction, cfpfunction, cfsfunction, envfunction, envexpst, historynum, conexp, savehistcon, cexpst and iniset; they are temporarily saved in the logical files, fsmfile, cfpfile, cfsfile, envfile, envexpstfile, historyfile, conexpfile, conhistfile, cexpstfile and inisetfile (15, 117-126) respectively. At the end of the execution of the program TRANSFORM, these temporary files are merged together by the SAS command procedure SAS.COM to a single module of PASCAL procedures and named USER.PAS. The program TRANSFORM also generates an error message file, errfile (15, 116), to record the error messages.

The following sub-sections will describe the structure of the main program TRANSFORM and the major procedures called by the main program. The major procedures include tranfsm : processes the FSM instances; trancfp: processes the CFP instances; trancfs: processes the CFS instances; resmname: processes the DELP, DELS, QUE, CLK, and DER instances; tranenv: processes the ENV instances; tranvar, tranreg, and trancon: process both the regular and conditional variable history instances; tranexp: processes the regular expression history instances, and inisetgen: generates the procedure iniset as described in Chapter III.C.7.

9. Structure of program TRANSFORM

The program TRANSFORM controls the flow of the Transformation process. The program starts out by assigning special strings to some string variables. The string variables saword[i] (16, 819-828) represent the local state and input/output variables named in the SAN model. The string variables pasword[i] (16, 829-926) represent all the PASCAL reserved words. The string variables sysword[i] (16, 927-935) represent some of the SAS system variables recognized in the Transformation process. The program then opens the SAN model specification file for reading by calling reset(sanfile) (16, 936) and opens the error message file to be written by calling rewrite(errfile) (16, 937). After that, the headings of all the procedure modules which will be generated by the Transformation process, are written into their corresponding files (16, 942-959). For example, the headings for the

procedure holding all the transformed local procedures in the CFP components to be written into the cfpfile file, is as follows (20, 787-790):

```
procedure   cfpfunction (num:integer),
var     cfpproc:integer,
begin
    case  num  of
```

The number of the local procedures declared for each kind of components are set to zero (16, 964-971). The headers for the list of each kind of component are initialized to nil. The boolean variable sanend is set to false indicating that the sanfile still has some components to be scanned.

Program TRANSFORM uses procedure readstr(var infile:text) (16, 198) to access the SAN file, represented by a logical name sanfile. The SAN file is scanned sequentially. In the beginning, procedure reset(sanfile) (16, 936) is called to set the SAN file pointer to the beginning of the SAN file. Everytime procedure readstr(sanfile) is called, a string is read from the SAN file. A string is a sequence of characters from the character set

symbol=['a'..'z',, 'A'..'Z', '0'..'9','%', '$'], (16, 158).

The character string is assigned to the global variable, tempstr. Procedure readstr(sanfile) also returns the last character read, which is not an element of symbol. The last character read is assigned to the global variable, c.

Procedure readstr(sanfile) starts out by setting tempstr to a string with only blank characters (16, 201). Notice there is a write statement

following each read statement in procedure readstr(sanfile). The purpose is to copy all the processed string into the error file, errfile. In case of any syntax error, the position of the last processed string can be located from the errfile. The repeat loop (16, 204-218) reads a character from the SAN file. If the character c is a blank, tab, or page mark character, then the process is repeated. Basically, the repeat loop skips all the leading blank, tab or page mark character of a string.

If the first character is a left brace, " ", then the characters after the left brace are skipped until the right brace character is encountered (16, 222-237). Lines (16, 222-237) allow procedure readstr(sanfile) to skip all the comment statements. After all the leading blank, tab, or page mark characters and the comment statements are skipped, procedure readstr(sanfile) starts to read the string. The while loop (16, 242-254) checks if the last character read is an element of the character set, symbol, and if the string is still less than al (which is forty) characters long, and eof(sanfile) is not true. The reading process continues until one of the above check statements is false. Procedure readstr(sanfile) sets tempstr to the last string read and sets c to the last character read.

The program TRANSFORM then calls the procedure readstr (16, 988) to read a character string from the sanfile. The procedure readstr returns the character string in the string variable tempstr. Then a while loop (16, 989-1002) is used to process all the component specifications. The program will jump out of the while loop if the end

of the sanfile is reached or sanend is true (16, 989). Inside the while loop, tempstr is compared with the nine kinds of component name. If tempstr equals FSMCON, then procedure tranfsm is called to process the FSM component (16, 991). (The FSMCON, CFPCON and so on are global constants declared in the data structure file for the strings 'FSM', 'CFP' and so on, respectively). If tempstr equals CFPCON, then procedure trancfp is called to process the CFP component (16, 992). If tempstr equals CFSCON, then procedure trancfs is called to process the CFS component (16, 993). If tempstr equals DELPCON, DELSCON, CLKCON, QUECON or DERCON, then procedure resmname is called by passing delpsym, delssym, clksym, quesym or dersym to process the DELP, DELS, CLK, QUE or DER component (16, 994-998). If tempstr equals ENVCON, then procedure tranenv is called to process the ENV component (16, 999). If tempstr does not equal any of the above component kinds, then sanend is set to true (16, 1000). If sanend is false, then procedure readstr is called again to read another character string.

If sanend or eof(sanfile) is true (16, 989), then the program starts to process the performance trace and initialization instances. This process also uses a while loop (16, 1005). Inside the while loop, the tempstr is compared to the various performance trace instance kinds and the initialization instance. If tempstr equals VARHISTCON, then procedure tranvar (16, 1007) is called to process the regular or conditional variable history. If tempstr equals EXPHISTCON, then procedure tranexp (16, 1008) is called to process the regular expression history. If tempstr equals INITCON, then procedure traninit (16, 1009) is called to

process the initialization instance. If tempstr is not equal to any of the above character strings, then procedure error(err33) (16, 1012) with an error message number, is called. If any error occurs, then the Transformation process will halt by calling the procedure stop (16, 1013). If there is no error, then procedure readstr is called again to read in another character string. The while loop process continues until the end of the sanfile is reached.

After processing the SAN model, the ending parts of all the procedure modules generated by the Transformation process are written into their corresponding files (16, 1021-1030). For example, the ending parts for the procedure holding all the transformed local procedures for the CFP components to be written into the cfpfile file is as follows (20, 779-782):

```
        otherwise execerr(E26);
        if errflag then execstop;
        end;   end of cfpfunction
```

The procedure inisetgen (16, 1031) is called to generate the module iniset. If there is any error encountered during the Transformation process, procedure ptranerr (16, 1033) is called to print out the error messages. At the end, the error message file is closed to finish the whole Transformation process.

10. Procedure tranfsm, trancfp and trancfs

Procedure tranfsm (17, 443) transforms all the procedures defined in the FSM instances of a SAN model into a globally compatible procedure

fsmfunction (20, 755). Procedure tranfsm first reads the FSM component name by calling readstr (17, 450). The FSM component name is stored into the FSM component name linked list; the head of the linked list is pointed to by fsmpt (17, 466). The procedure then skips all the specification until the string 'FNS', 'END', or eof(sanfile) (17, 470) is encountered. The syntax of the rest of the FSM specification will be checked later in the Data Input process.

If tempstr equals 'FNS', the procedure saves the function type (17, 475). The procedure then reads another string. If tempstr equals 'procedure', the number of the FSM procedure is increased by one (17, 484), the procedure heading (17, 485) and procedure transproc (17, 486) are called to transform the local procedure into a globally compatible procedure. If tempstr equals 'list', the procedure skipproc (17, 489) is called to skip to the end of the FNS function specification.

Procedure tranfsm then skips all other specifications until the string 'FOUTP', 'FOUTS', 'END', or of(sanfile)(17, 497) is encountered. If tempstr equals 'FOUTP' or 'FOUTS', the procedure goes through similar steps to the FNS transformation (17, 499-544).

After processing FNS, FOUTP and FOUTS specifications, the procedure skips all other lines until the string 'END' or eof(sanfile) (17, 543) is encountered. The 'END' indicates the end of the FSM instance specification. If procedure tranfsm encounters end of file before the string 'END', the procedure error(err7)(17, 547) is called.

Whenever the Transformation process encounters a syntax error, the procedure error will be called. An error message is passed to the procedure error via a predefined integer constant. In the above example, the procedure error(err7) is called. err7 represents the error message number 7. In procedure error (16, 261), the error message number 7 means 'missing END at the end of a State Machine' (16, 317-318).

Procedure trancfp (17, 554) transforms all the procedures defined in a CFP instances in a SAN model into a globally compatible procedure cfpfunction (20, 38). The process of transformation is similar to that for a FSM except that procedure trancfp transforms only the local procedure in the FOUTP specification. The details can be seen in the procedure trancfp source listings (17, 554-613).

Procedure trancfs (17, 618) transforms all the procedures defined in the CFS instances in a SAN model into a globally compatible procedure cfsfunction (20, 48). The process of transformation is similar to that for a FSM except that procedure trancfs transforms only the local procedure in the FOUTS specification. The details can be seen in the procedure trancfs source listings (17, 618-677).

11. Procedure resmname

Procedure resmname (17, 681) processes all the DELP, DELS, QUE, CLK and DER components of a SAN model. Since all the components of the above kinds have a predefined function and do not have any user defined procedure, procedure resmname does not have to do any transformation

beside recording all the component names. Procedure resmname first reads the component name by calling readstr (17, 688). Depending on the kind of component, the component name is stored into its corresponding component name linked list (17, 703-710). The head of the DELP, DELS, QUE, CLK and DER linked list is pointed to by delppt, delspt, quept, clkpt and derpt respectively. The procedure then skips all the specification lines until the string 'END' or eof(sanfile) is encountered (17, 714). If procedure resmname encounters end of file before the string 'END', the procedure error(err7) (17, 717) is called.

12. Procedure tranenv

Procedure tranenv (17, 1111) transforms all the procedures and boolean expressions defined in the ENV components of a SAN model into globally compatible procedures and boolean expressions as described in Chapter III.C.2 and Chapter III.C.3. Procedure tranenv first reads the ENV component name by calling readstr (17, 1118). The ENV component name is stored into the ENV component linked list, the head of the linked list is pointed to by envpt (17, 1133). The procedure then skips all the specification until the string 'FUNCTION', 'END', or eof(sanfile) (17, 1137) is encountered.

If procedure tranenv encounters 'END' or eof(sanfile) before 'FUNCTION', then procedure error(err36) (17, 1139) is called to report the error. If tempstr equals 'FUNCTION', the procedure reads another string. If tempstr equals 'procedure', the number of the ENV procedure is increased by one (17, 1151) and the procedure dirtranp (17, 1152) is

called to transform the ENV procedure as described in Chapter III.C.2. If tempstr does not equal either 'procedure' or 'terminal', then procedure error(err50) is called to report the error. Else, procedure tranenv skips all other specifications until the string 'STARTEXP', 'END' or eof(sanfile) is encountered. If tempstr does not equal 'STARTEXP', then procedure error(err37) is called to report an error. Otherwise, it starts the transformation of the expression as described in Chapter III.C.3. The number of the ENV expression is increased by one (17, 1167). The headings of the ENV expression to appear in the function envexpst (20, 83-96) are written into the ENV expression file, envexpstfile (17, 1171-1175). The procedure transexp is called to transform the SAN boolean expression into a globally compatible boolean expression. After the boolean expression transformation, procedure tranenv skips all other specification lines until the string 'END' or eof(sanfile) is encountered (17, 1184). If the procedure encounters end of file before the string 'END', the procedure error(err7) (17, 1187) is called.

13. Procedures tranvar, tranreg and trancon

Procedure tranvar (17, 1416) identifies whether the current variable history instance is a regular variable history or a conditional variable history. Procedure tranvar first reads the variable history instance name by calling procedure readstr (17, 1423). The instance name is temporarily saved in the global variable chist (17, 1428). The procedure then calls procedure readstr to read another string. If tempstr equals 'regular', then the procedure tranreg is called; else, if

tempstr equals 'conditional', then the procedure trancon is called; else, the procedure error(err53) is called (17, 1433-1435). Procedure tranvar then skips all the specification until the string 'END' or eof(sanfile) is encountered (17, 1443). If the procedure encounters end of file before the string 'END', the procedure error(err7) (17, 1444) is called.

Procedure tranreg (17, 1299) is called by procedure tranvar to transform the SAN global variables to PASCAL compatible variables and to establish trace procedure historynum to record the variables trace as described in Chapter III.C.4. Procedure tranreg first initializes the number of processed SAN global variables to be zero (17, 1305). The procedure then reads a string by calling procedure readstr (17, 1307). If tempstr does not equal 'VARIABLES', then procedure error(err59) (17, 1309) is called to report an error. If no error occurs at this point, procedure tranreg will start the Transformation process. The procedure first increases the number of regular variable history instances by one. The procedure then writes the headings (17, 1319-1321) of the regular variable instance into the file historyfile for the procedure historynum (20, 101). Procedure tranreg then calls procedure provar (17, 1327) to transform the SAN global variables into PASCAL global compatible variables; and increases the number of regular variable history instances by one; these steps are repeated until all the SAN global variables are processed which is indicated by a delimiter semicolon (17, 1336). At the end, procedure tranreg writes the string 'end' to

the file historyfile (20, 103) to indicate the end of all the variables traced in a regular variable history instance.

Procedure trancon (17, 1349) is called by procedure tranvar to transform the SAN global variables to PASCAL global compatible variables, to transform the SAN boolean expression into a globally compatible expression and to establish the boolean function conexp and the procedure savehistcon for conditional variables to be traced as described in Chapter III.C.5. Procedure trancon goes through a similar process to generate the procedure savehistcon as if procedure tranreg generates the procedure historynum. Procedure savehistcon is stored in the file conhistfile. After processing all the SAN variables to generate procedure savehistcon, procedure trancon continues to transform the SAN boolean expression in the conditional variable history instance. The procedure reads a string by calling the procedure readstr (17, 1396). If tempstr does not equal 'CONDITION', then procedure error(err54) (17, 1398) is called to report an error. The procedure then writes the headings of the boolean expression to appear in the boolean function conexp in the file conexpfile (17, 1402-1404). The procedure transexp is called to transform the SAN boolean expression into a globally compatible expression and to write it into the boolean function conexp (20, 113). At the end, procedure trancon writes the string 'end' in the file conexpfile (20, 114) to indicate the end of the boolean expression for a conditional variable history instance.

14. Procedure tranexp

Procedure tranexp transforms the SAN boolean expressions of all the regular expression history instances into globally compatible expressions and puts all the PASCAL expressions together in the boolean function cexpst (20, 135) as described in Chapter III.C.6. The boolean function cexpst is stored in a temporary file cexpstfile. Procedure tranexp first reads the expression history instance name (17, 1457) and temporarily stores the name in the global variable chist. The procedure then reads another string. If tempstr equals 'regular', then tranexp continues to read another string, else procedure error(err53) (17, 1465) is called to report an error. If the next string is not 'EXPRESSION' (17, 1470), then procedure tranexp reports an error by calling procedure error(err5).

If no error is encountered at this point, procedure tranexp increases the number of regular expression histories by one. The procedure then writes the heading (17, 1479-1481) into the file cexpstfile for the boolean function cexpst (20, 139). Procedure transexp is then called to transform the SAN boolean expression into a globally compatible expression and to write into the boolean function cexpst (20, 140). Procedure tranexp then writes the string 'end' in the file cexpstfile (20, 142). The procedure then skips all other specification lines until the string 'END' or eof(sanfile) is encountered. If procedure tranexp encounters end of file before the string 'END', then it reports an error by calling procedure error(err7) (17, 1509).

15. Procedure inisetgen

Procedure inisetgen (18, 608) generates the procedure iniset (20, 903) as described in Chapter III.C.7. Within the procedure inisetgen, there is a procedure wsmws (18, 617-656), which is called by procedure inisetgen to write the smword[i] and smsym[i] value of each component (20, 905-914) in the file, inisetfifle. Procedure inisetgen first writes the heading (18, 664-666) of the procedure iniset, which is stored in the file inisetfile. In lines (18, 668-704), procedure inisetgen writes all the corresponding smword[i] and smsym[i] into procedure iniset (20, 905-914). Procedure inisetgen then writes the first and last component name for each kind of component (18, 708-842) into procedure iniset (18, 916-933). Procedure inisetgen continues to write the set of component names of each kind of component (18, 846-854) into procedure iniset (20, 934-942). Finally, procedure inisetgen writes the array of string constants (18, 858-866) into procedure iniset (20, 943-950), and writes the ending part of procedure iniset.

D. Data Input

The Data Input process reads the SAN system specification file to configure the simulated system by establishing the interconnections, the sets of state, inputs, and outputs, and output functions, and the next state functions of the system°s components. This process also establishes the necessary default values for system component and simulator variables. The system specification syntax is also checked

against the SAN syntax as described in Chapter II. The details of the syntax checking will be mentioned as we walk through the implementation of the Data Input process. If the Data Input process encounters any syntax error, the process writes the error message to the error file, named errfile, and stops.

The Data Input process is located in line 867 to line 898 in the SAS main program (1, 867-898). The Data Input process first reads in a character string to identify the kind of component to be processed. If the character string equals 'FSM', 'CFP', CFS', 'DELP', 'DELS', 'QUE', 'CLK', 'DER', or 'ENV', then the procedure subfsm, subcfp, subcfs, subdelp, subdels, subque, subclk, subder, or subenv, respectively, is called to read information for the corresponding component (2, 867-884). The above steps are continued until all the component specifications are processed or the end of the SAN file eof(sanfile), is encountered.

If the character string does not identify one of the nine kinds of components, the character string is examined to see if it denotes a performance trace or initialization instance. If the character string equals 'VARHISTORY', 'EXPHISTORY', or 'INIT', then the procedure provarhist, proexphist, or proindata is called to read the appropriate information from the SAN file (2, 876-885). These steps are continued until all the performance trace instances and the initialization instance are processed (i.e., until the end of the SAN file). If there is any component instance other than that of a performance trace or initialization instance, then an error is reported by calling the procedure error(err33) (2, 894).

In the following subsections, we will describe the implementation of all the major procedures of the Data Input process, namely: subfsm (3, 2768), subcfp (3, 2847), subcfs (3, 2901), subdelp (3, 2955), subdels (3, 3000), subque (3, 3050), subclk (2, 3115), subder (3, 3167), subenv (3, 3204), provarhist (4, 1006), proexphist (4, 1046), and proindata (5, 745).

1. Procedure subfsm

Procedure subfsm (3, 2768) governs the sequence for processing each FSM component specification. Procedure subfsm will call a set of procedures, such as prosm, pros, proxp, proxs, prozp, prozs, profns, profsmfoutp, and profsmfouts to process different lines of FSM component specification. The function and the implementation details of these procedures will be described after the description of the procedure subfsm. We point out now that at the end of each of these procedures, procedure readstr is called to read a character string for the next procedure.

Procedure subfsm starts out by calling procedure prosm (3, 2773) to process the first line of the FSM specification. Procedure subfsm then expects the next line to be the state set specification. If the next character string, tempstr, equals 'S', then procedure pros (3, 2775) is called to process the 'Sline', as mentioned in the SAN syntax diagram in the Appendix; else, the procedure error(err2) is called to report a syntax error. The next line should be the default initial state specification. If tempstr equals 'defsinit', then procedure subfsm

processes the 'defsinitline'. Procedure readstr (3, 2783) is called to read the default initial state value. The default initial state value is checked as to whether it is a subset of the state set by calling the procedure ckvalidset (3, 2785). If the default initial state value is not a subset of the state set, then procedure error(err65) is called to report an error (3, 2786); otherwise, the default initial state value is inserted in the FSM component current state variable, fsm[tempsm].cs (3, 2789). Procedure readstr (3, 2792) is called to read a character string for the next line.

If tempstr equals 'XP', then procedure proxp(fsmym) (3, 2798) is called to process all the 'XPline' of the FSM component specification; otherwise, procedure error(err3) is called to report an error. Since the FSM specification has the option not to define any static input, pulsed output or static output, the next line may be a XS, XP, or ZS line. If tempstr equals 'XS', then procedure proxs(fsmsym) (3, 2800) is called to process all the 'XSline'. If tempstr equals 'ZP', then procedure prozp(fsmsym) (3, 2805) is called to process all the 'ZPline'. If tempstr equals 'ZS', then procedure prozs(fsmsym) (3, 2808) is called to process all the 'ZSline'.

After processing all the input and output specifications, procedure subfsm starts to process the next state function and the output function. For an FSM, there must be a next state function, FNS. If tempstr equals 'FNS', then procedure profns (3, 2811) is called to process the FNS specification, otherwise procedure error(err5) is called. The FSM component may or may not have the FOUTP and FOUTS

defined. If tempstr equals 'FOUTP', then procedure profsmfoutp (3, 2817) is called to process the FOUTP specification. If tempstr equals 'FOUTS', then procedure profsmfouts (3, 2818) is called to process the FOUTS specification.

At the end, procedure subfsm expects the specification of the default execution time. If tempstr equals 'deftexec', then procedure prodeft(4) (3, 2821) is called to read the default execution time; otherwise, procedure error(err40) is called to report an error. The default execution time is assigned to the variable, fsm[tempsm].texec (3, 2824). After the default execution time, procedure subfsm expects an 'END' followed by a semicolon. If that is true, procedure subfsm finishes processing a FSM component; otherwise, procedure error(err7) (3, 2831) is called.

The remaining parts of this sub-section will describe the function and the implementation details of procedures prosm, pros, proxp, proxs, prozp, prozs, profns, profsmfoutp, and profsmfouts, which are called by procedure subfsm to read the FSM specification.

a. Procedure prosm (kind : smtype) (3, 1096): processes the specification of the first line of the nine kinds of components. Procedure prosm first checks if there is a blank character between the specification of the kind and the component name. If it is not a blank character, then procedure error(err24) is called. Procedure prosm then calls procedure readstr and skipdel to read the name of the component and skips all the blank characters following the component name. Procedure cksmname(tempstr, tempsm, match) is then called to check if

the component name, identified by tempstr belongs to the set of component names identified by the Transformation process. If the component name belongs to those component names, the variable match is set true and the component's corresponding enumerated name is assigned to the variable tempsm (3, 1104). If the component name does not belong to those component names, procedure error(err34) is called (3, 1108). Procedure prosm continues to read another character string by calling procedure readstr. If tempstr does not equal 'interlace', then procedure error(err30) is called; otherwise, procedure readstr (3, 1113) is called to read a character string. The character string will be used by the parent procedure, who called the procedure prosm, to identify the next line of the specification. In this case, the parent procedure is subfsm, which called procedure prosm to process the first line of a FSM component specification. The next character string read by the procedure readstr (3, 1113), at the end of the procedure prosm, should be the state set character string, 'S'.

b. Procedure pros (3, 1120): processes the 'Sline' specification of the FSM components. Procedure pros first checks if the last character read by the procedure readstr is a blank character. If the last character, c (3, 1125), is a blank character, then procedure readstr is called to skip to the next delimiter. Procedure pros checks if the delimeter is a colon (3, 1129); if c does not equal ':', then procedure error(err20) is called.

Procedure pros continues to read the state set values by repeatedly calling procedure readstr (3, 1135); each value is stored in a linked

list pointed to by the header, fsm[tempsm].S (3, 1145-1148). This process is repeated until the delimiter is a semicolon (3, 1151). If the number of state set values is greater than nSmax, which is 100, then procedure error(err9) is called. At the end, procedure readstr is called to read a character string for the next line of the specification.

c. Procedure proxp(kind:smtype) (3, 1164): processes the 'XPline' specification of all the components. The local variable kind (3, 1164) is passed to procedure proxp to identify the kind of component being processed. In the beginning, the local variable i is initialized to zero to indicate the number of processed XP inputs. In lines (3, 1179-1181), the number of processed XP input is increased by one, the XP number is read by the read statement; and if the XP number, xpnum, does not equal i then procedure error(err11) is called. Procedure readstr is called. If tempstr does not equal 'FROM' or 'UNCONNECTED', then procedure error(err31) is called. If tempstr equals 'FROM' and the delimiter is a blank or tab character, then procedure readstr (3, 1191) is called to read the source component name. Procedure cksmname(tempstr, sourcesm, match) (3, 1194) is called to check if tempstr belongs to the predefined component names. If it does, procedure cksmname returns with the variable match equal to true and the variable sourcesm equal to the corresponding enumerated name of the component.

In lines (3, 1200-1214), if the source is unconnected, then procedure skipdel (3, 1200) is called to skip to the next delimiter;

else, the regular pulsed output specification, ZP.n, is processed from line (3, 1203) to line (3, 1212). In line (3, 1213) the delimiter colon, which separates the source input and the input set specification, is checked.

In lines (3, 1217-1290), the XP input values are read and stored into the XP[i] linked list of the corresponding component. An XP input value is read by calling procedure readstr (3, 1218). If the delimiter between two XP input values is not a comma or a semicolon, then procedure error(err21) (3, 1220) is called; or if the first character in the XP input value is a delimiter, then procedure error(err10) is called. Procedure new(stringpt) (3, 1223) is called to create a new stringset (1, 206) to store the XP input value (3, 1225-1226). Depending on the kind of component, the stringset is inserted into its corresponding linked list. In the case of a FSM component, the new stringset pointed to by the pointer stringpt is inserted at the end of the linked list fsm[tempsm].XP[xpnum] (3, 1230-1236), where tempsm is the component enumerated name and xpnum is the current XP number. This process is repeated until the delimiter after the XP input string is a semicolon (3, 1290). At the end, in lines (3, 1295-1322) a new stringset with a null character, '-', is created and inserted into the corresponding XP input set. In line (3, 1325), procedure readstr is called; if tempstr equals 'XP', which means another 'XPline' needed to be processed. The procedure proxp will go back to line (3, 1175) to process another XP input. If tempstr does not equal 'XP', the number of the processed XP inputs are recorded (3, 1328-1335). This ends the procedure proxp.

d. Procedure proxs(kind:smtype) (3, 1344): processes the 'XSline' specification of all the components. The 'XSline' specification is the same as the 'XPline' specification, except that the 'XP' in the 'XPline' specification is replaced by 'XS'. In the same manner, procedure proxs is the same as procedure proxp, except that the former is used to process 'XP' inputs and the latter is used to process 'XS' inputs.

e. Procedure prozp(kind:smtype) (3, 1502): processes the 'ZPline' specification of all the components. The local variable kind (3, 1502) is passed to procedure prozp to identify the kind of component being processed. In the beginning, the local variable i is initialized to zero to indicate the number of processed ZP outputs.

The repeat loop (3, 1515-1757) processes one ZP output. At first, the number of the processed ZP output is increased by one; and the ZP number, zpnum is read (3, 1519-1523). If the ZP number is not in sequence, then procedure error(err13) is called. The procedure readstr (3, 1527) is called to read the character string 'TO' or 'UNCONNECTED'. Lines (3, 1528-1543) check the character string and the delimiter after the character string. In line (3, 1547), the local variable j is set to zero to indicate the number of destinations from this ZP output (fanout).

The second repeat loop (3, 1549-1628), which resides within the first repeat loop (3, 1515-1757), reads the destination names and stores them into the component's data structure. If tempstr does not equal 'UNCONNECTED', then it reads the destination name (3, 1553-1577). In line

(3, 1580) the local variable, j, indicating the number of destinations is increased by one. The destination component kind:destkind, name:destname, and pulsed input number:xpnum are recorded in the data structure of the processing component (3, 1581-1617). The second level repeat loop is repeated until the delimiter following the destination name is not a comma (3, 1618).

Lines (3, 1622-1631) record the number of fanouts for the particular pulsed output. Procedure prozp then checks if the delimiter between the destination name and the output set is a colon. If it is a colon, then procedure error(err20) is called.

The third repeat loop (3, 1638-1717), which also resides within the first repeat loop, reads the pulsed output set values and stores them into the component's data structure. The procedure readstr (3, 1639) is called to read an output string. If the delimiter following the output string is not a comma or semicolon, then procedure error(err21) is called; if the output string is a delimiter, then procedure error(err10) is called. Procedure new(stringpt) (3, 1649) is called to create a new stringset to record the output string (3, 1647-1648). Depending on the kind of the component being processed, the new stringset is inserted at the end of the corresponding linked list. In the case of a FSM component, the new stringset pointed to by the pointer stringpt (3, 1646) is inserted at the end of the linked list pointed to by the pointer, fsm[tempsm].zp[zpnum] (3, 1652-1658), where tempsm is the component enumerated name and zpnum is the current ZP number. This repeat loop (3, 1638-1717) is repeated until the delimiter

after the ZP output string is a semicolon (3, 1717). At the end, in lines (3, 1721-1752) a new stringset with a null character, '-', is created and inserted into the corresponding ZP output set. In line (3, 1756), procedure readstr is called; if tempstr equals 'ZP', which means another 'Zline' needs to be processed, the first repeat loop (3, 1515-1757) is repeated. If tempstr does not equal 'ZP', the number of the processed ZP outputs is recorded (3, 1760-1768). This ends the procedure prozp.

f. Procedure prozs(kind:smtype) (3, 1778): processes the 'ZSline' specification of all the components. The 'ZSline' specification is the same as the 'ZPline' specification, except that the ZP in the 'ZPline' specification is replaced by 'ZS'. In the same manner, procedure prozs is the same as procedure prozp, except that the former is used to process 'ZP' inputs and the latter is used to process the 'ZS' inputs.

g. Procedure profns (3, 2301): processes the next state function (FNS) specification of the FSM component. Procedure profns first checks if the delimiter following the string 'FNS' is a colon. If the delimiter is not a colon, then procedure error(err20) (3, 2312) is called. Procedure readstr (3, 2315) is called to read the type of FNS specification. If tempstr equals 'list', then the FNS list specification will be processed (3, 2319); else, if tempstr equals 'procedure', then the FNS procedure specification will be processed; otherwise, procedure error(err50) is called (3, 2448).

Lines (3, 2319-2439) processes the FNS list specification. The type of FNS specification is first recorded (3, 2324). Procedure

readstr (3, 2326) is called to read the string for the current state. Procedure profns then starts with the repeat loop (3, 2329-2439) to process a FNS transition. Whether the string for the current state is an element of the FSM component state set is checked by calling the procedure ckvalidset(fsm[tempsm].S,tempstr) (3, 2331). If tempstr is not an element and does not equal don't care, '*', then procedure error(err65) is called to create a record element to record the string for the current state (3, 2335-2337). Procedure profns then checks if the delimiter following the current state value is a slash, '/'. If the delimiter is not a slash, then procedure error(err19) (3, 2340) is called.

Line (3, 2343) checks if there is any pulsed input string to be read. If there is, the loop (3, 2348-2358) will read in the string for the pulsed inputs and store them in the array, fnspt^.xp[i] (3, 2355). The loop is repeated until the delimiter after the pulsed input string is not a comma (3, 2358). If the number of pulsed input strings read does not equal the number of pulsed inputs specified for the component instance, then procedure error(err15) is called.

Line (3, 2363) checks if there is any static input string to be read. If it is true, the loop (3, 2371-2379) will read in the string for the static inputs and store them in the array, fnspt^.xs[i] (3, 3275). The loop is repeated until the delimiter after the static input string is not a comma (3, 2379). If the number of static input strings read does not equal the number of static inputs specified for the component instance, then procedure error (err15) is called.

Procedure rdimply (3, 2385) is called to read the special character string, '=>'. If tempstr does not equal '=>', then procedure error(err16) (3, 2388) is called. Procedure readstr (3, 2392) is called to read the next state string. Lines (3, 2401-2417) check if the next state string is a string constant or used as a local input or output variable to identify the string by calling the procedure ckst(st,tempsym,num) (3, 2404). If the next state string is a string constant, the string constant is stored in the variable, fnspt^.nexts.sval (3, 2411). If the next state string is a local input or output variable, the kind of local input or output (cxp, cxs, czp, czs) is stored in the variable, fnspt^.nexts.id and the input or output number is stored in the variable, fnspt^.nexts.num. At the end, the record for the FNS transition pointed to by the pointer, fnspt, is inserted into the FSM component's FNS entry linked list, which is pointed to by the pointer, fsm[tempsm].fnsfirst (3, 2421-2429). Procedure readstr (3, 2435) is called to read another string. If tempstr does not equal 'END', then the loop (3, 2329-2439) is repeated to process another transition; otherwise, the list specification is done.

Lines (3, 2443-2448) process the FNS procedure specification. Since the procedure had been extracted and transformed during the Transformation process; all the procedure profns has to do in the Data Input process is to record the type of the FNS specification (3, 2445) and the procedure number for the FSM component kind (3, 2447), which is equal to the case index. This procedure will be referenced by the SAS. The procedure skipproc is called to skip to the end of the procedure (3,

2447). Procedure readstr (3, 2451) is called to read another string for the next line of the specification.

h. Procedure profsmfoutp (3, 2134): processes the pulsed output function (FOUTP) specification of the FSM component. The FOUTP specification also has either the list or procedure type. The format for the procedure type is the same as the FNS procedure. The format for the list type is similar to FNS list, except that in each transition line the FNS has only one character string for the next state value while the FOUTP may have multiple character strings for the next pulsed output values. The implementation of procedure profsmfoutp is similar to the procedure profns except for the above difference. Readers can refer to the explanation of procedure profns to understand the implementation of procedure profsmfoutp (3, 2134-2289).

i. Procedure profsmfouts (3, 2013): processes the static output function (FOUTS) specification of the FSM component. The format in specifying the FOUTS is similar to FNS and FOUTP. The details can be referred to in Chapter II.B. The approach to implement procedure profsmfouts is similar to the implementation of procedure profns. Readers can refer to the explanation of procedure profns to understand the implementation of procedure profsmfouts (3, 2013-2122).

2. Procedure subcfp and subcfs

Procedure subcfp (3, 2847) governs the sequence of processing the CFP component specification. Procedure subcfp starts out by calling procedure prosm (3, 2855) to process the first line of the CFP

specification. If the next character string is 'XP', then procedure proxp(cfpsym) (3, 2858) is called to process all the 'XPline' of the CFP component specification; else, procedure error(err3) is called. Since the CFP specification has the option not to define any static input, the next line may be a XS or ZS line. If tempstr equals 'XS', then procedure proxs(cfpsym) (3, 2863) is called to process all the 'XSline'. If tempstr equals 'ZP', then procedure prozp(cfpsym) (3, 2866) is called to process all the 'ZPline'.

After processing all the input and output specifications, procedure subcfp starts to process the pulsed output function. If tempstr equals 'FOUTP', then procedure procfpfoutp (3, 2871) is called to process the FOUTP specification, else procedure error(err6) is called. The details of the implementation of procedure procfpfoutp in lines (3, 2603-2758), are similar to the implementation of procedure profsmfoutp (3, 2134-2289). At the end, procedure subcfp expects the specification of the default execution time. If tempstr equals 'deftexec', then procedure prodeft(r) (3, 2876) is called to read the default execution time, otherwise procedure error(err40) is called. The default execution time is assigned to the variable, cfp[tempsm].texec (3, 2879). Procedure subcfp then expects an 'END' followed by a semicolon. If tempstr equals 'END' and the delimiter is a semicolon, procedure subcfp finishes processing a CFP component; else, procedure error(err7) (3, 2883) is called.

Procedure subcfs (3, 2901) governs the sequence of processing the CFS component specification. The implementation details of procedure

subcfs are similar to procedure subcfp, which had been explained in the above paragraphs.

3. Procedure subdelp and subdels

Procedure subdelp (3, 2955) governs the sequence of processing the DELP component specification. Procedure subdelp starts out by calling procedure prosm(delpsym) (3, 2963) to process the first line of the DELP specification. Procedure subdelp then processes the XP input. If tempstr equals 'XP', then procedure proxp(delpsym) (3, 2966) is called to process the 'XPline' of the DELP component specification; else, procedure error(err3) is called. The next step is to process the 'ZPline'. If tempstr equals 'ZP', then procedure prozp(delsym) (3, 2972) is called to process the 'ZPline' of the DELP component specification, else procedure error(err4) is called. Procedure subdelp starts to record the default delay time by calling procedure prodeft; else, procedure error(err41) is called. The default delay time is saved in the variable, delp[tempsm].tdel. Procedure readstr is called to read another character string. If tempstr does not equal 'END' or the delimiter following the string is not a semicolon, then procedure error(err7) is called to report an error, else, procedure subdelp finishes processing a DELP component.

Procedure subdels (3, 3000) governs the sequence of processing the DELS component specification. The DELS specification is similar to the DELP specification except that XP and ZP in the DELP specification are

replaced by XS and ZS. The implementation of procedure subdels is also similar to the implementation of procedure subdelp.

4. Procedure subque

Procedure subque (3, 3050) governs the sequence of processing the QUE component specification. Procedure subque starts out by calling procedure prosm(quesym) (3, 3058) to process the first line of the QUE specification. Procedure subque then processes the XP inputs. If tempstr equals 'XP', then procedure proxp(quesym) (3, 3061) is called to process the 'XPline' of the QUE component specification; otherwise, procedure error(err3) (3, 3062) is called. The next step is to process the 'ZPline'. If tempstr equals 'ZP', then procedure prozp(quesym) (3, 3067) is called to process the 'ZPline'; else, procedure error(err4) (3, 3068) is called. After processing ZP, procedure subque expects a 'ZSline'. If tempstr equals 'ZS', then procedure prozs(quesym) (3, 3074) is called to process the 'ZSline'; else, procedure error(err4) (3, 3075) is called.

Procedure subque sets the initial state, size, queue pointer and static output of the queue (3, 3079-3082). Procedure subque continues to process the default enqueue and dequeue time. If tempstr equals 'deftenq', then procedure prodeft(r) is called to read the default enqueue time; else, procedure error(err42) is called. The default enqueue time is assigned to the variable que[tempsm].tenq (3, 3088). Procedure readstr (3, 3090) is called to read the string 'deftdeq'. If tempstr equals 'deftdeq', then procedure prodeft(r) is called to read the

default dequeue time; else, procedure error(err43) (3, 3092) is called. The default dequeue time is assigned to the variable que[tempsm].tdeq (3, 3098). Procedure readstr is called to read the string 'END'. If tempstr does not equal 'END' or the delimiter following 'END' is not a semicolon, then procedure error(err7) is called; otherwise procedure subque is finished.

5. Procedure subclk

Procedure subclk (3, 3115) governs the sequences of processing the CLK component specification. Procedure subclk starts out by calling procedure prosm(clksym) (3, 3123) to process the first line of the CLK specification. Procedure subclk then processes the XP input. If tempstr equals 'XP', then procedure proxp(clksym) (3, 3126) is called to process the 'XPline'; else, procedure error(err3) (3, 3127) is called. The next step is to process the 'ZPline'. If tempstr equals 'ZP', then procedure prozp(clksym) (3, 3132) is called to process the 'ZPline'; otherwise, procedure error(err4) (3, 3133) is called. After processing ZP, procedure subclk expects a 'ZSline'; else, procedure error(err4) (3, 3138) is called.

Procedure subclk sets the initial state and static output of the CLK component (3, 3142-3143). Procedure subclk continues to process the default clock period. If tempstr equals 'deftclk', then procedure prodeft(r) is called to read the default clock period; else, procedure error(err44) (3, 3146) is called. The default clock period is assigned to the variable clk[tempsm].tclk (3, 3148). Procedure readstr

is called to read the string 'END'. If tempstr does not equal 'END' or the delimiter following 'END' is not a semicolon, then procedure error(err7) is called; else, procedure subclk is finished.

6. Procedure subder

Procedure subder (3, 3167) governs the sequence of processing the DER component specification. Procedure subder starts out by calling procedure prosm(dersym) (3, 3174) to process the first line of the DER specification. Procedure subder then processes the XS input. If tempstr equals 'XS', then procedure proxs(dersym) (3, 3177) is called to process the 'XSline'; else, procedure error(err3) (3, 3178) is called. The next step is to process the 'ZPline'. If tempstr equals 'ZP', then procedure prozp(dersym) (3, 3183) is called to process the 'ZPline'; else, procedure error(err4) (3, 3184) is called. At the end, procedure subder checks for the string 'END' following with a semicolon. If tempstr does not equal 'END' or the delimiter following 'END' is not a semicolon, then procedure error(err7) is called to report error; else, procedure subder is finished.

7. Procedure subenv

Procedure subenv (3, 3204) governs the sequence of processing the ENV component specification. Procedure subenv starts out by calling procedure prosm(envsym) (3, 3214) to process the first line of the ENV specification. Since the specification of the inputs and outputs are optional in an ENV component, the next line may be an XP, XS, ZP or ZS

line. First, it checks if tempstr equals 'XP'; then procedure proxp(envsym) (3, 3217) is called to process all the 'XPline'. Second, it checks if tempstr equals 'XS'; then procedure proxs(envsym) (3, 3218) is called to process all the 'XSline'. Third, it checks if tempstr equals 'ZP'; then procedure prozp(envsym) (3, 3220) is called to process all the 'ZPline'. Fourth, it checks if tempstr equals 'ZS'; then procedure prozs(envsym) is called to process all the 'ZSline'.

After processing all the input and output specifications, procedure subenv starts to process the ENV function. If tempstr equals 'FUNCTION' (3, 3224), then it processes the ENV function specification (3, 3226-3246); else, procedure error(err36) (3, 3247) is called. To process the ENV function specification, procedure readstr (3, 3229) is called to read the type of ENV function. If tempstr equals 'terminal', then the type of ENV function is recorded in the variable env[tempsm].func (3, 3222); else, if tempstr equals 'procedure', then the ENV function variable is recorded as procedure type (3, 3240); the number of ENV procedure is increased by one; and the procedure specification is skipped by calling procedure skipproc; otherwise, procedure error(err51) (3, 3243-3244) is called.

Procedure readstr (3, 3250) is called to read another character string. If the component has static outputs (3, 3252), then the character string should be 'defzsinit'. If tempstr equals 'defzsinit', then procedure prodefzs(envsym.tempsm) (3, 3255) is called to read in the default static output values; else, procedure error(err45) is called. Procedure subenv continues by reading in the default execution

time. If tempstr equals 'deftexec', then procedure prodeft(r) (3, 3261) is called to read the default execution time; else, procedure error(err40) is called. The default execution time is assigned to the variable env[tempsm].texec (3, 3265). Procedure readstr (3, 3267) is called to read another character string. If tempstr equals 'mulpulsecheck', then procedure readstr (3, 3273) is called again to read another string. If tempstr equals 'false', then the variable env[tempsm].mulpulsecheck is set to false else the variable is set to true (3, 3274-3277). If tempstr does not equal 'mulpulsecheck', then by default the variale env[tempsm].mulpulsecheck is set to false (3, 3272).

Procedure subenv starts to process the start expression, STARTEXP, specification. If tempstr equals 'STARTEXP', then procedure skipexp (3, 3285) is called to skip to the end of the expression (since the expression had been transformed in the Transformation process); otherwise procedure error(err37) is called. The number of the ENV start expression is increased by one, and stored in the variable env[tempsm].envexpno (3, 3289-3290). Procedure readstr (3, 3293) is called to read the string 'STATREXPCHECK'. If tempstr equals 'STARTEXPCHECK', then procedure readstr (3, 3299) is called again to read another string. If tempstr equals 'never', then the variable env[tempsm].checkopt is set to never; else, if tempstr equals 'everyevent', then the variable is set to everyevent; else, if tempstr equals 'everytimechange', then the variable is set to everytimechange, else procedure error(err39) is called (3, 3300-3303). If tempstr does not equal 'STARTEXPCHECK', then procedure error(err38) (3, 3305) is

called. Procedure readstr (3, 3306) is called to read 'END'. If tempstr does not equal 'END', or the delimiter following 'END' is not a semicolon, then procedure error(err7) (3, 3308) is called. At the end all the pulsed outputs of the ENV component are set to null, '-'.

8. Procedure provarhist, provarreg and provarcon

Procedure provarhist (4, 1006) processes the variable history (VARHISTORY) specification. It checks if the VARHISTORY is of type regular or conditional, then procedure provarreg or provarcon, respectively, is called to process the specification. Procedure provarhist starts out to check if there is a blank character after the string 'VARHISTORY'. If it is not a blank character, then procedure error(err24) (4, 1015) is called. Procedure readstr (4, 1019) is called to read the variable history instance name. The instance name is temporarily saved in the local variable histname (4, 1022). Procedure provarhist then checks for a colon after the instance name (4, 1025). Procedure readstr (4, 1029) is called again to read another string. If tempstr equals 'regular', then procedure provarreg(histname) is called to process the regular variable history instance; else, if tempstr equals 'conditional', then procedure provarcon(histname) is called to process the conditional variable history instance, else procedure error(err53) (4, 1035) is called.

Procedure provarreg(histname:string) (4, 908) processes the regular variable history instance specification. Procedure provarreg starts out to create a new record for the regular variable history instance by

calling procedure new(regpt) (4, 915). The record is pointed to by pointer regpt. The details on the record elements can be seen in the data structure file (1, 529-538). The instance name is saved in the element regpt^.histname, and the pointer to the next instance record is set to nil (4, 919-920). The current regular variable history instance record is added to the end of the regular variable history instance linked list (4, 926-937). The header of the linked list is pointed to by the pointer, varhistregpt (4, 924). The local variable temp is set to point to the current regular variable history instance record (4, 940). The integer, i, identifying the number of regular variable history instances is saved in the element, temp^.num. Procedure provar(temphist) (4, 946) is called to read in the set of variable names and create a linked list for the set of variables; procedure provar(temphist) also returns a linked list pointer, temphist, pointing to the beginning of the variable linked list. The linked list is saved by assigning temphist to the current regular variable history instance record element, temp^.varstat (4, 949).

Procedure readstr (4, 952) is called to read the string 'DHISTORY'. If tempstr does not equal 'DTHISTORY', then procedure error(err51) (4, 958) is called. The time interval for 'DTHISTORY' is read by calling the procedure read(sanfile,r) (4, 962). The time interval is stored in the element, temp^.dthist (4, 968). Procedure readstr (4, 970) is called to read the delimiter semicolon. If it is not a semicolon, then procedure error(err22) is called. Procedure readstr (4, 976) is called to read the string 'CHECKOPT'. If tempstr equals 'CHECKOPT', then

procedure readstr (4, 982) is called again to read the check option string. If tempstr equals 'never', then the regular variable history instance record element, temp^.checkopt, is set to never; else, if tempstr equals 'everyevent', then temp^.checkopt is set to everyevent; else, if tempstr equals 'everytimechange', then temp^.checkopt is set to everytimechange; else procedure error(err39) is called (4, 982-989). Procedure readstr (4, 794) is called to read 'END'. If tempstr does not equal 'END' and the delimiter following 'END' is not a semicolon, then procedure error(err7) is called; else, procedure provarreg is finished.

Procedure provarcon(histname:string) (4, 822) processes the conditional variable history instance specifications. Procedure provarcon starts out by creating a new record for the conditional variable history instance by calling procedure new(conpt) (4, 830). The record is pointed to by pointer conpt. The details on the record elements can be seen in the data structure file (1, 542-549). The instance name is saved in the element conpt^.histname and the pointer to the next instance record is set to nil (4, 833-834). The current conditional variable history instance record is added to the end of the conditional variable history instances linked list (4, 838-851). The header of the linked list is pointed to by the pointer, varhistconpt (4, 838). The local variable temp is set to point to the current conditional variable history instance record (4, 854). The integer i identifying the number of conditional variable history expressions is saved in the element, temp^.num (4, 857). Procedure provar(temphist)

(4, 860) is called to read in the set of variable names and create a linked list for the set of variables; procedure provar(temphist) also returns a linked list pointer, temphist, pointing to the beginning of the linked list. The linked list is saved by assigning temphist to the current conditional variable history instance record element, temp^.varstat (4, 863).

Procedure readstr (4, 866) is called to read the string 'CONDITION'. If tempstr equals 'CONDITION', then procedure skipexp is called to skip to the end of the boolean expression; otherwise procedure error(err54) is called (3, 872-873). The boolean expression is skipped because the boolean expression had been transformed by the Transformation process. Procedure readstr (4, 878) is called to read the string 'CHECKOPT'. If tempstr equals 'CHECKOPT', then procedure readstr (4, 884) is called again to read the check option string. If tempstr equals 'never', then the conditional variable history instance record element, temp^.checkopt, is set to never; else, if tempstr equals 'everyevent', then temp^.checkopt is set to everyevent; else, if tempstr equals 'everytimechange', then temp^.checkopt is set to everytimechange; else, procedure error(err39) is called (4, 882-891). Procedure readstr (4, 896) is called to read 'END'. If tempstr does not equal 'END', and the delimiter following 'END' is not a semicolon then procedure error(err7) is called (4, 897); else, procedure provarcon is finished.

9. Procedure proexphist

Procedure proexphist (4, 1046) processes the expression history (EXPHISTORY) specification. Procedure proexphist starts out to check if there is a blank character after the string 'EXPHISTORY'. If it is not a blank character, then procedure error(err24) (4, 1058) is called. Procedure readstr (4, 1061) is called to read the expression history instance name. The instance name is temporarily saved in the local variable histname (4, 1064). Procedure proexphist then checks for a colon after the instance name (4, 1067). Procedure readstr (4, 1071) is called again to read another string. If tempstr equals 'regular', then the expression history instance specification will be processed (4, 1076-1160); otherwise procedure error(err53) is called to report an error.

In line (4, 1076), procedure new(exppt) is called to create a new record for the expression variable history instance. The details on the record elements can be seen in the data structure file (1, 552-561). The instance name is saved in the element exppt^.histname; the pointer to the next instance record is set to nil; and the pointer to the expression status record, exppt^.expstat, is also set to nil (4, 1079-1081). The current expression history instance record is added to the end of the expression history instance linked list (4, 1085-1098). The header of the linked list is pointed to by the pointer, exphistpt (4, 1085). The local variable temp is set to point to the current expression history instance record (4, 1111). The integer i identifying the number of the expression history instance is saved in the element,

temp .num (4, 1104). Procedure readstr (4, 1107) is called to read the string 'EXPRESSION'. If tempstr equals 'EXPRESSION', then procedure skipexp (4, 1109) is called to skip to the end of the boolean expression; else, procedure error(err55) is called. The boolean expression is skipped because it had been transformed by the Transformation process into a global boolean expression. The remaining lines of procedure proexphist process the DTHISTORY and the CHECKOPT specification, which is the same as that of procedure provarreg (4, 908). Readers can refer to Chapter III.D.7 to follow the explanation of procedure provarreg.

10. Procedure proindata

Procedure proindata (5, 745) processes the initialization specification. In the initialization instance, users are allowed to initialize the simulation beginning and ending time (tbeg and tend), the seed value for the random number generator (seed), and to set the variable, mulpulsecheck, to indicate whether multiple pulses arriving simultaneously at a component are trapped or not.

The loop (5, 757-822) allows the initialization of the above variables to appear in any order. The loop is repeated until the string 'END' is encountered. This loop does not detect multiple initialization of variables, but procedure proindata leaves only the last initializiation value of the variable.

Procedure readstr (5, 758) is called to read a string. If tempstr equals 'tbeg', then procedure read(sanfile,lr) is called to read the tbeg

value (5, 762-771); else, if tempstr equals 'tend', then procedure read(sanfile,lr) is called to read the tend value (5, 777-786); else, if tempstr equals 'seed', then procedure read(sanfile,i) is called to read the seed value; else, if tempstr equals 'mulpulsecheck', then the boolean value is read to assign to the variable, mulpulsecheck (5, 803-814); else, if tempstr equals 'END', then procedure proindata finishes processing the initialization instance; else procedure error(err7) is called.

E. Initialization

The Initialization process initializes the user defined system to some predefined states, in which the system is ready to be executed in the SAS environment. The Initialization process is located in lines (900-908) in the SAS main program (2, 900-908). There are two kinds of initialization processes, namely New Initialization and Restart Initialization. Right after the Data Input process, the user terminal will display a line to ask if New Initialization or Restart Initialization is to be invoked (2, 900). If Restart Initialization is to be invoked, then procedure reset(rsfile) (2, 904) is called to open the restart initialization file, rsfile, to be read; procedure reload(rsfile) (2, 905) is called to reload the system status from rsfile file; and procedure close(rsfile) is called to close the rsfile file (2, 906). Else, procedure iniload (2, 908) is called to start the New Initialization process.

1. New Initialization process

New Initialization process assumes that all of the system components are in their default initial states and default initial static outputs as defined in the SAN system specification. All the pulsed inputs and outputs are assigned with a string null, '-'. Basically the New Initialization process does the following:

a) Maps the initial state of all FSM, QUE, and CLK instances into their corresponding static outputs.

b) Loads all the static outputs of all FSM, CFP, CFS, DELS, QUE, CLK and ENV instances into their corresponding destination static inputs. If the destination instance is a DER, then sets the current state of the DER instance equal to its current static input (the current state of a DER instance always equals the static input of the DER except at the moment at which a static input transition takes place which is supposed to be instantaneous).

c) Checks if the current static outputs of all the CFS components are equal to the FOUTS mapping of their current static inputs, and the current static outputs of all the DELS components equal to their current static inputs. If they are equal, then the system is initially stable else the system is initially not stable.

d) Initializes all the regular variable history instance and expression history instance next data collection times to tnow.

e) Finally, control is given to the System Executive to start the system execution.

Procedure iniload (7, 890) implements the New Initialization process. Procedure iniload starts out to set tnow equal to TINIT, the beginning time of the initialization period (7, 904). Procedure iniload then maps the initial state of all the FSM components into their corresponding static outputs and loads the static outputs into their destinations. Line (7, 909) checks if the variable fsml, representing the last FSM component, equals fsm0. If fsml equals fsm0, which means there is no user defined FSM component, then the FSM component initialization process is skipped, else the for loop (910-933) initializes all the FSM components, starting from the first FSM component fsmf to the last FSM component fsml, as follows:

a) sets the next state value equal to the current state (7, 914),

b) executes the Static Output Function (FOUTS) to map the current state into current static outputs (7, 915-918).

c) initializes all the pulsed inputs and outputs to null (7, 920-926),

d) loads all the static outputs into their corresponding distination inputs (7, 927-931).

In a similar manner, lines (7, 938-952), the initial state, static output, pulsed inputs and pulsed outputs of all the QUE component are initialized. Lines (7, 957-970) initialize the initial state, static output, pulsed input and pulsed output of all the CLK components. Lines (7, 974-984) initialize the static outputs of all the CFS components.

Lines (7, 988-990) load the static output of all the DELS components into their destinations. Lines (7, 995-1000) initialize the current state and the pulsed output of all the DER components. Lines (7, 1004-1017) initialize the pulsed inputs and outputs of all the CFP components to null, and load the pulsed outputs into their destination components and initialize all the pulsed inputs to null. Lines (7, 1021-1026) initialize the pulsed input and output of all the DELP components to null. Lines (7, 1032-1044) load the static outputs of all the ENV components into their destination components and initialize all the pulsed input to null. Procedure inistab (7, 1046) is called to check if the system is initially stable or not (the details of procedure inistab will be explained in the next paragraph). If the system is stable, the current simulation time (tnow) is set to tbeg. The event file pointer is set to nil (7, 1052). Procedure rexs (7, 1054) is called to reset the loading status of all the static inputs, indicating that none of the static inputs have been changed at the current simulation time. Procedure restatus (7, 1056) is called to reset all the CFS, DELS, and DER components execution status to idle, because their execution status may be set to pend state during the stability test. The SAS defined ENV components (SYSTEM-MONITOR and env0) are initialized as terminal ENVs with check option equal never (7, 1059-1065). The next data collection times of all the regular variable history and expression history instances are set to tnow (7, 1069-1081). This completes the New Initialization process.

Procedure inistab (7, 769) checks if the user defined system is initially stable or not. SAS expects users to define the initial value of the system components such that the current static outputs of all the CFS components are equal to the FOUTS mapping of their current static inputs and the current static outputs of all the DELS components equal their current static inputs. Procedure inistab will execute the FOUTS of all the CFS components to check if there are any discrepancies in the static outputs of the CFS components. Procedure inistab also checks if the current static outputs of all the DELS component equal their current static inputs. If the system is initialized according to the above conditions, there should be no change in the outputs of the CFS components and the current static outputs of the DELS components should equal their current static inputs. In this case, the system is said to be stable, otherwise the system is unstable.

Procedure inistab starts out to assume the system is stable (7, 776). The FOUTS of each CFS component is executed. The next static output values are compared to the current static output values; if they are not equal, then the variable stable is set to false and the name of the unstable component name is recorded (7, 779-795). Procedure inistab continues to check if the static output of each DELS component equals its static input. If any of them is not equal, then the local variable stable is set to false and the unstable component name is recorded. If the user defined system is stable, then the message 'SYSTEM IS STABLE' is printed on the terminal else the message 'SYSTEM IS

UNSTABLE' is printed on the terminal. These complete the stability tests.

2. Restart Initialization process

The Restart Initialization process allows users to continue the execution of the user defined system from the state in which the previous run stopped. This process requires the final status of the total simulation system to be saved at the end of the previous simulation run. The final status of the total simulation system is stored in the restart file, rsfile, according to the following steps:

a) The cs, cxp, cxs, czp, czs, cxsload, texec, execstatus, and the update state records of all FSM components are saved. The update state record contains the nts, ntzp, ntzs and the update time of the component.

b) The cxp, cxs, czp, cxsload, texec, execstatus, and the updadte state records of all CFP components are saved.

c) The cxs, czs, cxsload, texec, execstatus and the update-state record of all CFS components are saved.

d) The cxp, czp, tdel, execstatus and the delay link record of all DELP components are saved. The delay link record contains all the pulsed or transition values which will be loaded into the pulsed output.

e) The cxs, czs, tdel, execstatus and the delay link record of all DELS components are saved.

f) The cxp, czp, czs, tenq, tdeq, enqstatus, deqstatus and the queue status of all QUE components are saved. The queue status contains the state, the size and all the data elements of the QUE component.

g) The cs, cxp, czp, czs, tclk, execstatus, and the update-state of all CLK components are saved.

h) The cs, cxs, czp and execstatus of all DER components are saved.

i) The cxp, cxs, czp, czs, cxsload, texec, execstatus, tstart, and the update-state record of all ENV components are saved.

j) All the elements of the event file are saved.

k) The last executed component kind, component name, event type, tnow, tprev, tfin, and mulpulse value are saved.

In order to continue the system execution from the state in which the previous run stopped, the simulator will start with the Data Input process to reconfigure the system. Then the simulator gives control to the Restart initialization process which in effect follows the above steps to retrieve all the information stored in the restart file, rsfile. At this point all the system components' status and the simulator variables have the same value when the system execution was stopped last time. The simulator then gives control to the System Executive to start system execution again.

Performance data collected during the last system execution are not restored. The linked list records for performance data storage are re-initialized as if no data have been stored. The performance data

collection procedure will start to collect data as if the system execution is started at the beginning.

At the end of a simulation, the simulator asks the user to indicate if the final status of the total simulation system is to be saved (2, 917). If the user responsed yes, then procedure savesysvar (rsfile) (2, 923) is called to save the final status of the total simulation system.

Procedure savesysvar(varoutfile:text) (8, 1201) starts out to check if there is any FSM component in the SAN model. If the last FSM component, fsml, does not equal fsm0, then procedure ssfsm(outfile,name) (8, 1209) is called to save the final status of all the FSM components. The name of the FSM component is passed to procedure ssfsm(outfile,name) by the variable, name, starting from the first FSM component, fsml, to the last FSM component denoted by variable, fsml.

Procedure ssfsm(var outfile:text,name:smname) (8, 744) saves the final status of a FSM component in the output file, outfile. The name of the FSM component is passed to the procedure ssfsm by the variable name. Lines (8, 750-756) show that the execute status (execstatus), the current state (cs), the next state (nts), the current pulsed inputs (cxp[i]), the current static inputs (cxs[i]), the current pulsed outputs (czp[i]), the next pulsed outputs (ntzp[i]), the current static outputs (czs[i]), the next static outputs (ntzs[i]), the next update time (time) and next update task (task) of the FSM component, name, are saved in the file, outfile.

After saving the final status of all the FSM components, procedure savesysvar continues on to save the final status of all the CFP, CFS,

DELP, DELS, DER, CLK, QUE and ENV components in the same manner as the final status of all the FSM components are saved (8, 1210-1225). Procedure sssvar(outfile) (8, 1228) is called to save system variables and the event file.

Procedure sssvar(var outfile:text) (8, 920) saves all the system variables and the event file in the file, outfile. Procedure sssvar starts out to save the event file first. The local variable, tempset, is set to point to the header of the event file. The variable indicating the number of event sets is initialized to zero (8, 929). The while loop (8, 931-935) finds out the number of event sets in the event file and the event set number is saved in the file, outfile. The for loop (8, 938-948) saves all the event file information in the file, outfile. First, the event set time and event count of an event set is saved (8, 940). Second, the component kind, component name, and the event type of an event is saved; this step is repeated until all the event entries of an event set are saved (8, 942-948). The for loop (8, 938-948) is continued until all the event sets are saved. At the end, the current simulation time (tnow), the previous simulation time (tprev), the last executed component kind, component name, and event type are saved in the file, outfile (8, 951-952).

After saving the event file and the system variables, the next data collection times of all the performance trace instances are saved by calling procedure ssstat(outfile) (8, 1231). Procedure ssstate(var outfile:text) (8, 1177) basically saves the next data collection times

for all the regular variable history instances and expression history instances. That completes the process of saving the final status of the total simulation system.

In the initialization process, the simulator asks the user to indicate if it is a new initializiation or restart initialization (1, 900-908). If it is a restart initialization, then procedure reset(rsfile) is called to reset the restart initialization file, rsfile, for reading; procedure reload(rsfile) is called to read the restart initialization file, rsfile, to reconfigure the user defined system from the state in which the previous run stopped; and procedure close(rsfile) is called to close the restart initialization file.

Procedure reload(var infile:text) (8, 1238) basically does the reverse process of procedure savesysvar(varoutfile.text) (8, 1201) to read the restart initialization file and reconfigure the user defined system from the state in which the previous run stopped. At the end, procedure reload also initializes the two SAS defined ENV components, S$SYSTEM-MONITOR and env0 to terminal ENV with check option equal never.

F. System Executive

System Executive, which is the heart of the State Architecture Simulator (SAS), carries out the simulated execution of the user's system. Inside the System Executive, performance traces are saved as appropriate in a performance data file, system status reports are saved in a system data file, and user interaction with the model through the

execution of terminal environment components is carried out through the user's terminal. Simulated system execution error reports are also made to the user via the terminal.

In the following sub-sections, we will first describe the control flow of the System Executive to give readers a general idea of how the System Executive performs system execution based on the event file. The discussion of the control flow of the System Executive will touch several important ideas such as event file management, scheduling and execution of start and update events, performance trace collection, and the time advancement process. The details of the above ideas and their implementations will be explained after the explanation of the control flow of the System Executive.

1. Control flow of System Executive

Before we describe how the System Executive performs system execution based on the event file, we want to describe the general philosophy of the System Executive. SAS is an event driven simulator.

There is a multi-linked list event file to keep track of all the events to be executed. There are two types of events, namely start and update for each component kind, except the DER component which has only a start event. Each start event of a component is scheduled (put) into the event file as a result of an input excitation (arrival of a pulsed input or change of a static input). The update event of a component is put (scheduled) into the event file at the end of the execution of the

component's start event when the component's execution time is greater than zero.

There is a general exception to the above discussion for the ENV component. Each ENV component has a start expression and a start expression check option associated with the component. Whenever the start expression of the ENV component evaluates true at the check option time (everyevent, everytimechange, never), a start event of the ENV component is scheduled into the event file. That is why later on in the control flow of the System Executive the start expression of each ENV component is checked both at the end of each event execution and at the end of the execution of all the events in an event set. Furthermore, when the start event of an ENV component is executed, it can initiate a future start event of the ENV component. To avoid the confusion of having more than one event of the same ENV component in the event file for the same time, the future start event of the ENV component is not scheduled into the event file right away. Rather, the ENV component variables tstart and updatestate.task are set to indicate a start event of the ENV component needs to be scheduled at the time indicated by tstart. Later on in the control flow of the System Executive, each ENV component is checked if a start event of the component needs to be scheduled just before an event is removed to be executed and after all the events in an event set is executed.

As for the performance trace instances, the information indicating the next recording time of a performance trace is stored in the performance trace data structure. Later on in the control flow of the

System Executive, the performance trace instances are checked if the performance trace is to be recorded.

The control flow of the System Executive is shown in Figure 3.7 and is described as follows:

(a) It checks if the current simulation time, tnow, exceeds the ending of the simulation time, tend; or if the system halt variable, syshalt, is true. If one of them is true, the simulator will exit from the System Executive, otherwise it continues at (b).

(b) If there is any event in the event file, then it continues at (c); otherwise it goes to (d).

(c) While there is at least one event waiting to be executed at tnow, it goes to (c.i); otherwise it goes to (d).

i) It first checks if there is any ENV start event to be scheduled at tnow. If that is true, then an ENV start event is scheduled at tnow.

ii) It removes an event-entry from the event file.

iii) Depending on the type of event (start or update) and the kind of component, the proper procedure will be executed.

iv) After the execution of every event, the System Executive will check if any ENV start event needs to be scheduled at tnow. If yes, a start event for that ENV is scheduled.

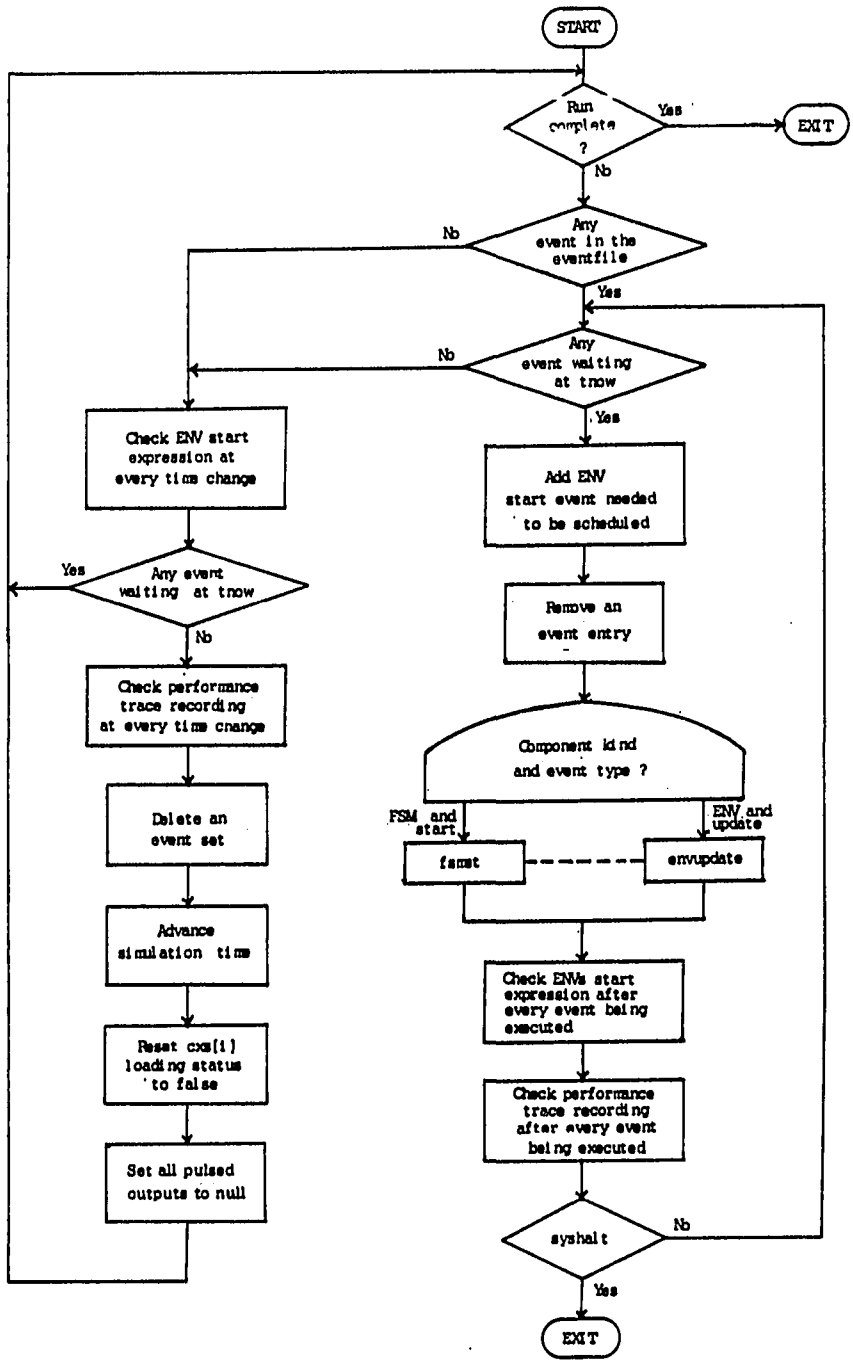


Figure 3.7. Control flow of the System Executive

v) It also checks if any performance trace needs to be collected. If yes, the performance trace collection procedure will be executed.

vi) It checks the system halt variable again. If syshalt equals true, then the simulator will exit from the System Executive, otherwise it goes back to (c).

(d) After the execution of all the events in the current event-set and before the current simulation time is increased, it checks if any ENV component start event needs to be scheduled. If yes, a start event of those ENV components will be scheduled at the current simulation time.

(e) It checks if there is any event waiting to be executed at the current simulation time, tnow. If yes, then it goes to (a); otherwise it goes to (e.i).

i) It checks if there is any performance trace to be collected at every time change. If yes, the performance trace collection procedures will be executed.

ii) It deletes the event-set, with the current simulation time, from the event file.

iii) It updates the next simulation time.

iv) It resets the loading status of static inputs of all the components to false, which implies the static inputs have not been changed at the current simulation time.

v) It resets the pulsed outputs of all the components to null.

(f) It goes to (a).

When the simulator exits from the System Executive, the simulator will ask the user if the total status of the simulated system is to be saved. If yes, the total status of the simulated system is saved in the restart initialization file, rsfile. The simulator also saves the performance trace in the performance data file, datafile, and the system status data in the system data file, sysfile. This is the end of the simulation.

2. Procedure internact

Procedure internact (9, 1033) implements the control flow of the System Executive. In the following discussion, we will delay the explanation of the details of some procedures called by procedure internact. Procedure internact starts out with a while loop (9, 1053-1250) to check if the simulation run ends or not (9, 1053). If not, procedure internact checks if there is any event in the event file (9, 1056). If yes, procedure internact proceeds to execute all the events in the current event set with the event set time equal to the current simulation time (9, 1063-1119).

If the current event set time is smaller than the current simulation time, then procedure execerr(envinstc,i,E1) is called (9, 1059). The parameters envinstc and i passed to procedure execerr are not used in this case; while the integer E1 is the error message number. If syshalt is set true at the end of the execution of procedure execerr, then procedure execstop is called to halt the system execution (9, 1049).

A second while loop (9, 1063-1119) executes all the events in the current event set. The while loop check if there is any event in the current event set (9, 1063). If true, it executes the while loop. Lines (9, 1066-1089) check all the ENV components if any of their start events need to be scheduled at tnow. If the ENV component's variables updatestate.task equals add and tstart equals tnow, which implies a start event of the ENV component needed to be scheduled at tnow, then the execution state of the ENV component is checked. If the ENV component is not in Busy state, then the variables for scheduling an event are set up and procedure schedule (9, 1080) is called to schedule a start event for the ENV component. If the ENV component is in Busy state, then procedure execerr(envinstc,i,E29) is called to report an execution error (9, 1083).

Procedure remevent (9, 1092) is called to remove an event from the event file; and the removed event is executed by calling procedure execevent (9, 1094). The details of procedure remevent and execevent will be explained in later sections. After executing an event, each ENV component is examined. If the ENV component is not the last executed component, and the ENV component's execute state is not idle, and the ENV component's check option is everyevent, then procedure checkexp(everyevent) is called to check the start expression (STARTEXP), as mentioned in Chapter II, of the ENV component (9, 1097-1106). If the start expression evaluates true, then procedure checkexp(everyevent) schedules a start event for the ENV component. Procedure savehistreg(everyevent), savehistcon(everyevent), and

saveexphist(everyevent) are called to check if there is any regular variable history, conditional variable history, and expression history, respectively, to be collected; the history traces will be recorded by those three procedures if they are needed. These complete one cycle of the while loop (9, 1063-1119) to execute an event from the current event set.

At the end of the execution of all the events in the current event set and before updating the current simulation time, lines (9, 1124-1152) check if any ENV component start event needs to be scheduled. If yes, a start event of the ENV component is scheduled at tnow.

Lines (9, 1154-1156) check if there is any event waiting to be executed at the current simulation time (some ENV start events may be scheduled just before the current simulation time is updated in lines (9, 1124-1152). If yes, procedure internact skips all the lines (9, 1159-1248) and goes back to beginning of the while loop (9, 1053); else, lines (9, 1159-1248) are executed. Procedure savehistreg(everytimechange), savehistcon(everytimechange), and saveexphist(everytimechange) (9, 1161-1165) are called to record performance traces at every time change. If the current event set does not have any event, then the current event set is deleted from the event file (9, 1170-1171). Lines (9, 1174-1175) set the next event time; if the event file is empty, then it sets the next event time greater than the ending of the simulation time to indicate no event needs to be executed until the end of the simulation time. Line (9, 1178) calls

function minitsave to obtain the nearest time that a performance trace is to be recorded and assigns it to the variable ntsave. Line (9, 1181) calls function minienvtstart to obtain the nearest time that an ENV component start event had been indicated to be scheduled, and assigns it to the variable ntstart. The current simulation time, tnow, is saved in the variable, tprev (9, 1185). Function mini(ntsave,ntevent,ntstart) is called to pick up the minimum of those three variables and assigns it to tnow, the current simulation time (9, 1186). If the current simulation time, tnow, has been advanced, then the loading status, cxsload[i], of the static inputs of all the components are reset to false; the pulsed outputs, czp[i], of all the components are set to null (9, 1191-1244). Procedure internact completes the execution of all the events in an event set and goes back to the beginning of the while loop (9, 1053) to start another event set. The while loop is repeated until the current simulation time exceeds the ending of the simulation time or the system halt variable is set to true.

3. Event file management

In the following paragraphs, we will discuss the general organization of the event file, the steps involved in putting a new event in the event file (schedule an event) and the steps involved in removing an event from the event file.

The event file keeps track of all the current and future events needing to be executed. There are two types of events, start and update. The details of start and update events will be explained in

later sections. All these events, with their attributes which include the component kind, the component name, the type of event, and the execution time of the event, are stored in the event file. The event file is organized in a multi-linked list structure as shown in Figure 3.8. The multi-linked list groups all the events with the same execution time into one event set. Each event set has a record of elements which consists of the following:

a) time - the execution time of the event set

b) count - the total number of event entries in the event set

c) nextevent - a pointer to the first event entry of the event set

d) next - a pointer to the next event set

Each event entry, within an event set, also has a record of elements which consists of the following:

a) kind - the kind of the component

b) name - the name of the component

c) action - the type of event (start or update)

d) next - a pointer to the next event entry within the event set

Event sets are organized in chronological order. The event set with the lowest execution time is stored on the top of the list. Within an event set, the event entries with ENV kind are stored on the top of the list, and the event entries with other kinds are stored below the ENV kind. The ENV events are given higher execution priority within an event set because we want the user acting through the terminal ENVs to

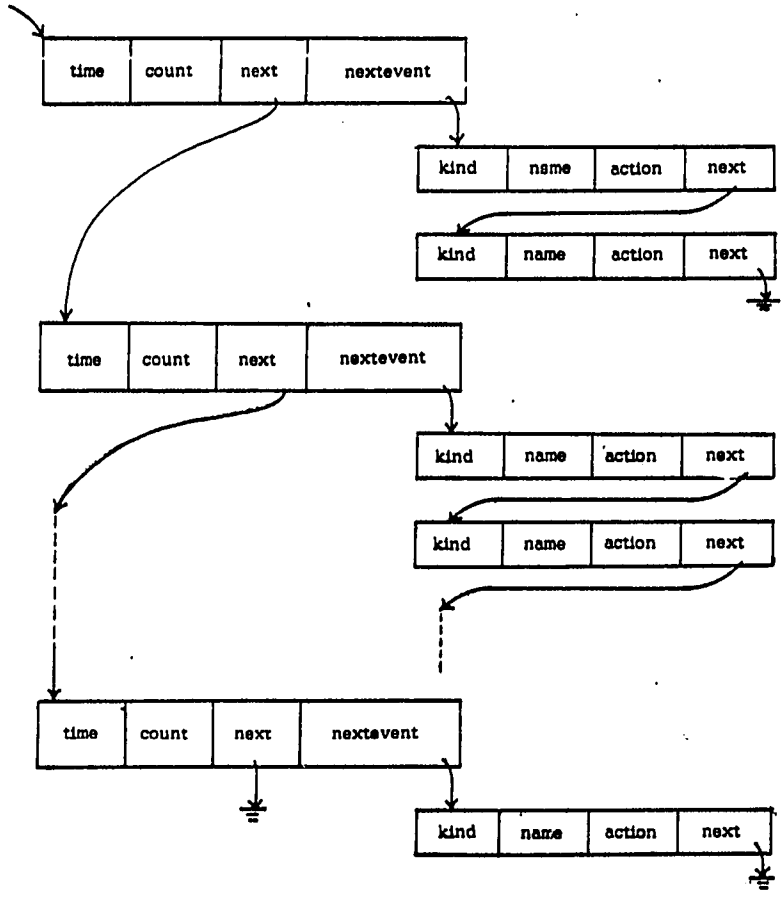


Figure 3.8. Event file structure

have first chance to examine the system status. When a new event is to be scheduled in the event file, the new event will be inserted into the event file according to the above order.

The exact order in which an event entry is removed from the event file is as follows:

a) Those event entries in the event set with the lowest execution time (earliest event time) will be removed first.

b) Within an event set, event entries with ENV kind will be removed first.

c) Within the set of event entries with ENV kind, or within the rest of the event entries in an event set, an event entry is randomly removed.

Here, we show an example on the priority of removing an event from the event file. Suppose the event file has four event entries to be executed at simulation time = 5.0 and one event entry to be executed at simulation time = 6.0 as shown in Figure 3.9a. If one event is to be removed from the event file, according to the priority established, the update event with ENV kind is removed first and the resulting event file is shown in Figure 3.9b. If another event is to be removed at this point, one of those three events with execution time = 5.0 is to be picked randomly. The resulting event file may look like one of the figures as shown in Figure 3.9c(1), Figure 3.9c(2), or Figure 3.9c(3); if the event with component name Sendmgr, Medium, or Sendbit, respectively, is removed.

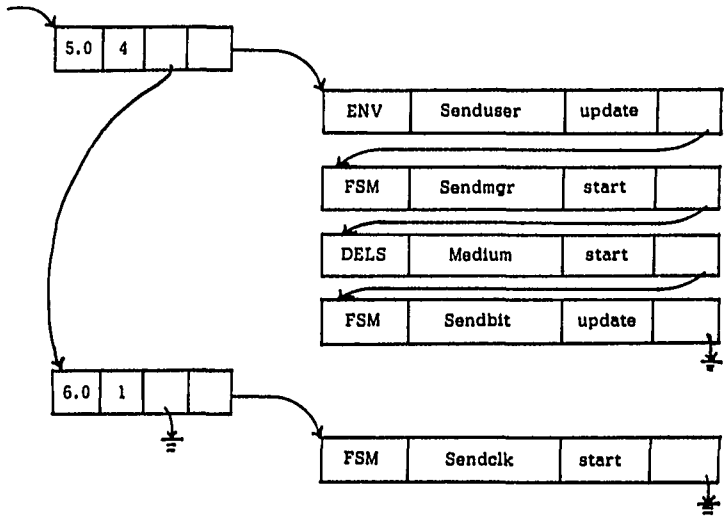


Figure 3.9a. Current event file at time = 5.0

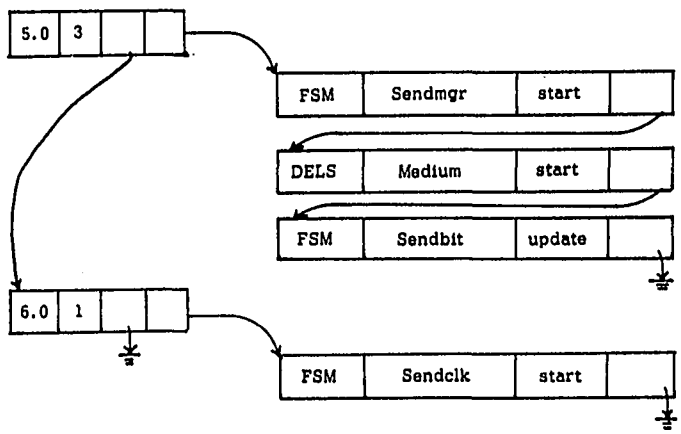


Figure 3.9b. One event is removed from event file in Figure 3.9a

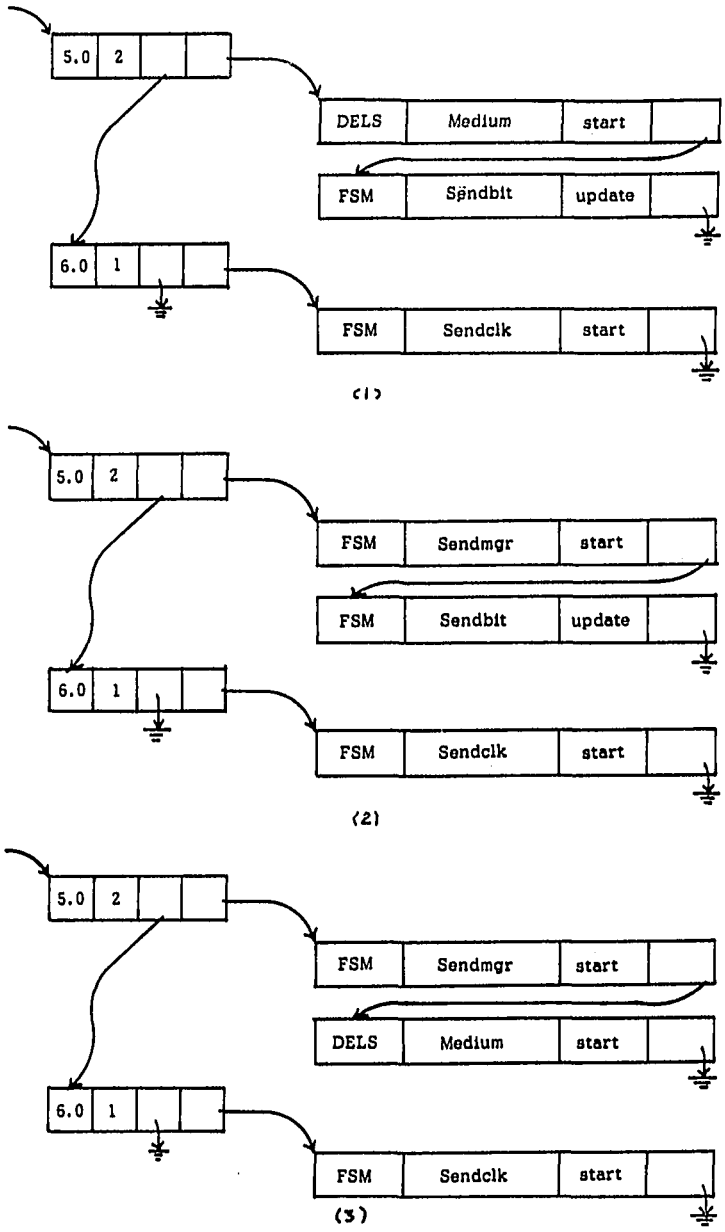


Figure 3.9c. (1), (2) and (3) show the three possible results after one event is removed from the event file in Figure 3.9b

4. Procedure schedule and remevent

Procedure schedule (11, 750) implements the process of putting a new event (scheduling an event) into the event file. Just before the procedure schedule is called, the global variables identifying the attributes of an event (component kind, component name, event time, event type) have to be set, e.g. lines (, 1076-1080), so that procedure schedule knows the type of event and its attributes needing to be put into the event file. Procedure schedule starts out to assign the header of the event file to the local variable, curl (11, 758). If the event file is not empty (11, 759), then the event set list is searched until the event set time is greater than or equal to the event time of the new event (indicated by the variable file_time or the end of the event set list is reached (11, 767-771). If one of the event set times matches the file_time, then the event set count is increased by one, which means a new event is to be added into the event set (11, 773), else a new event set has to be created (11, 777-780) for the new event and inserted into the event file (11, 781-794). In line (11, 759), if the event file is empty, then a new event set is created and the event file pointer, eventfilept, is initialized to point to the event set (11, 800-806).

At this point, the variable, curl, is pointing at the event set in which the new event is to be inserted. The variable, overlap, is set to indicate if there is overlap of the same event in the same event set. The while loop (11, 615-821) checks if there is any identical event already in the same event set. If there is an identical event, then the

event count of the event set is decreased by one (11, 823), else a new event entry is created (11, 827-832) and inserted into the event entry list of the event set (11, 838-869). If the event entry list is empty, then the new event entry is inserted at the top of the list (11, 837-843); else, if the first event entry of the event entry list is not an ENV component, then the new event entry is inserted to the top of the list (11, 845-849); else, the new event entry is inserted after all the ENV event entries in the event entry list (11, 858-869). At the end, the scheduled event and its attributes are printed on the user terminal (11, 870-873). This completes the scheduling process.

Procedure remevent (9, 809) implements the process of removing an event from the event file. After executing this procedure, the current removed event and its attributes: component kind, component name, and event type, are stored in the global variables: csmkind, csmname, and csmevent, respectively. Procedure remevent starts out to check if the event file is empty or not. If the event file is empty, then procedure execerr(name,num,E19) (9, 817) is called; else, the proper event is to be removed from the event file (9, 822-860). Function envno is called to find the number of ENV event entries in the current event set and assigned to the local variable i (9, 822). If there is at least one ENV event entry, then the function randint(i) is called to randomly pick a value between 1 and i to indicate which ENV event entry is to be removed (9, 825); else, function randint(eventfilept^.count) is called to randomly pick a value between 1 and eventfilept^.count to indicate which event entry is to be removed (9, 826). If the returned

integer variable, n, is 1, the first event entry is to be removed; 2 implies the second event entry is to be removed, and so on. If n equals 0, which it should not, then procedure execerr(name,num,E19) is called to report an error (9, 827); otherwise, procedure remevent goes down the event entry list sequentially to pick out the nth event (9, 834-841). The global variables corresponding to the current removed event are loaded. A description of the removed event is printed on the terminal (9, 848-852). The event entry is removed from the event set (9, 855-856), event set pointers are adjusted, and the event count of the current event set is decreased by one (9, 859). This completes the process of removing an event from the event file.

5. Execution of SAN components

SAS is an event driven simulator. There are two types of events, start and update. Also there are three execution states related to events: Idle, Pending (Pend), and Busy. In the SAS program, we will use lower case for the three execution states idle, pend, and busy. In the following sub-sections, we will explain the meaning of the execution states: how each component changes its execution state with respect to its input excitations, execution of start and update events, start and update event scheduling, and the detailed steps involved in executing start and update events.

a. Finite State Machine (FSM): exists in three different execution states as shown in Figure 3.10. When a FSM component is in the Idle execution state, it is understood that the component has no defined

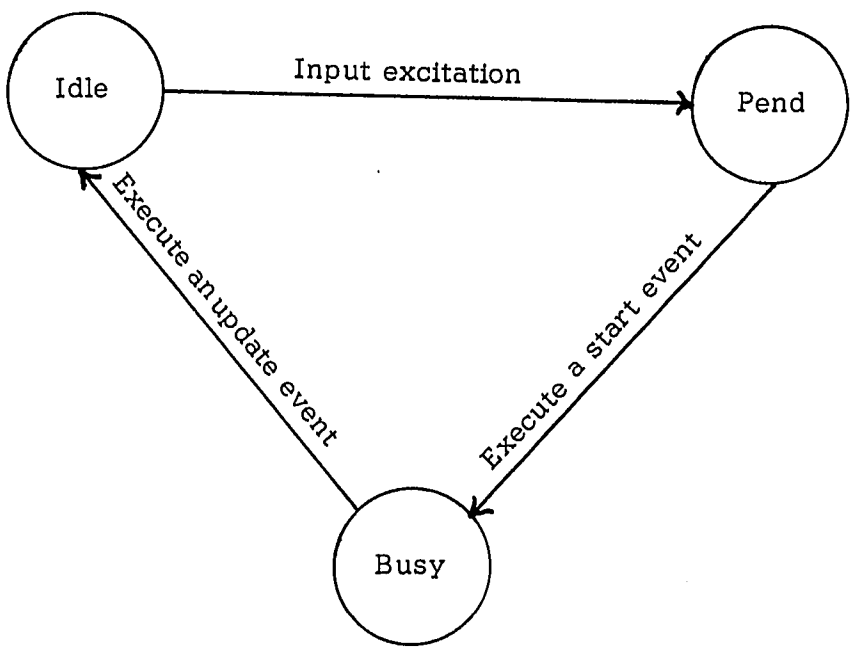


Figure 3.10. Execution state transition diagram for FSM, CFP and CFS

pulsed inputs or outputs; the current state and the static outputs of the component are defined. At the arrival of a pulsed input, the execution state of the component is set to Pending and a start event for the component is scheduled in the event file. The reason for the Pending state is to allow SAS, a sequential procedure, to emulate the simultaneous execution of multiple components at the same simulation time. The Pending state identifies that a pulsed input has arrived at the FSM component and the component is waiting to be executed at the current simulation time. When executing the start event of a Pending FSM, the execution state is set to Busy. If the component execution time is greater than zero, then an update event for the component is scheduled, else the update event of the component is executed immediately. When executing the update event of a busy component, the execution state is set to Idle again. This completes the execution cycle of a FSM component.

The detailed steps in executing a start event of a FSM component are as follows:

i) The execution state of the component is set to Busy.

ii) The Next State Function (FNS) of the component is executed.

iii) The Pulsed Output Function (FOUTP) of the component is executed.

iv) The Static Output Function (FOUTS) of the component is executed using the new next state.

v) The update record, which stores the next state, next pulsed and static outputs, and the update time, of the component is loaded.

vi) If the component execution time is greater than zero, then an update event of the component is scheduled, else the update event of the component is executed immediately.

The detailed steps in executing an update event of a FSM component are as follows:

i) The execution state of the component is set to Idle

ii) The pulsed inputs of the component are reset to null

iii) Update the current state of the component from the update record

iv) Update the pulsed and static outputs of the component from the update record

v) The updated pulsed and static outputs are loaded into their destination components

b. Pulsed Combinational Function (CFP): also exists in three different execution states as shown in Figure 3.10. The changes of the execution state with respect to its input excitation, and the execution of its start and update events is similar to that of a FSM.

The detailed steps in executing a start event of a CFP component are as follows:

i) Set the execution state to Busy

ii) Execute the Pulsed Output Function

iii) Store the update time in the update record.

iv) If the component execution time is greater than zero, then an update event of the component is scheduled, else the update event of the component is executed immediately.

The detailed steps in executing an update event of a CFP component as follows:

i) Set the execution state to Idle.

ii) Reset the pulsed inputs to null.

iii) Update the pulsed outputs and load the updated pulsed outputs into their destination components.

c. Static Combinational Function (CFS): also exists in three different execution states as shown in Figure 3.10. The changes of the execution state with respect to its input excitation and the execution of its start and update event are similar to that of a FSM, except that the input excitation of a CFS component is due to a change of static inputs while that of a FSM component is due to the arrival of pulses at the pulsed inputs.

The detailed steps in executing a start and update event of a CFS component are similar to that of a CFP component, except that the former executes the Pulsed Output Function of the component and the latter executes the Static Output Function of the Component.

d. Pulsed Delay (DELP): exists in only two different execution states as shown in Figure 3.11. At the arrival of a pulsed input, the execution state of the component is set to Pend and a start event of the

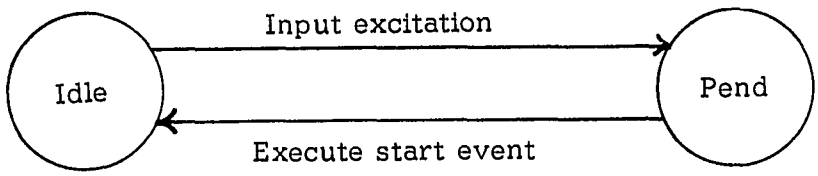


Figure 3.11. Execution state transition diagram for DELP, DELS, CLK and DER

component is scheduled into the event file. After executing the start event of the DELP component, the execution state of the component is set to Idle and an update event of the component is scheduled into the event file.

The detailed steps in executing a start event of a DELP component are as follows:

i) Set the update time, which is equal to the current simulation time + the delay time

ii) Store the pulse in the update record

iii) Schedule an update event of the component at the update time

v) Set the execution state to Idle

The detailed steps in executing an update event of a DELP component are as follows:

i) Remove the current pulsed output from the update record

ii) Load the pulsed output into its destination components

e. Static Delay (DELS): exists in two different execution states as shown in Figure 3.11. The changes of the execution state with respect to its input excitation and the execution of its start and update events are similar to those of a DELP component, except that the input excitation of a DELP component is due to the arrival of a pulse at the pulsed input while that of a DELS component is due to a change of its static input. The detailed steps in executing a start and update event of a DELS component are also similar to those of a DELP component.

f. Queue (QUE): has more complicated execution state transitions than the other component kinds as shown in Figure 3.12. The

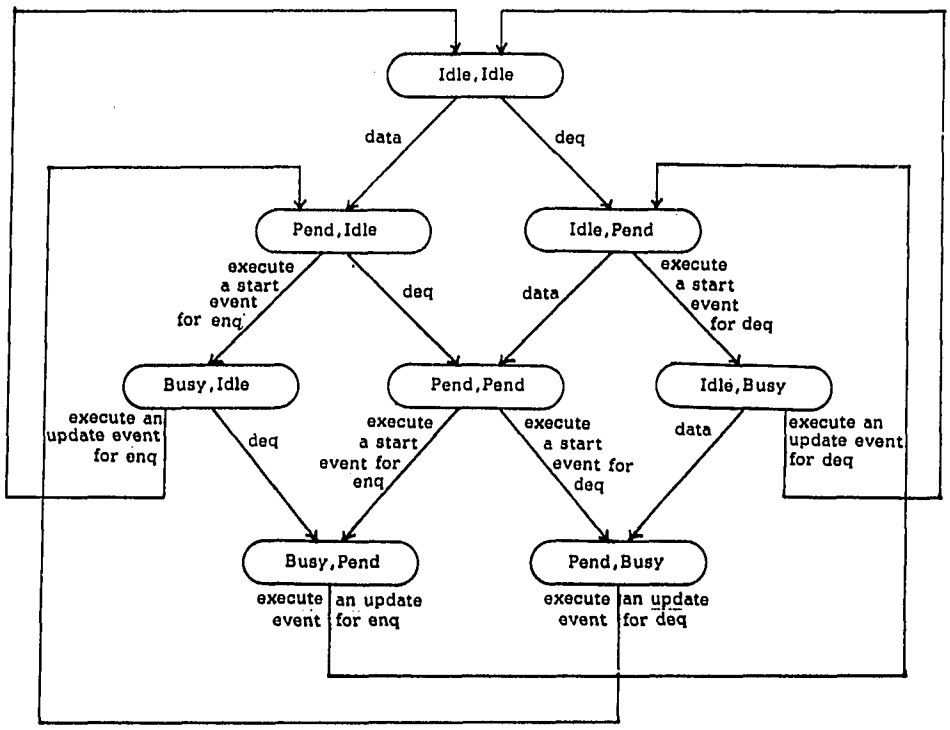


Figure 3.12. Execution state transition diagram for QUE

execution state of a QUE depends both on the enqueue status (the execution state of loading new data into the queue) and dequeue status (the execution state of pulsing out data from the queue). With each state in Figure 3.12, we associate a status doubleton; the value on the left represents the enqueue status and that on the right represents the dequeue status; e.g. when they are both in Idle state, the execution state of queue is represented by the state (Idle,Idle). The transitions of the execution state of a QUE are as follows:

i) At state (Idle,Idle), if a pulsed datum arrives, then it will get into state (Pend, Idle) and schedule a start event for the enqueueing process; if a deq control input arrives, then it will get into state (Idle, Pend) and schedule a start for the dequeueing process. At the arrival of either a pulsed datum (enqueue) or a deq control input (dequeue), a start event is scheduled only if both the enqueue state and dequeue state are Idle. The queue may be busy reacting to only one pulsed input at a time. If either a pulsed datum or a deq control input arrives while the queue is busy reacting to the other pulse, then the queue will not become busy with the later pulse until the reaction to the first pulse is done. If both a pulsed datum and a deq control input arrive at the same time, then one of the pulses is randomly picked to be executed first, while the other will have to wait until the end of the reaction to the first pulse.

ii) At state (Pend, Idle), if a start event for the enqueueing process is executed, then it will get into state (Busy, Idle); if the enqueue execution time is zero, then the update event of the enqueueing process is executed immediately, else an update event of the enqueueing process is scheduled. If a deq control input arrives, then it will get into state (Pend, Pend).

iii) At state (Idle, Pend), if a start event for the dequeueing process is executed, then it will get into state (Idle, Busy); if the dequeue execution time is zero, then the update event of the dequeueing process is executed immediately, else an update event of the enqueueing process is scheduled. If a pulsed datum arrives, then it will get into state (Pend, Pend).

iv) At state (Busy, Idle), if an update event for the enqueueing process is executed, then it will get into state (Idle, Idle); if a deq control input arrives, then it will get into state (Busy, Pend).

v) At state (Idle, Busy), if an update event for the dequeueing process is executed, then it will get into state (Idle, Idle); if a pulsed datum arrives, then it will get into state (Pend, Busy).

vi) At state (Pend, Pend), if a start event for the enqueueing process is executed, then it will get into state (Busy, Pend); if the enqueue execution time is zero, then the update event of the enqueueing process is executed immediately, else an update event of the enqueueing process is scheduled.

If a start event for the dequeueing process is executed, then it will get into state (Pend, Busy); if the dequeue execution time is zero, then the update event of the dequeueing process is executed immediately, else an update event of the dequeueing process is scheduled.

vii) At state (Busy, Pend), if an update event for the enqueueing process is executed, then it will get into state (Idle, Pend) and schedule a start event for the dequeueing process.

viii) At state (Pend, Busy), if an update event for the dequeueing process is executed, then it will get into state (Pend, Idle) and schedule a start event for the enqueueing process.

The detailed steps in executing a start event of a QUE component are as follows:

i) If the execution state is (Idle, Pend), then the start event of the dequeue process is executed; if the execution state is (Pend, Idle), then the start event of the enqueue process is executed; else, if the execution state is (Pend, Pend), then it randomly picks either the start event of the dequeue process or of the enqueue process to be executed.

ii) If the start event of the enqueueing process is executed, then it sets the enqueue status to Busy and sets the enqueue process update time; if the queue is open, then it sets the next update state to closedempty and the next pulsed output to the current pulsed data; if the queue is closed, then it sets the next update state to closednotempty, the next

pulsed output to null, and the next update task to add (which indicates a pulsed datum will be added into the queue at the update time).

iii) If the start event of the dequeueing process is executed, then it sets the dequeue status to Busy and sets the dequeue process update time; if the queue is open or closedempty, then it sets the next update state to open and the next pulsed output to null; if the queue is closednotempty, then it sets the next update task to delete (which indicates that the oldest stored datum is to be dequeued and pulsed out); if the number of data entries in the queue is 1, then the next update state is set to closedempty, else the next update state is set to closednotempty.

iv) If the next update time equals the current simulation time, then the update event of the QUE component is executed, else an update event of the QUE component is scheduled.

The detailed steps in executing an update event of a QUE component are as follows:

i) Set the current state and static output of the queue.

ii) Load the static output into its destination component.

iii) If it is an enqueue process update, then the current pulsed output is set according to the update record; if the next update task is add, then the pulsed data input is inserted into the queue; the pulsed input is set to null; the enqueue status is set to Idle; if the dequeue status is in Pend state, then a

start event of the queue is scheduled into the event file.

iv) If it is a dequeue process update, then a pulsed data is deleted from the queue and assigned to the current pulsed output; the current dequeue control input is set to null; the dequeue status is set to Idle; if the enqueue status is in Pend state, then a start event of the queue is scheduled into the event file.

v) Loads the pulsed output into its destination components.

g. Derivative (DER): exists in two different execution states as shown in Figure 3.11. At the change of the DER component's static input, the execution state of the component is set to Pend and a start event of the component is scheduled into the event file. After executing the start event of the DER component, the execution state of the component is set to Idle. The DER component reacts to the change of its static input instantaneously, which means when a rising or falling edge in the static input is detected, a r or f pulse is pulsed out at the same simulation time. Since the DER component always has zero execution time, the update event for the DER component will not be needed.

The detailed steps in executing a start event of a DER component are as follows:

i) If a rising edge is detected, then the pulsed output is set to r, else, if a falling edge is detected, then the pulsed output is set to f.

ii) The pulsed output is loaded into its destination components.

iii) The current state is set equal to the current static input.

iv) The execution state is set to Idle.

h. Clock (CLK): exists in two different execution states as shown in Figure 3.11. At the arrival of a start pulse at the CLK component pulsed input, the execution state of the component is set to Pend and a start event for the component is scheduled into the event file. After executing the start event of the CLK component, the execution state of the component is set to Idle and an update event of the component is scheduled into the event file.

The detailed steps in executing a start event of a CLK component are as follows:

i) If the pulsed input is a start pulse and if the clock is in the running state, then the scheduled update event of the clock is deleted from the event file; the clock is set to running state; the time for the timeout is set; and the update state record is set with nts and ntzs[1] equal 'expired', ntzp[i] equals 'timeout', and the update time equals the timeout time.

ii) If the pulsed input is a reset pulse and if the clock is in the running state, then the scheduled update event of the clock is deleted from the event file; the clock is set to the reset state; and the update state record is set with nts and ntzs[1] equal 'reset', ntzp[1] equals null, and the update time equals tnow.

iii) If the update time equals tnow, then the update event of the clock is executed, else an update event of the clock is scheduled into the event file.

The detailed steps in executing an update event of a CLK component are as follows:

i) If the ntzp[1] equals 'timeout', then the czp[1] is set to 'timeout' and the current pulsed output is loaded into its destination components.

ii) If the ntzs[1] does not equal czs[1], then czs[1] is set to equal to ntzs[1] and the current static output is loaded into its destination components.

i. Environment (ENV): exists in three different execution states as shown in Figure 3.13. At Idle state, if a pulsed input arrives, or the ENV start expression evaluates to true, or a start event is scheduled at the current simulation time (in executing an ENV start event, the user may file another start event of the component to be scheduled at a future time), then the execution state of the component is set to Pend and a start event of the ENV component is scheduled into the event file. At Pend state, after executing a start event of the ENV component, the execution state of the component is set to Busy; if the component execution time is greater than zero, then an update event of the component is scheduled, else the update event of the component is executed immediately. After executing the update event of the component, the execution state of the component is set to Idle state again.

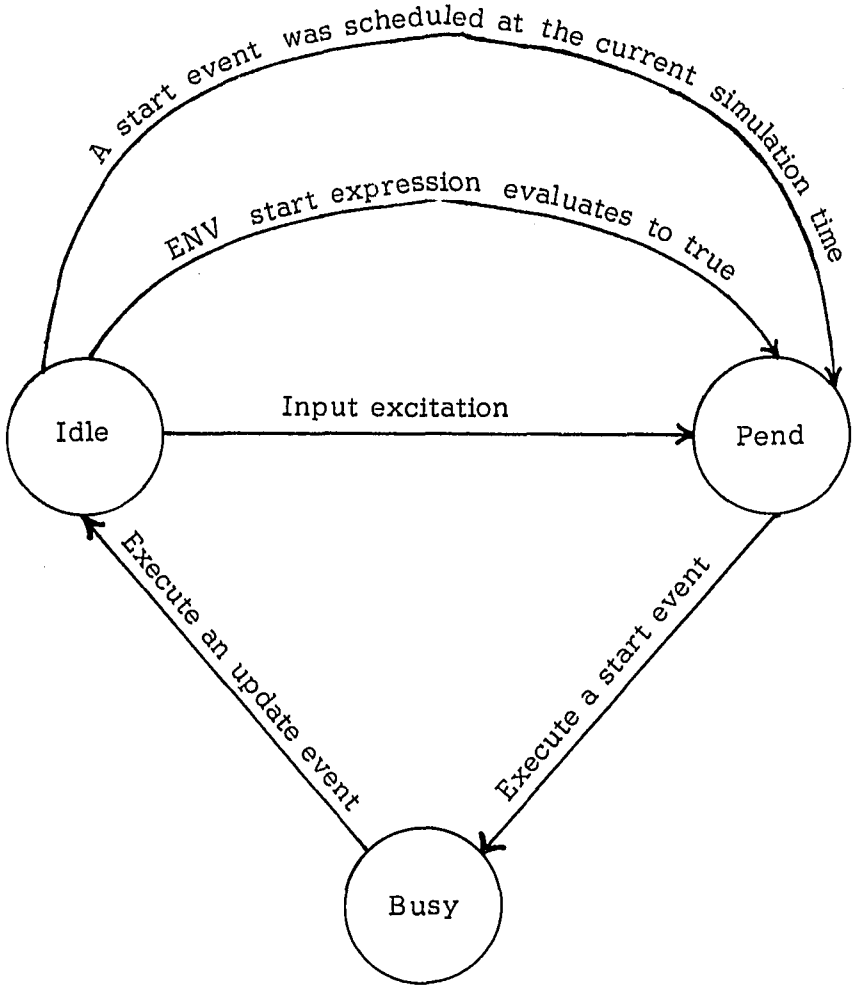


Figure 3.13. Execution state transition program for the ENV component

The detailed steps in executing a start event of an ENV component are as follows:

i) The execution state of the component is set to Busy.

ii) If the ENV function is terminal, then the terminal function procedure is called, else, if the ENV function is procedure, then the ENV procedure, envfunction, is called.

iii) Set the update time.

iv) If the update time equals tnow, then the update event of the component is executed immediately, else an update of the component is scheduled into the event file.

The detailed steps in executing an update event of an ENV component are as follows:

i) Set the execution state to Idle.

ii) Clear all the pulsed inputs to null.

iii) Update the pulsed and static outputs of the component from the update record.

iv) The updated pulsed and static outputs are loaded into their destination components.

6. Implementation of the start and the update event of each component kind

This section discusses procedures fsmst, fsmupdate, cfpst, cfpupdate, cfsst, cfsupdate, delpst, delpupdate, delsst, delsupdate, quest, queupdate, clkst, clkupdate, envst, envupdate, and derst, which implement the start and the update event of the FSM, CFP, CFS, DELP, DELS,

QUE, CLK, ENV components and the start event of the DER component, respectively.

a. Procedure fsmst(name:smname) (10, 1153): implements the start event of the FSM components. The variable name indicates the particular component for which the start event is to be executed. Procedure fsmst first checks if the execution state is pend or not (10, 1160). If it is, then the start event is executed (10, 1162-1197), else procedure execerr(name,num,E2) (10, 1202) is called to report an execution error by passing the name of the current executing component, name, and the error message, E2, and a dummy integer, num.

In executing the start event, procedure fsmst first sets the execution state to busy (10, 1162). It then starts to execute the FNS, FOUTP, and FOUTS functions of the component (10, 1164-1172). If the FNS type is list, then procedure fsmfns(name) is called to match the current state and input array with the FNS list to produce the next state value; else, if the FNS type is a procedure, then procedure fsmfunction(fsm[name].fnsproc) is called to execute the FNS procedure, which is indexed by the procedure number, fsm[name]fnsproc; else, procedure execerr(name,num,E28) is called (10, 1165-1169). If the system halt variable, syshalt, is set true then procedure execstop is called to stop the simulation execution. FOUTP and FOUTS functions (10, 1172-1182) are implemented similarly to the FNS function. At the end, the next update time is set (10, 1185). If the next update time equals tnow, then the FSM update event is executed, else an update event of the FSM component is scheduled into the event file.

Inside procedure fsmst, procedures fsmfns(name), fsmfoutp(name), and fsmfouts(name) are called to execute the list type specification of the FNS, FOUTP, and FOUTS functions. As for the procedure type specification of the FNS, FOUTP, and FOUTS functions, procedure fsmfunction(num) is called by passing the appropriate procedure number, num, to execute their functions. Procedure fsmfunction(num:integer) (20, 755) contains all the FNS, FOUTP, and FOUTS procedures defined in the FSM components. The variable, num, passed to procedure fsmfunction(num) is used as a case index to branch to the section of code to execute the appropriate function.

Procedure fsmfns(name:smname) (10, 933) implements the execution of the list type specification of the FNS function. Procedure fsmfns starts out to set temp pointing to the first entry of the FNS list (10, 941). Procedure fsmfns then searches the FNS list until the current state and input values match with one of the entries in the FNS list or the end of the FNS list is reached. The while loop (10, 948-980) performs the searching. Each loop matches one entry of the FNS list to the current state and input values. Line (10, 950) checks if the current state value equals the state value in the FNS list entry or if the state value in the FNS list entry is don't care. If so, the procedure fsmfns continues on to match the pulsed and static inputs; otherwise procedure fsmfns advances to the next FNS list entry (10, 979). Lines (10, 953-962) check if the current pulsed inputs match the pulsed input value in the FNS list entry. If they match, then the local variable xpok is set to true. If xpok is true, then procedure fsmfns

continues on to check if the current static input values match the static input values in the FNS list entry. If they match, then the local variable xsok is set to true. If the current state pulsed and static input values equal those of an FNS list entry, then the local variable match is set to true. If match is not true, then procedure fsmfns advances to the next FNS list entry. The while loop (10, 948-980) is repeated until match is true or the end of the FNS list is reached.

If the current state and input value match one of the FNS list entries, then the next update state, nts is loaded (10, 985-1000). The character string denoting the next update state value may be a string constant or variable. If the character string is a string constant, then nts is set to equal the character string, temp .nexts.sval (10, 988). If the character string is a string variable, which may be a local cs, cxp[i], cxs[i], czp[i] or czs[i] variable, then the appropriate value is assigned to the next update state, nts (10, 991-998).

Procedure fsmfoutp(name:smname) (10, 1012) and procedure fsmfouts (name:smname) (10, 1098) implement the execution of the list type specifications of the FOUTP and FOUTS functions. The implementation details of procedures fsmfoutp and fsmfouts are similar to that of procedure fsmfns. Readers can refer to the discussion of procedure fsmfns to understand the procedures fsmfoutp and fsmfouts.

b. Procedure fsmupdate(name:smname) (10, 1216): implements the update event of the FSM components. The variable name passed to the procedure fsmupdate indicates the particular component for which the update event is to be executed. Procedure fsmupdate starts out to check

if the execution state is in Busy state or not (10, 1226). If not, then procedure execerr(name,num,E4) is called to report an execution error. Procedure fsmupdate continues by checking if the update time equals tnow (10, 1229). If not, then procedure execerr(name,num,E3) (10, 1274) is called. Procedure fsmupdate then starts to update the FSM component. The execution state is first set to Idle. All the pulsed inputs are set to null. Function ckvalidset(S,updatestate.nts) is called to check if the next update state value is an element of the state set. If the next update state value is an element of the state set, then the next update value is assigned to the current state variable, cs, otherwise procedure execerr(name,num,E25) is called. Lines (10, 1244-1255) update all the pulsed outputs. For each of the next update pulsed outputs, if the output is not null, then it is checked if it is an element of the pulsed output set by calling procedure ckvalidset(ZP[i],updatestate,ntzp[i]). If it is not, then procedure execerr(name,num,E26) is called, else the current pulsed output is updated and procedure schdestxp(fsmsym,name,i) is called to load the pulse into its destination components. If the next update pulsed output is null, then the current pulsed output is set to null.

Lines (10, 1260-1268) update all the static outputs. If each of the next update static outputs does not equal its current static output, then the next update static output is checked if it is an element of the static output set by calling procedure ckvalidset(ZS[i],updatestate.ntzs[i]). If it is not, then procedure execerr(name,num,E27) is called, else the current static output is

updated and procedure schdestxs(fsmsym,name,i) is called to load the static output into its destination components. If the next update static output equals the current static output, then nothing is done. At the end, all the next update pulsed outputs are set to null (10, 1271).

Inside procedure fsmupdate, procedures schdestxp(fsmsym,name,i) and schdestxs(fsmsym,name,i) are called to load the pulsed and static outputs into their destination components. Procedure schdestxp(kind:smtype;name:smname;num:integer) (11, 1354) sorts out the kind of component needing to be processed. It then calls the appropriate procedure to load the component's pulsed output into its destination components (11, 1359-1368). In this case a FSM component's pulsed output needs to be processed. The procedure fsmdestxp(name,num) (11, 1360) is called to load each component's pulsed output into its destination components.

Procedure fsmdestxp(name:smname;num:integer) (11, 1031) finds out the destination of the pulsed output, fsm[name].czp[num]. It then calls procedure loadxp to load the pulsed output into its destination. Procedure fsmdestxp starts out by initializing the fanout number, i, to 1 (11, 1040). The while loop (11, 1041-1068) first picks out the destination component kind (dkind), component name (dname), the pulsed input number (dindex), and the value to be loaded into the destination component pulsed input (dvalue). If the pulsed output is not connected anywhere, then procedure fsmdestxp does nothing, else procedure loadxp(dkind,dname,dindex,dvalue) (11, 1051) is called to load the pulsed output into its destination components. Procedure fsmdestxp sets

up the variables for a start event for each of the destination components to be scheduled. If the component kind is not a QUE, then procedure schedule is called to schedule a start event into the event file; else, for a QUE, if both the enqueue state and the dequeue state of the queue are idle, then a start event is scheduled, else it does nothing. (A start event of a QUE is scheduled only if both the enqueue status and dequeue status are idle as discussed in Chapter III.F.5.f.) At the end, the fanout number is increased by one. The loop (11, 1041-1068) is repeated until all the destinations are treated or the fanout number exceeds the maximum fanout number.

Procedure loadxp(kind:smtype;name:smname;num:integer; xpvalue:string) (11, 884) loads a value into the destination pulsed input. Procedure loadxp starts out by assigning temp pointing to the string set of the pulsed input set according to the kind of component to be loaded (11, 893-901). The while loop (11, 909-914) checks if the value to be loaded is an element of the pulsed input set. If so, the local variable valid is set to true, else valid is set to false. If valid, then procedure execerr(name,num,E4) is called to report an execution error with name indicating the component name, num indicating the pulsed input, and E4 indicating the error message. If valid is true, then, depending on the kind of component procedure, loadxp branches off to the appropriate lines. As an example, if the component kind is a FSM then lines (11, 921-937) are executed. If the execution state of the component is idle, then the pulsed input is loaded and the execution state is set to pend; else if the execution state is pend, multiple

pulses option is enabled, and the current pulsed input is null, then the pulsed input is loaded, if the pulsed input is not null, then procedure execerr(name,num,E18) is called (11, 931-933). If the execution state is pend and multiple pulses option is not enabled, then procedure execerr(name,num,E15) is called. The details of loading the pulsed inputs of different components are similar. This completes the loading process of a pulsed output into a pulsed input.

To load a pulsed output from a FSM component into its destination pulsed input, first procedure schdestxp(kind:smtype;name:smname;num: integer) (11, 1354) is called. Procedure schdestxp will call procedure fsmdestxp(name:smname;num:integer) (11, 1031) to find out the destination component. Procedure fsmdestxp will call procedure loadxp(kind:smtype;name:smname;num:integer;xpvalue:string) (11, 884) to load each pulsed output value into the pulsed input. In the same manner, to load each static output from a FSM component into its destination static inputs, first procedure schdestxs(kind:smtype;name: smname;num:integer) (11, 1780) is called, which in turn calls procedure fsmdestxs(name:smname;num:integer) (11, 1504) to find a destination static input. Procedure fsmdestxs then calls procedure loadxs(kind: smtype;name:smname;num:integer;xpvalue:string) (11, 1381) to load the static output value into the static input. The implementation details of procedures schdestxs, fsmdestxs, and loadxs are similar to those of procedures schdestxp, fsmdestxp, and loadxp. Readers can refer to the discussion of procedure schdestxp, fsmdestxp, and loadxp to understand the implementation of procedure schdestxs, fsmdestxs, and loadxs.

c. Procedures cfpst(name:smname) (10, 1368) and cfpupdate(name: smname) (10, 1416): implement the start and update events of the CFP component. The implementation details of procedures cfpst and cfpupdate are similar to those of procedures fsmst and fsmupdate, except that the former only has to execute the pulsed output function and load the pulsed outputs into their destinations, while the latter has to execute the next state function, pulsed output function, and static output function and load both the pulsed and static outputs into their destinations. Readers can refer to the explanation of procedures fsmst and fsmupdate to understand the implementation of procedures cfpst and cfpupdate.

d. Procedures cfsst(name:smname) (10, 1537) and cfsupdate(name: smname) (10, 1585): implement the start and update events of the CFS component. The implementation details of procedures cfsst and cfsupdate are also similar to those of procedures fsmst and fsmupdate, except that procedures cfsst and cfsupdate only have to execute the static output function and load the static outputs into their destinations. Readers can refer to the explanation of procedure fsmst and fsmupdate to understand the implementation or procedures cfsst and cfsupdate.

e. Procedure delpst(name:smname) (10, 1627): implements the start event of the DELP component. Procedure delpst starts out by checking if the execution state is pend. If it is false, then procedure execerr(name,num,E2) is called; else the next update time is set (10, 1640), a new record is created to save the next update pulse (10, 1643-1647), the new record is inserted into the DELP component update

record by calling procedure insdelp(name,temp) (10, 1650), an update event of the DELP component is scheduled into the event file (10, 1653-1656), and the execution state is set to idle (10, 1659). This completes the start event of a DELP component.

f. Procedure delpupdate(name:smname) (10, 1668): implements the update event of the DELP component. Procedure delpupdate starts out by checking if the list for the update pulse record is empty or not. If it is empty, then procedure execerr(name,num,E5) (10, 1696) is called, else, the first update time is checked. If the update time is not equal to tnow, then procedure execerr(name,num,E3) (10, 1693) is called, else the update pulse is checked if it is an element of the pulsed output set. If it is not, then the procedure execerr(name,num,E26) (10, 1684) is called; else the current pulsed output is loaded, procedure schdestxp(delpsym,name,i) is called to load the pulsed output into its destinations, and the update pulse record is removed from the list of the update pulse record. This completes the update event of a DELP component.

g. Procedures delsst(name:smname) (10, 1702) and delsupdate (name:smname) (10, 1742): implement start and update events of the DELS component. The implementation details of procedures delsst and delsupdate are similar to those of procedures delpst and delpupdate except that the former delay the pulsed input values to appear at the pulsed output, while the latter delay the static input transitions to appear at the static output. Readers can refer to the explanation of

procedures delpst and delpupdate to understand the implementation of procedures delsst and delsupdate.

h. Procedure quest(name:smname) (10, 1937): implements the start event of the QUE component. If the enqueue status is idle and the dequeue status is pend, then procedure deqst (10, 2016) is called to execute the start event of the dequeue process; else, if the enqueue status is pend and the dequeue status is idle, then procedure enqst (10, 2018) is called to execute the start event of the enqueue process; else, if both the enqueue and dequeue status are pend, then procedure quest will randomly pick either procedure enqst or procedure deqst (10, 2022-2024); else procedure execerr(name,num,E6) is called (10, 2027). At the end, the update time is set; if the update time equals tnow, then procedure queupdate(name) is executed, else an update event of the QUE component is scheduled into the event file.

Procedure enqst (10, 1940) implements the start event of the enqueueing process. The enqueue status is first set to busy. The enqueue process update time is set (10, 1948). If the queue is open, then the next update state and static output is set to closedempty and the next pulsed output is set to the current pulsed data; else the next update state and static output are set to closednotempty, the next pulsed output is set to null, and the update task is set to add (which indicates a pulsed data will be added into the queue at the update time) (10, 1960-1963).

Procedure deqst (10, 1971) implements the start event of the dequeue process. The dequeue status is first set to busy. The dequeue

process update time is set (10, 1979). If the queue is open or closedempty, then the next update state and static output are set to open and the next pulsed output is set to null; else, if the queue has only one data element, then the next update state and static output are set to closedempty, else they are set to closednotempty; also, the next update task is set to delete (which indicates that the oldest stored data is to be dequeued at the next update time) (10, 1992-2003).

i. Procedure queupdate(name:smname) (10, 2051): implements the update event of the QUE component. Procedure queupdate starts out by checking if the update time equals tnow. If not, then procedure execerr(name,num,i) (10, 2141) is called, else procedure queupdate proceeds to update the queue. The current state and static output of the queue is updated (10, 2065-2071). If the enqueue status is busy and the dequeue status is not busy, then the enqueue process is updated (10, 2077-2094); else, if the dequeue status is busy and the enqueue status is not busy, then the dequeue process is updated (10, 2100-2120); else procedure execerr(name,num,E7) is called. At the end, if a pulse is generated at the pulsed output, then procedure ckvalidset(ZP[1],czp[1]) is called to check if the pulse is an element of the pulsed output set. If it is not, then procedure execerr(name,num,E26) is called, else procedure schdestxp(quesym,name,1) is called to load the pulsed output value into its destination components (10, 2131-2134). The next update pulsed output is cleared to null (10, 2137). This completes the update process.

j. Procedure derst(name:smname) (10, 1785): implements the start event of the DER component. If the execution state is not pend, then procedure execerr(name,num,E2) is called, else, if the static input changes from low (l) to high (h) or from 0 to 1, then the current pulsed output is set to rising (r) and procedure schdestxp(dersym,name,1) is called to load the pulsed output into its destination; else, if the static input changes from high (h) to low (l) or from 1 to 0, then the current pulsed output is set to falling (f) and procedure schdestxp(dersym,name,1) is called to load the pulsed output into its destination (10, 1797-1809). The current state is set to the current static input and the execution state is set to idle (10, 182-1815). This completes the execution of a DER component.

k. Procedure clkst(name:smname) (10, 1829): implements the start event of the CLK component. Procedure clkst starts by checking the execution state of the CLK component. If the execution state is not pend, then procedure execerr(name,num,E2) is called (10, 1891); else procedure clkst checks the type of pulsed input. If it is a start pulse, then, if the clock is running, procedure delevent(clksym,name,update) is called to delete the update event of the clock, which was scheduled by the arrival of a previous start input; the current state and static output of the clock are set to running; the update time for the timeout pulse is set; and the next update state and static output are set to expired (10, 1843-1857); else, if the pulsed input is reset, then the next update state and static output are set to reset; the next update pulsed output is set to null; if the

current state is running, then procedure delevent(clksym,name,update) is called to delete the update event scheduled in the event file; the current state of the clock is set to reset; and the update time is set to tnow.

At the end, the execution state is set to idle and, if the update time equals tnow, then the update event of the clock is executed, else an update event of the clock is scheduled into the event file (10, 1877-1887). This completes the execution of the start event of a CLK component.

l. Procedure clkupdate(name:smname) (10, 1898): implements the update event of the CLK component. If the update time does not equal tnow, then procedure execerr(name,num,E3) is called (10, 1930); else, if the next pulsed output equals timeout, then the current pulsed output is set to timeout and procedure schdestxp(clksym,name,1) is called to load the pulsed output into its destinations; if the next static output does not equal the current static output, then the current static output is assigned with the next static output value and procedure schdestxs(clksym,name,1) is called to load the static output into its destination. The current state is updated and the next pulsed output is set to null. This completes the execution of the update event of a CLK component.

m. Procedure envst(name:smname) (10, 2155): implements the start event of the ENV component. Procedure envst starts out to check the execution state. If the execution state is not pend, then procedure execerr(name,num,E2) is called; else the execution state is set to

busy; the update time is set; if the ENV function is terminal, then procedure termfunction(name) is called to allow the user to examine and assign system variables via the terminal; else, if the ENV function is procedure, then procedure envfunction(procno) is called to execute the user defined ENV procedure; else procedure execerr(name,num,E28) is called (10, 2172-2175); the update time is set again in case the component execution time, texec, is changed during the execution of the ENV function; if the update time equals tnow, then the update event of the ENV component is executed, else an update event of the ENV component is scheduled into the event file. At the end, if a future start event of the ENV component is to be scheduled and the time for the start event is smaller than the next update time, then procedure execer(name,num,E29) is called. This completes the execution of the start event of an ENV component.

n. Procedure envupdate(name:smname) (10, 2214): implements the update event of the ENV component. Procedure envupdate starts out to check the execution state. If the execution state is not busy, then procedure execerr(name,num,E4) is called (10, 2277); else, if the update time does not equal tnow, then procedure execerr(name,num,E3) (10, 2274) is called, else procedure envupdate starts to update the ENV component (10, 2228-2270). The execution state is set to idle, all the current pulsed inputs are set to null, the current pulsed outputs are updated and loaded into their destinations, and the current static outputs are updated and loaded into their destinations. This completes the execution of the update event of the ENV component.

7. Execution of a terminal and procedure ENV

When a terminal ENV is executing, SAS will prompt the user with a '#' sign on the terminal. The user can use the on-line Terminal Mode Command Language, as described in Figure 3.14, to interactively query the status of the global system, change the check option of the ENV start expression, assign pulsed and static outputs of the currently executing ENV, store the status of a named component or all the system components, schedule a future start event and an update event of the currently executing ENV, and stop the simulation system execution.

Procedure termfunction(name:smname) (12, 867) implements the execution of a terminal ENV. Procedure termfunction starts out by printing the ENV name and the current simulation time on the terminal (12, 1222-1224). The input line variable, line[ii], is set to contain only blank characters. A '#' sign is printed on the terminal to indicate the system is in terminal mode (12, 1235). A line is read from the terminal and saved in the input line variable, line[ii] (12, 1238-1243). If the number of characters for each command line exceeds 132 characters, then the terminal mode error code, err, is set to 1 (12, 1250); else the first non-blank character is picked out to match with the terminal mode command code. If the non-blank character, cc, is 'P' or 'p', then procedure pout(name) is called to print out the component status; else, if cc equals 'S' or 's', then procedure sout(name) is called to save the component status in the file, sysfile; else, if cc equals 'A' or 'a', then procedure aout(name) is called to assign new values for the currently executing ENV parameters; else, if cc equals 'F'

Format	Description
P name	Print the status of the named component
P all	Print the status of all the system components
P eventfile	Print the contents of the event file
S name	Save the current status of the named component in the system data file
S all	Save the current status of all the system components in the system data file
A ZP.n := zp	Assign the value zp to the indexed pulsed output of the currently executing environment
A ZS.n := zs	Assign the value zs to the indexed static output of the currently executing environment
A STARTCHECK := option	Assign the value option (never, everytimechange or everyevent) to the currently executing environment
F T	Schedule a future start event for the currently executing environment at time T, T must be greater than or equal to tnow
F +T	Schedule a future start event for the currently executing environment at time tnow + T, T must be greater than or equal to zero
U T	Schedule an update event for the currently executing environment at time T, T must be greater than or equal to tnow, this command makes the currently executing environment busy until T
U +T	Schedule an update event for the currently executing environment at time tnow + T, T must be greater than or equal to zero, this command makes the environment busy until tnow + T
E	Exit from the terminal mode and continue system execution
H	Stop the system execution, exit from SAS

Figure 3.14. Terminal Mode Command Language

or 'f', then procedure fevent(name) is called to indicate a future start event of the currently executing ENV needs to be scheduleld; else, if cc equals 'U' or 'u', then procedure fupdate(name) is called to set the next update time of the currently executing ENV; else, if cc equals 'H' or 'h', then the system halt variable is set true to indicate the end of the system execution and the local variable exit is also set true to indicate the end of the terminal mode execution; else, if cc equals 'E' or 'e', then the local variable exit is set true; else, if cc equals ' ', then the local variable exit is set false to indicate a blank line and try again; else the terminal mode error variable, err, is set to 2 to indicate an unrecognized command (12, 1262-1275). If the error variable, err, is greater than zero, then procedure perr is called to print out the terminal mode command language error message (12, 1278). The process is repeated until the local variable exit is set true (12, 1280). This completes the execution of a terminal ENV.

The implementation details of the procedures which process the different terminal mode commands are not discussed in the dissertation but users can refer to these listings: procedure pout(name:smname) (12, 1141), sout(name:smname) (12, 1171), aout(name:smname) (12, 997), fevent(name:smname) (12, 1078), fupdate(name:smname) (12, 1112), and perr (12, 1199).

When a procedure ENV is to be executed, SAS calls the procedure envfunction(num:integer) (20, 824). Procedure envfunction, which is generated by the Transformation process, consists of all the user

defined ENV procedures. The proper ENV procedure is referenced by passing an integer, num, to the procedure envfunction(num:integer).

Within an ENV procedure, a set of SAS procedures can be called to emulate the Terminal Mode Command Language.

Procedure pname(name:smname) (12, 749) emulates the P name terminal command. Procedure pname calls procedure psm(output,name) to print out the status of the named component at the user terminal.

Procedure pall (12, 757) emulates the P all terminal command. Procedure pall calls procedure savesys(output) to print out the status of all the system components at the terminal. Procedure peventfile(outfile) (12, 837) emulates the P eventfile terminal command.

Procedure sname(name:smname) (12, 764) emulates the S name terminal command. Procedure sname calls procedure psm(sysfile,name) to write the status of the named component in the sysfile file, the system status data file.

Procedure sall (12, 770) emulates the S all terminal command. Procedure sall calls procedure savesys(sysfile) to write the status of all the system components in the sysfile file.

Procedure azp(num:integer;zpvalue:string) (12, 775) emulates the AZP.n:=zp terminal command. Procedure azp sets the next pulsed output variable, ntzp[num], to equal zpvalue. Procedure azs(num:integer; zsvalue:string) (12, 784) emulates the AZS.n:=zs terminal command. Procedure azs sets the next static output variable, ntzs[num], to equal zsvalue. Procedure astartcheck(check:option) (12, 790) emulates the

ASTARTCHECK:=option terminal command. Procedure astartcheck sets the check option variable, checkopt, to check.

Procedure fabs(time:real) (12, 796) emulates the F T terminal command. Procedure feventabs starts out to check if the next start event scheduling time is smaller than tnow. If so, then procedure execerr(csmname,num,E33) is called, else two global variables are set to indicate a start event of the component is to be scheduled at tstart (12, 803). Procedure finc(time:real) (12, 808) emulates the F +T terminal command. Procedure finc sets the two global variables to indicate a start event of the component is to be scheduled at tstart.

Procedure uabs(time:real) (12, 820) emulates the U T terminal command and procedure uinc(time:real) (12, 829) emulates the U +T terminal command.

8. Performance traces

SAS offers several trace functions in addition to those already described for the Terminal Mode Command Language. In particular, the user may specify that SAS sample and save the values of any specified set of SAN system variables or the value of a boolean function of the SAN system variables. The trace functions are invoked by augmenting the initial SAN system model with one or more instances of the three kinds of trace specifications: regular variable history, conditional variable history, and regular expression history. Their specification formats are shown in Figure 3.15.

```
VARHISTORY name : regular
   VARIABLES : varname, varname,... ;
   DTHISTORY : Thist ;
   CHECKOPT : never (or everytimechange or everyevent);
END;

VARHISTORY name : conditional
   VARIABLES : varname, varname,... ;
   CONDITION : boolean expression;
   CHECKOPT : never (or everytimechange or everyevent);
END;

EXPHISTORY name : regular;
   EXPRESSION : boolean expression;
   DTHISTORY : Thist ;
   CHECKOPT : never (or everytimechange or everyevent);
END;
```

Figure 3.15. Trace utilities specification formats

To specify a regular variable history instance, the VARHISTORY instance type is first declared and the unique instance name given. The word regular following the instance name indicates a regular variable history instance. Next a set of SAN global variable names are specified to indicate that SAS should sample and save the values of these variables.

The time interval at which the regular variable history trace is sampled is given in DHISTORY. The check option of the regular variable history trace is specified in the line CHECKOPT. A trace may be disabled by setting CHECKOPT to never. If the CHECKOPT is everytimechange, then SAS will take precisely one sample for the trace just before advancing simulated time forward from the current event time (i.e., after all model activity at the current event time has taken place). If CHECKOPT is set to everyevent, SAS will sample and record the designated variables after the execution of every event.

The conditional trace operates much the same as a regular trace, the difference being that the sample times are determined dynamically during simulation execution to correspond to those times when the specified expression evaluates to true. The expression is tested for the condition at the times specified in CHECKOPT. The regular expression history trace operates similarly to the regular variable trace; except that at the sampling time, the value of the boolean expression is recorded for the regular expression history and the value of a set of SAN variables are recorded for the regular variable history.

Procedure savehistreg(opt:option) (14, 744) implements the regular variable history instance trace recording process. The parameter opt

indicating whether the trace of the instance with check option everyevent or everytimechange is to be recorded. Procedure savehistory starts out by assigning temp to point to the beginning of the regular variable history record list. The data structures of different performance traces are described in Chapter III.B.5. (Readers are advised to refer to the data structures for better understanding.) Procedure savehistreg then goes through a while loop (14, 751-756) to examine all the regular variable history instances. It checks if the instance's check option equals opt and the next trace recording time equals tnow. If so, then procedure historynum(temp^.num) is called to record the value of each variable defined in the instance. The variable temp is set to point to the next instance record. The loop (14, 751-756) is repeated until all the regular variable history instances are checked.

Procedure historynum(num:integer) (20, 849) is generated by the Transformation process. Procedure historynum consists of a sequence of procedure calls to procedure history. In the example model, Simple.dat, there is only one regular variable history instance and only one variable trace is specified in the instance, that is why there is only one call to procedure history(fsm[fsm1].czs[1],1,1) (20, 854) to record the value of the variable, fsm[fsm1].czs[1]. The details on the generation of procedure historynum were described in Chapter IV.C.4.

Procedure history(tempvalue:string;m,n:integer) (14, 764) records the value of the n^{th} variable in the m^{th} regular variable history instance; n and m are the order of appearance in the SAN file.

Procedure history starts out to get a new statlink record (1, 503-508). The value of the variable and the current simulation time are recorded (14, 774-775). The local variable tempvarhist is set to point to the beginning of the regular variable history instance record list. The for loop (14, 779-780) moves the pointer pointing to the n^{th} regular variable history instance record. The local variable, temphist, is set to point to the record of the first variable. The for loop (14, 784-785) moves the pointer pointing to the record of the n^{th} variable. The statlink record, which holds the value of the variable and the current simulation time, is inserted in the beginning of the list of the data element of the variable. This completes the process of recording a value of a variable in an instance.

Procedure savehistcon(opt:option) (20, 874) implements the conditional variable history instance trace recording process, the parameter opt indicating whether the trace of the instance with check option everyevent or everytimechange is to be recorded. Procedure savehistcon is generated by the Transformation process. The details on the genertion of procedure savehistcon were described in Chapter III.C.5. Procedure savehistcon starts out by setting temp pointing to the beginning of the conditional variable history instance record list (20, 878). If a conditional variable history instance is specified to be recorded and the header of the instance record list is nil, then procedure execerr(csmname,num,E22) is called. In the example model, Simple.dat, there is only one conditional variable history instance and only one variable trace is specified in the instance. Line (20, 882)

checks if the instance's check option equals opt and function conexp(i) is called to check if the conditional expression is true. If they are both true, then procedure stathist(env[env2].zp[1],1,1) is called to record the value of the variable, env[env2],czp[1] in the conditional variable history instance record list. The implementation of procedure stathist(tempvalue:string;m,n:integer) (14, 795) is similar to that of procedure history(tempvalue:string,m,n:integer) (14, 764). Readers can refer to the description of procedure history to understand the implementation of procedure stathist.

Procedure saveexphist(opt:option) (14, 846) implements the regular expression history instance trace recording process, the parameter opt indicating whether the trace of the instance with check option everyevent or everytimechange is to be recorded. Procedure saveexphist starts out by assigning temp to point to the beginning of the regular expression history record list (14, 852). Procedure saveexphist then goes through a while loop (14, 855-861) to examine all the regular expression history instances. It checks if the instance's check option equals opt and the next trace recording time equals tnow. If so, then procedure insstatus(cexpst(i),temp) is called to record the boolean value of the expression in the instance. The local variable temp is set to point to the next instance record. The loop (14, 855-861) is repeated until all the regular expression history instances are checked.

Procedure insstatus(cst:boolean,tempexphist:exphistlink) (14, 826) starts out by getting a new status record (1, 513-518) and stores the boolean value, cst and the current simulation time in the new status

record (14, 832-835). The new status record is then inserted in the current regular expression history record pointed to by the pointer tempexphist (14, 839-840).

When procedure insstatus(cexpst(i),temp) is called inside procedure saveexphist, the boolean value, cst, is obtained by calling the function cexpst(i) (14, 858). Function cexpst(i:integer):boolean (20, 890) was generated by the Transformation process. This function consists of all the boolean expressions specified in the regular expression history instance. In the example model, Simple.dat, there is only one expression history trace to be recorded. The expression is evaluated and assigned to the function variable cexpst (20, 895).

9. Initialization specification

The user may optionally include an initialization specification as shown in Figure 3.16 in the SAN system specification. The user may assign the values of simulation time at which the run should begin (tbeg) and end (tend), else the default values 0.0 and 10.0 are assigned, respectively. The user may indicate whether SAS should trap multiple simultaneous pulsed inputs to single components, other than environments for which the trap function is individually specified. If the SAS variable mulpulsecheck is set to true, then SAS will trap the above mentioned multiple simultaneous pulsed inputs. The default value for mulpulsecheck is false. The user may also reassign the seed value of the random number generating function, seed. The default value for seed is 23467823.

```
INIT
   tbeg : Tbeg;
   tend : Tend;
   mulpulsecheck : true ( or false)
   seed : integer;
END;
```

Figure 3.16. Initialization specification format

IV. APPLICATION OF THE STATE ARCHITECTURE NOTATION AND STATE ARCHITECTURE SIMULATOR IN SIMULATING DATA COMMUNICATION PROTOCOLS

This chapter explains the operational steps in running SAS. It begins with the presentation of a SAN model, moves to the Transformation process and through to the creation of the SAN model executable image. This chapter also describes the steps involved in executing the SAN model executable image and different steps in correcting SAN model specification errors. The last three sections illustrate the use of the SAN and the SAS in simulating data communication protocols by three examples. The first example simulates a small size discrete system with three components. The purpose of this example is to walk through the steps in specifying a SAN model, creating an executable image of the SAN model and executing the SAN model in the SAS environment. The second example simulates the start-stop protocol [Piatkowski 1981]. We use all nine kinds of components in the Start-Stop SAN model. The last example illustrates the use of the SAN and the SAS in simulating a fairly complicated data communication protocol, the Advanced Data Communication Control Procedures (ADCCP) [Piatkowski 1979].

A. Operational Steps in Running SAS

In using SAS to exercise a SAN model, there are two main steps to follow.

1. Specification

The user specifies the system using the SAN formats as described in Chapter II and stores the specification in a data file e.g. simple.dat;1, where "simple" is the name of the data file.

2. Creation and execution of an executable image

The user invokes the SAS procedure SAS.COM to create an executable image for the simulated system. There are five steps in the creation and execution of the executable image, namely TRANSFORM, APPEND, COMPILE, LINK and RUN.

a. TRANSFORM: performs the Transformation process as described in Chapter III.C. Before executing the SAN model, SAS scans the SAN model and compiles the procedures and boolean expressions into global PASCAL procedures. Actually, SAS generates ten different procedures as described in the Transformation process. They are as follows:

fsmfunc.pas	contains all the procedures declared in the FSM instances
cfpfunc.pas	contains all the procedures declared in the CFP instances
cfsfunc.pas	contains all the procedures declared in the CFS instances
envfunc.pas	contains all the procedures declared in the ENV instances

envexpfil.pas — contains all the boolean expressions declared in the ENV instances

histfile.pas — contains the procedure historynum(num:integer) which is used to sample and save system variables defined in the regular history instances

statsup.pas — contains the function conexp(num:integer) :boolean, which holds all the boolean expressions defined in the conditional variable history

conhistfil.pas — contains the procedure savehistcon(opt: option), which is used to save system variables defined in the conditional variable history instances

cexpfile.pas — contains the function cexpst(num:integer): boolean, which holds all the boolean expressions defined in the regular expression history instances

iniset.pas — establishes the enumerated names of all instances corresponding to the component name; establishes an array of string variables which contains the strings declared in the procedure of the SAN specification.

b. APPEND: merges the above ten procedures together into one PASCAL module, USER.PAS.

c. COMPILE: compiles the USER.PAS module to produce an object mode file, USER.OBJ.

d. LINK: links the object module, USER.OBJ, with the SAS predefined library SAS.OLB to produce an executable image for the simulated system, namely SAS.EXE

e. RUN: executes the executable image of the simulated system.

To invoke the SAS command procedure, the user types in the following command:

"@sas filename step".

Two parameters must be supplied to the command procedure: "filename" representing the SAN file containing the SAN model and "step" representing the starting step of the SAS command procedure. There are five steps in the SAS command procedure as mentioned above. Before a simulated system is executed, its executable image has to be created first. To do this, the user should invoke the command procedure with step="TRANSFORM" to perform the Transformation process. For example:

"@sas filename TRANSFORM".

SAS will print out the component names in the SAN model on the terminal, when each component is being transformed. The details of the SAS output message on the terminal will be demonstrated via an example in the next section. If there is any error in the Transformation process, then SAS prints out the following message on the terminal:

"TRANSFORMATION ERRORS

ERRORS . . . Take appropriate step to do correction

To check error, type errfile.dat",

The details of the error correction steps will be explained later in this section. If there is no error in the Transformation process, the ten different procedures are generated as mentioned above and SAS prints out the following message:

"TRANSFORMATION IS GOOD

Append user Module".

The above message indicates that SAS is in the APPEND step to merge all the procedures into a PASCAL module, USER.PAS. At the end of the APPEND step, SAS continues on to the COMPILE step and prints out the following message:

"Compile USER.PAS".

In this step, the module USER.PAS is submitted to the PASCAL compiler to generate the object module, USER.OBJ. If there is any error in compiling the USER.PAS, the PASCAL compiler will print out the error message on the terminal and SAS will stop and print out the following message:

"SAS TERMINATED".

If there is no error in compiling the USER.PAS, SAS will print out the following message:

"Link USER.OBJ with SAS.OLB".

The above message indicates that the USER.OBJ is linked with the SAS predefined library, SAS.OLB, to produce the simulated system executable image, SAS.EXE. At the end of the LINK step, SAS prints out the following message:

"System is ready to run

To run the system just type 'run'

run or exit".

At this point, the user can type in 'run' to execute the simulation system. If the user types in 'exit' then SAS exits from its command procedure and returns to the VAX command mode, which will print out a '$' sign to indicate the VAX command mode. After the simulated system executable image is created, the user can invoke the SAS command procedure to execute the system without going through the Transformation process again.

"@sas filename run"

There are three major parts in the execution of the simulated system under the SAS environment. First SAS will call the Data Input process to read the SAN specification of the simulated system. In case of any SAN syntax error, SAS prints out an error message on the terminal and stops. After SAS executes its Data Input process, it will ask the user if the current simulated system execution starts with New Initialization or Restart Initialization. SAS prompts the user with the message on the terminal:

"New Initialization: Y=Yes or N=No".

A user response of 'N' or 'n' means the current execution starts with Restart Initialization; otherwise, the current simulated system execution starts with New Initialization. If it is a New Initialization, SAS will check if the simulated system is initially stable or not. If the system is unstable, SAS stops and prompts the

user with a message:

"SYSTEM IS UNSTABLE".

If the system is stable, SAS will prompt the user with the message:

"SYSTEM IS STABLE".

SAS then starts to execute the System Executive. In case an execution error occurs, the System Executive releases its control to the System-Monitor which allows the user to examine the current system status. The user can use the Terminal Mode Command Languages to examine the status of all the system components. The user may decide to continue or halt the system execution. If an execution error occurs, the execution of the event of the error component is aborted. If the user decides to continue on the system execution, certain unexpected system execution errors may be generated because of the previous execution error. Examples of the use of the Terminal Mode Command Languages will be discussed in the next section. If no execution error occurs, the System Executive continues execution of the simulated system until the current simulation time exceeds the ending time of the simulation or the system halt variable is set to true. When the execution of the System Executive finishes, SAS asks the user if the current simulated system status need to be saved by prompting the following message:

"Do you want to save the current simulated system status?

Y=Yes N=No?".

A user response of 'Y' or 'y' means the current simulated system status will be saved in the rsfile.dat file for future Restart Initialization.

At the end of system execution, the system variable traces are stored in the datafile.dat file; and if any system status information had been saved during the execution, they are stored in the sysfile.dat file. In the next section, we will describe several examples of using SAN and SAS to simulate systems.

3. General steps on errors correction

In case any error occurred in the process of creating an executable image of the simulated system, the user has to take appropriate steps to make corrections and invoke the SAS command procedure again. Depending on which step the error occurred in, different corrections are needed.

If an error occurred in the TRANSFORM step, then the user should take the following actions:

a) Examine the errfile.dat file to find out what kind of errors occurred in the Transformation process. Error messages are located at the end of the errfile.dat file.

b) Make necessary corrections to the SAN file as suggested by the error message.

c) Try again by typing "@sas filename transform".

If any error occurred in the COMPILE step, then the user should take the following actions:

a) Examine the user.lis file to locate the error; user.lis is a source listing file generated by the PASCAL compiler in compiling user.pas.

b) Make necessary corrections to the SAN file as suggested by the error messages.

c) Try again by typing "@sas filename transform".

No error should occur in the APPEND and LINK steps. In case there is an error, the error should be reported to the implementor or maintainer of SAS.

At the RUN step, there are three major parts in the execution of of the simulated system under the SAS environment as mentioned in the previous sub-section. During the Data Input process, if a SAN syntax error occurred, the user should take the following correction steps:

a) Examine the errfile.dat file to locate the error. Error messages are located at the end of the errfile.dat file.

b) Make necessary corrections to the SAN file as suggested by the error messages.

c) Users are encouraged to consider trying the RUN step again by typing "@sas filename run". To start at the RUN step, the users should make sure that none of the following SAN specifications were changed in correcting the error:

   i) Deletion or addition of a component instance,

   ii) Change of instance name,

   iii) Reordering of the instances in the specification SAN file,

iv) Changes in procedure or boolean expression declaration,

v) Changes of variable names in the variable history trace.

d) Even though it is usually safe to restart from the RUN step, if the above changes are not made, the user can always start from the TRANSFORM step by typing "@sas filename transform". In this case, user.pas, is always compatible to the SAN model.

During the New Initialization process, if the simulated system is unstable, the user should take the following corrective steps:

a) Examine the sysfile.dat file to locate which components are initially unstable.

b) Make necessary corrections to the initial values of the system components, or to the system logic by addition or deletion of components, or charging the system component interconnections.

c), d) Same as in the Data Input process.

During the System Executive process, SAS may encounter some system execution errors. When this happens, SAS will print out the error message identifying the faulty component, and release control to the System-Monitor, which allows the user to examine the current system status. The user can use the Terminal Mode Command Language to examine the status of all system components and decide to continue or halt the system execution. Details using the Terminal Mode Command Languages will be shown in the next section.

B. A Simple Discrete System Simulation

1. Description of the SAN model

This sub-section illustrates the use of the SAN and the SAS to simulate discrete systems via a simple discrete system, stored in the file, simple.dat. A block diagram of the system is shown in Figure 4.1 and the SAN model is listed in Figure 4.2. Users may notice four different syntax errors in the SAN model of Figure 4.2 shown in circles. Those errors are purposely left in the SAN model for the demonstration of error correction steps in the creation and execution of the SAN model executable image.

The system has three components. One of them is a terminal ENV component named Terminal. The execution time of Terminal is zero and the start expression check option (STARTEXPCHECK) is everytimechange, which allows the start event of Terminal to be scheduled after all the events in the current event set are executed. Terminal has a pulsed output connected to the pulsed input of the ENV component named Pulsegen. Pulsegen is a procedure ENV. The execution time of Pulsegen is one time unit and the STARTEXPCHECK is never. Upon the arrival of a start pulse from Terminal, Pulsegen will produce either a 'reset' or 'inc' pulse at its pulsed output. Procedure Pulsegenproc performs the function of randomly producing either a 'reset' or 'inc' pulse. The pulsed output is loaded into its destination component named Counter. Counter is a FSM component with execution time equal to zero. Counter records the number of 'inc' pulses that have arrived at its pulsed input

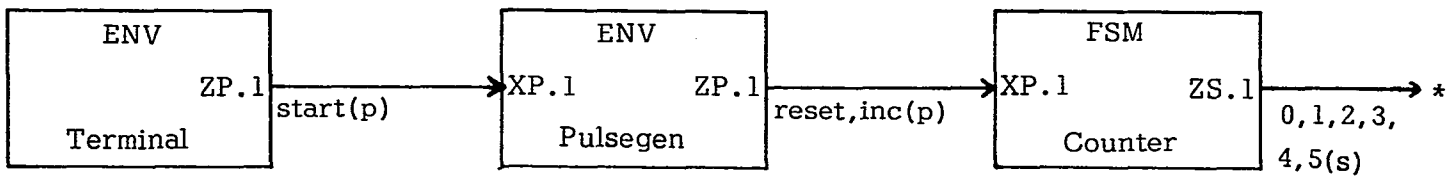


Figure 4.1. Block diagram of a simple discrete system

```
ENV Terminal : interlaced
        ZP.1 TO Pulsegen.XP.1 : start;
        FUNCTION : terminal;
        deftexec : 0.0;
        STARTEXP : true;
        STARTEXPCHECK : everytimechange;
END;

ENV Pulsegen : interlaced
        XP.1 FROM Terminal.ZP.1 : start;
        ZP.1 TO Counter.XP.1 : reset,inc;
        FUNCTION : procedure
          procedure Pulsegenproc;
          var i:integer;
              a:array[1..2] of string;
          begin
              a[1]:='reset                                                    ';
              a[2]:='inc                                                      ';
              i:=randint(5);
              with env[USPulsegen] , updatestate do
              begin
                  if(i=3) then ntzp[1]:=a[1]
                          else ntzp[1]:=a[2];
              end; {end with}
          end;
        END;
        deftexec:1.0;
        STARTEXP : false;
        STARTEXPCHECK : never;
END;

FSM Counter : interlaced
        S : 0,1,2,3,4,5;
        defsinit : 0;
        XP.1 FROM Pulsegen.ZS.1 : reset,inc;
        ZS.1 UNCONNECTED : 0,1,2,3,4,5;
        FNS : procedure
          if cxp.1='reset' then nts:='0'
          else
          begin
              if cs='0' then nts:='1'
              else if cs='1' then nts:='2'
              else if cs='2' then nts:='3'
              else if cs='3' then nts:='4'
              else if cs='4' then nts:='5'
              else nts:='5';
          end;
        END;
        FOUTS : procedure
          ntzs.1:=nts;
        END;
        deftexec : 0.0;
END;

VARHISTORY TraceCount : regular
        VARIABLES : Counter.czs.1;
        DTHISTORY : 1.0;
        CHECKOPT : everytimechange;
END;

VARHISTORY TracePulse : conditional
        VARIABLES : Pulsegen.czp.1;
        CONDITION : Pulsegen.czp.1='reset';
        CHECKOPT : everytimechange;
END;

EXPHISTORY Tracestatus : regular
        EXPRESSION : (Pulsegen.czp.1='inc')and(Counter.czs.1='1');
        DTHISTORY : 1.0;
        CHECKOPT : everytimechange;
END;

INIT
        tbeg : 1.0;
        tend : 10.0;
END;
```

Figure 4.2. The SAN model of a simple discrete system

up to five. Whenever a 'reset' pulse arrives at Counter, the state of Counter will be set to '0'. Both the FNS and FOUTS of Counter are specified via a procedure. The SAN model also includes three different kinds of performance traces. The regular variable history instance, TraceCount, records the static output of Counter every time unit; it records just before simulation time is advanced. The conditional variable history instance TracePulse, records the pulsed output of Pulsegen whenever its value is 'reset'. The regular expression history records the boolean value of the expression (Pulsegen.czp.1='inc') and (Counter.czs.1='1') every time unit, it records just before simulation time is advanced. At the end, the Initialization instance sets the simulation beginning time, tbeg, and the simulation ending time, tend.

2. Creation of the simulated system executable image

The SAN model of Figure 4.2 is saved in file simple.dat. The simulated system executable image is created by invoking the SAS command procedure as shown in Figure 4.3. For clear exposition, we have underlined the commands typed in by users in Figure 4.3 and added some line numbers.

After the SAN model is created, the user types in the command in line (1) to invoke the command procedure SAS.COM by passing the SAN model file name, simple.dat, and the step, transform(1). SAS will start the Transformation process. During the Transformation process, SAS prints out the component names being processed on the terminal ("csm:= Terminal" means that the current state machine is Terminal). After

```
(1) $ @sas counter.dat transform
      PREVIOUS LOGICAL NAME ASSIGNMENT REPLACED
    csm:= Terminal
    DONE
(2) TRANSFORMATION ERRORS
(3) ERRORS...Take appropriate step to do correction
(4) To check error, type errfile.dat
(5) $ type errfile.dat
    ENV Terminal : interlaced
            ZP.1 TO Pulsesen.XP.1 : start;
            FUNCTION : terminal;
            deftexec : 0.0;
            STARTEXP : true;
            STARTEXPCHECK : everytimechange;
    END;

(6) EVN ^^^Pulsesen : interlaced
(7) error number =  33  in the line number          9
(8) error: undesired name......
(9)        It can only be FSM, CFP, CFS, DELP,
(10)        DELS, CLK, DER, QUE, ENV, VARHISTORY
(11)       EXPHISTORY, INIT, OR END.
(12) $ ed counter.dat
    Edit: DRB0:[B017044]COUNTER.DAT;1
(13) *sEVN$ENV$9
    00009    ENV Pulsesen : interlaced
(14) *eb
    [DRB0:[B017044]COUNTER.DAT;1]
(15) $ @sas counter.dat transform
      PREVIOUS LOGICAL NAME ASSIGNMENT REPLACED
    csm:= Terminal
    csm:= Pulsesen
    csm:= Counter
    DONE
(16) TRANSFORMATION ERRORS
(17) ERRORS...Take appropriate step to do correction
(18) To check error, type errfile.dat
(19) $ type errfile.dat
    ENV Terminal : interlaced
            ZP.1 TO Pulsesen.XP.1 : start;
            FUNCTION : terminal;
            deftexec : 0.0;
            STARTEXP : true;
            STARTEXPCHECK : everytimechange;
    END;
```

Figure 4.3. Example run to create an executable image for the SAN model, simple.dat

```
    FSM Counter : interlaced
            S : 0,1,2,3,4,5;
            defsinit : 0;
            XP.1 FROM Pulsegen.ZS.1 : reset,inc;
            ZS.1 UNCONNECTED : 0,1,2,3,4,5;
            FNS : procedure
              if cxp.1='reset' then nts:='0'
              else
              begin
                  if cs='0' then nts:='1'
(20)              else if xs='???'1' then nts:='2'
(21) error number =  74  in the line number        42
(22) error : unexpected character string
(23) $ ed counter.dat
     Edit: DRB0:[B017044]COUNTER.DAT;1
(24) *sxs$cs$42
     00042                    else if cs='1' then nts:='2'
(25) *eb
     [DRB0:[B017044]COUNTER.DAT;1]
(26) $ @sas counter.dat trnsform
       PREVIOUS LOGICAL NAME ASSIGNMENT REPLACED
(27) Steps unclear
(28) SAS TERMINATED
(29) $ @sas counter.dat transform
       PREVIOUS LOGICAL NAME ASSIGNMENT REPLACED
(30) csm:= Terminal
(31) csm:= Pulsegen
(32) csm:= Counter
(33) VARHIST name:= TraceCount
(34) VARHIST name:= TracePulse
(35) EXPHIST name := Tracestatus
(36) DONE
(37) TRANSFORMATION IS GOOD
(38) Append user module
(39) Compile user.pas
(40) Link user.obj with sas library
(41) System is ready to run
(42) To run the system just type 'run'
(43) run or exit: exit
     $
```

processing the component, Terminal, the Transformation process encounters a SAN specification error. SAS then terminates the Transformation process and prints out lines (2), (3), and (4) on the terminal. In (5), we type the errfile.dat file to locate the error. From (6) to (11), we notice that there is a misspelling of the instance kind ENV. From (12) to (14), we edit the simple.dat file to change the misspelled EVN to ENV. In (15), the SAS command procedure is invoked again to start from the Transformation process. While the Transformation process scans the component, Counter, it stops and indicates a Transformation error in (16) to (18). In (19), we type the errfile.dat file to locate the error. From (20) to (22), we notice that the string 'xs' can not exit by itself within a local procedure. The string can either be 'xs.n', 'xp.n', or 'cs', where n is an integer. In the example SAN model, the string 'cs' is expected at the above position. From (23) to (25), we edit the simple.dat file to change the string 'xs' to 'cs'. In (26), the SAS command procedure is invoked again, but this time the step is misspelled. SAS responds with the messages in (27) and (28) on the terminal. In (29), we invoke the SAS command procedure again. This time SAS prints out all the instance names and the message "TRANSFORMATION IS GOOD" to indicate that there is no SAN error detected in the Transformation process from (30) to (37). SAS then prints out the message "Append user module" to indicate that SAS is merging all the transformed procedures into a user module, user.pas. In (39), SAS prints out the message "Compile user.pas" to indicate that SAS had submitted the user.pas file to the PASCAL compiler

to compile the user.pas file. There is no compilation error in the user.pas file. In (40), SAS points out the message "Link user.obj with sas library" to indicate that the PASCAL linker is linking the user.obj with the sas.olb to produce an executable image of the SAN model. At the end of the linking process, SAS prompts the user with the messages in (41) to (43) to ask the user if he wants to execute the SAN model executable image or stop at this point. We type in the word 'exit' in (43) to exit from the SAS command procedure. This completes the creation of a SAN model executable image.

3. Execution of the SAN model executable image

In this sub-section, we will walk through the detailed steps to execute the SAN model in the file simple.dat. Figure 4.4 consists of a list of commands and responses in executing the SAN model executable image (simulated system). For clear exposition, the lines typed by the users are underlined and some line numbers added. In the actual run, the underlines do not appear. In line (1), we type in the command to invoke the SAS command procedure to execute the simulated system. While the SAS scans the SAN file, simple.dat, the SAS detects an undefined component name (smname) in line (10) of the simple.dat file. In (2), (3), and (4), the VAX editing commands are used to change the component name from Termainal to Terminal. In (5), we invoke the SAS command procedure to execute the simulated system again. This time we encounter another syntax error in (6). In (7), (8) and (9), we use the VAX editing commands to change 'ZS' to 'ZP'.

```
(1) $ @sas counter.dat run
     PREVIOUS LOGICAL NAME ASSIGNMENT REPLACED
     PREVIOUS LOGICAL NAME ASSIGNMENT REPLACED
   tempsm=Terminal
   tempsm=Pulsegen
   DONE
   ENV Terminal : interlaced
           ZP.          1 TO Pulsegen.XP.          1 : start;
           FUNCTION : terminal;
           deftexec : 0.000000000E+00;
           STARTEXP : true;
           STARTEXPCHECK : everytimechange;
   END;

   ENV Pulsegen : interlaced
           XP.          1 FROM Termainal.^^^ZP.1 : start;
   error number =  34  in the line number        10
   error: undefined smname
   Total number of errors is         1
(2) $ ed counter.dat
   Edit: DRB0:[B017044]COUNTER.DAT;1
(3) *sTermainal$Terminal$10
   00010           XP.1 FROM Terminal.ZP.1 : start;
(4) *eb
   [DRB0:[B017044]COUNTER.DAT;1]
(5) $ @sas counter.dat run
     PREVIOUS LOGICAL NAME ASSIGNMENT REPLACED
     PREVIOUS LOGICAL NAME ASSIGNMENT REPLACED
   tempsm=Terminal
   tempsm=Pulsegen
   tempsm=Counter
   DONE
   ENV Terminal : interlaced
           ZP.          1 TO Pulsegen.XP.          1 : start;
           FUNCTION : terminal;
           deftexec : 0.000000000E+00;
           STARTEXP : true;
           STARTEXPCHECK : everytimechange;
   END;

   ENV Pulsegen : interlaced
           XP.          1 FROM Terminal.ZP.          1 : start;
           ZP.          1 TO Counter.XP.          1 : reset,inc;
           FUNCTION : procedure
             function randint(max:integer):integer;extern;
             procedure Pulsegenproc;
             var i:integer;
                 a:array[1..2] of string;
             begin
```

Figure 4.4. Example run in executing the SAN model executable image

```
                    a[1]:='reset                                          ';
                    a[2]:='inc                                            ';
                    i:=randint(5);
                    with env[U$Pulsesen] , updatestate do
                    begin
                        if (i=3) then ntzp[1]:=a[1]
                                 else ntzp[1]:=a[2];
                    end; {end with}
                end;
            END;
            deftexec: 1.000000000E+00;
            STARTEXP : false;
            STARTEXPCHECK : never;
    END;

    FSM Counter : interlaced
            S : 0,1,2,3,4,5;
            defsinit : 0;
(6)         XP.          1 FROM Pulsesen.ZS.???1 : reset,inc;
    error number =  29  in the line number        36
    error:ZP is expected.
    Total number of errors is          1
(7) $ ed counter.dat
    Edit: DRB0:[B017044]COUNTER.DAT;1
(8) *sZS$ZP$36
    00036            XP.1 FROM Pulsesen.ZP.1 : reset,inc;
(9) *eb
    [DRB0:[B017044]COUNTER.DAT;1]
(10) $ @sas counter.dat run
      PREVIOUS LOGICAL NAME ASSIGNMENT REPLACED
      PREVIOUS LOGICAL NAME ASSIGNMENT REPLACED
(11) tempsm=Terminal
(12) tempsm=Pulsesen
(13) tempsm=Counter
(14) New Initialization ? Y=Yes or N=No. Y
    SYSTEM INITIALIZATION : CHECKING FOR UNSTABLE MACHINE
(15)  **  SYSTEM IS STABLE **
    SYSTEM HAS BEEN STARTED
(16) ---- SCHEDULE ----  ENV  Terminal    START   1.000000000E+00
    EXECUTE :  ENV  Terminal    START   1.000000000E+00

(17) ENV : Terminal    TNOW:=  1.000000000E+00
(18) #p all
    SYSTEM STATUS AT TNOW := 1.000000000E+00

    FSM     Counter
    CS:= 0
    cxp. 1:= -
    czs. 1:= 0
    Execution Delay:=  0.000000000E+00
    Execstatus:=  IDLE
```

Figure 4.4. (continued)

```
     ENV      Terminal
     czp. 1:= -
     Execution time:=  0.000000000E+00
     Execstatus:=  BUSY
     STARTEXPCHECK:= EVERYTIMECHANGE
     updatetime:= 1.000000000E+00
     ntzp. 1:=-

     ENV      Pulsegen
     cxp. 1:= -
     czp. 1:= -
     Execution time:=  1.000000000E+00
     Execstatus:=  IDLE
     STARTEXPCHECK:=            NEVER
(19) #s all
(20) #a ZP.1:=strt
     zp or zs number:=          1
     errnum:=4....output value is not valid
(21) #a ZP.1:=start
     zp or zs number:=          1
(22) #e
(23) ---- SCHEDULE ----  ENV  Pulsegen    START   1.000000000E+00
(24) EXECUTE :  ENV  Pulsegen    START   1.000000000E+00
(25) ---- SCHEDULE ----  ENV  Pulsegen   UPDATE   2.000000000E+00
(26) ---- SCHEDULE ----  ENV  Terminal    START   1.000000000E+00
(27) EXECUTE :  ENV  Terminal    START   1.000000000E+00

     ENV : Terminal     TNOW:=  1.000000000E+00
(28) #p eventfile
     Eventset.time:= 1.000000000E+00     Eventset.count:=          0

     Eventset.time:= 2.000000000E+00     Eventset.count:=          1
       UPDATE   ENV    Pulsegen
(29) #e
(30) EXECUTE :  ENV  Pulsegen   UPDATE   2.000000000E+00
(31) ---- SCHEDULE ----  FSM  Counter    START   2.000000000E+00
(32) EXECUTE :  FSM  Counter    START   2.000000000E+00
(33) ---- SCHEDULE ----  ENV  Terminal    START   2.000000000E+00
     EXECUTE :  ENV  Terminal    START   2.000000000E+00

(34) ENV : Terminal     TNOW:=  2.000000000E+00
(35) #w all
     errnum:=2....unrecognized command
(36) #p all
     SYSTEM STATUS AT TNOW := 2.000000000E+00

     FSM      Counter
(37) CS:= 1
     cxp. 1:= -
     czs. 1:= 1
```

Figure 4.4. (continued)

```
     Execution Delay:=  0.000000000E+00
     Execstatus:=  IDLE

     ENV      Terminal
     czp. 1:= -
     Execution time:=  0.000000000E+00
     Execstatus:=  BUSY
     STARTEXPCHECK:= EVERYTIMECHANGE
     updatetime:= 2.000000000E+00
     ntzp. 1:=-

     ENV      Pulsegen
     cxp. 1:= -
     czp. 1:= inc
     Execution time:=  1.000000000E+00
     Execstatus:=  IDLE
     STARTEXPCHECK:=            NEVER
(38) #a ZP.1:=start
     zp or zs number:=          1
(39) #e
(40) ---- SCHEDULE ----  ENV  Pulsegen    START   2.000000000E+00
(41) EXECUTE :  ENV  Pulsegen    START   2.000000000E+00
(42) ---- SCHEDULE ----  ENV  Pulsegen   UPDATE   3.000000000E+00
(43) ---- SCHEDULE ----  ENV  Terminal    START   2.000000000E+00
(44) EXECUTE :  ENV  Terminal    START   2.000000000E+00

     ENV : Terminal     TNOW:=  2.000000000E+00
(45) #p Pulsegen
     ENV      Pulsegen
     cxp. 1:= start
     czp. 1:= inc
     Execution time:=  1.000000000E+00
     Execstatus:=  BUSY
     STARTEXPCHECK:=            NEVER
     updatetime:= 3.000000000E+00
     ntzp. 1:=inc
(46) #e
(47) EXECUTE :  ENV  Pulsegen   UPDATE   3.000000000E+00
(48) ---- SCHEDULE ----  FSM  Counter    START   3.000000000E+00
(49) EXECUTE :  FSM  Counter    START   3.000000000E+00
(50) ---- SCHEDULE ----  ENV  Terminal    START   3.000000000E+00
     EXECUTE :  ENV  Terminal    START   3.000000000E+00

(51) ENV : Terminal     TNOW:=  3.000000000E+00
(52) #p Counter
     FSM      Counter
(53) CS:= 2
     cxp. 1:= -
     czs. 1:= 2
     Execution Delay:=  0.000000000E+00
     Execstatus:=  IDLE
```

Figure 4.4. (continued)

```
(54) #aZP.1:=start
     zp or zs number:=          1
(55) #e
     ---- SCHEDULE ----  ENV  Pulsegen     START    3.000000000E+00
     EXECUTE :  ENV  Pulsegen     START    3.000000000E+00
     ---- SCHEDULE ----  ENV  Pulsegen    UPDATE    4.000000000E+00
     ---- SCHEDULE ----  ENV  Terminal     START    3.000000000E+00
     EXECUTE :  ENV  Terminal     START    3.000000000E+00

     ENV : Terminal     TNOW:=  3.000000000E+00
(56) #p Pulsegen
     ENV      Pulsegen
     cxp. 1:= start
     czp. 1:= inc
     Execution time:=  1.000000000E+00
     Execstatus:=  BUSY
     STARTEXPCHECK:=            NEVER
     updatetime:= 4.000000000E+00
     ntzp. 1:=inc
(57) #e
     EXECUTE :  ENV  Pulsegen    UPDATE    4.000000000E+00
     ---- SCHEDULE ----  FSM  Counter     START    4.000000000E+00
     EXECUTE :  FSM  Counter     START    4.000000000E+00
     ---- SCHEDULE ----  ENV  Terminal     START    4.000000000E+00
     EXECUTE :  ENV  Terminal     START    4.000000000E+00

     ENV : Terminal     TNOW:=  4.000000000E+00
(58) #a ZP.1:=start
     zp or zs number:=          1
(59) #e
     ---- SCHEDULE ----  ENV  Pulsegen     START    4.000000000E+00
     EXECUTE :  ENV  Pulsegen     START    4.000000000E+00
     ---- SCHEDULE ----  ENV  Pulsegen    UPDATE    5.000000000E+00
     ---- SCHEDULE ----  ENV  Terminal     START    4.000000000E+00
     EXECUTE :  ENV  Terminal     START    4.000000000E+00

     ENV : Terminal     TNOW:=  4.000000000E+00
(60) #p Pulsegen
     ENV      Pulsegen
     cxp. 1:= start
     czp. 1:= inc
     Execution time:=  1.000000000E+00
     Execstatus:=  BUSY
     STARTEXPCHECK:=            NEVER
     updatetime:= 5.000000000E+00
     ntzp. 1:=inc
(61) #e
     EXECUTE :  ENV  Pulsegen    UPDATE    5.000000000E+00
     ---- SCHEDULE ----  FSM  Counter     START    5.000000000E+00
     EXECUTE :  FSM  Counter     START    5.000000000E+00
     ---- SCHEDULE ----  ENV  Terminal     START    5.000000000E+00
```

Figure 4.4. (continued)

```
    EXECUTE :  ENV  Terminal     START    5.000000000E+00

    ENV : Terminal     TNOW:=  5.000000000E+00
(62)#p Counter
    FSM      Counter
    CS:= 4
    cxp. 1:= -
    czs. 1:= 4
    Execution Delay:=  0.000000000E+00
    Execstatus:=  IDLE
(63)#a ZP.1:=start
    zp or zs number:=          1
(64)#
    e
    ---- SCHEDULE ----  ENV  Pulsegen     START    5.000000000E+00
    EXECUTE :  ENV  Pulsegen     START    5.000000000E+00
    ---- SCHEDULE ----  ENV  Pulsegen    UPDATE    6.000000000E+00
    ---- SCHEDULE ----  ENV  Terminal     START    5.000000000E+00
    EXECUTE :  ENV  Terminal     START    5.000000000E+00

    ENV : Terminal     TNOW:=  5.000000000E+00
(65)#p Pulsegen
    ENV      Pulsegen
    cxp. 1:= start
    czp. 1:= inc
    Execution time:=  1.000000000E+00
    Execstatus:=  BUSY
    STARTEXPCHECK:=            NEVER
    updatetime:= 6.000000000E+00
    ntzp. 1:=inc
(66)#e
    EXECUTE :  ENV  Pulsegen    UPDATE    6.000000000E+00
    ---- SCHEDULE ----  FSM  Counter     START    6.000000000E+00
    EXECUTE :  FSM  Counter     START    6.000000000E+00
    ---- SCHEDULE ----  ENV  Terminal     START    6.000000000E+00
    EXECUTE :  ENV  Terminal     START    6.000000000E+00

    ENV : Terminal     TNOW:=  6.000000000E+00
(67)#p Counter
    FSM      Counter
    CS:= 5
    cxp. 1:= -
    czs. 1:= 5
    Execution Delay:=  0.000000000E+00
    Execstatus:=  IDLE
(68)#a ZP.1:=stabt
    zp or zs number:=          1
(69)#e
    ---- SCHEDULE ----  ENV  Pulsegen     START    6.000000000E+00
    EXECUTE :  ENV  Pulsegen     START    6.000000000E+00
    ---- SCHEDULE ----  ENV  Pulsegen    UPDATE    7.000000000E+00
```

Figure 4.4. (continued)

```
     ---- SCHEDULE ----  ENV  Terminal     START    6.000000000E+00
     EXECUTE :  ENV  Terminal     START    6.000000000E+00

(70) ENV : Terminal     TNOW:=  6.000000000E+00
(71) #P PUlsegen
     errnum:=5....smname is not defined
(72) #P Pulsegen
     ENV      Pulsegen
     cxp. 1:= start
     czp. 1:= inc
     Execution time:=  1.000000000E+00
     Execstatus:=  BUSY
     STARTEXPCHECK:=             NEVER
     updatetime:= 7.000000000E+00
     ntzp. 1:=inc
(73) #a STARTCHECK:=never
     CHECKOPT:=             NEVER
(74) #f9.0
(75) #e
(76) EXECUTE :  ENV  Pulsegen    UPDATE    7.000000000E+00
(77) ---- SCHEDULE ----  FSM  Counter     START    7.000000000E+00
(78) EXECUTE :  FSM  Counter     START    7.000000000E+00
(79) ---- SCHEDULE ----  ENV  Terminal     START    9.000000000E+00
(80) EXECUTE :  ENV  Terminal     START    9.000000000E+00

     ENV : Terminal     TNOW:=  9.000000000E+00
(81) #a ZP1:=start
     errnum:=10...ZP or ZS or CHECKOPT is expected
(82) #a ZP.1:=start
     zp or zs number:=          1
(83) #e
     ---- SCHEDULE ----  ENV  Pulsegen     START    9.000000000E+00
     EXECUTE :  ENV  Pulsegen     START    9.000000000E+00
     ---- SCHEDULE ----  ENV  Pulsegen    UPDATE    1.000000000E+01
     EXECUTE :  ENV  Pulsegen    UPDATE    1.000000000E+01
     ---- SCHEDULE ----  FSM  Counter     START    1.000000000E+01
     EXECUTE :  FSM  Counter     START    1.000000000E+01
(84) THE SYSTEM HAS BEEN STOPPED
(85) Do you want to save the current simulated system status?
(86) Y=Yes     N=No    ?N
(87) DONE
     $ lo
```

Figure 4.4. (continued)

In (10), we invoke the SAS command procedure to execute the simulated system again. This time the SAS goes through the Data Input process. Lines (11), (12) and (13) indicate that the SAN specification of the component Terminal, Pulsegen and Counter are processed. In (14), the SAS asks the user if the initialization process ia a New Initialization. We type in 'Y' to indicate yes. In (15), the SAS indicates that the system is stable. The SAS gives control to the System Executive to execute the simulated system.

We initialized the simulation beginning time, tbeg to 1.0 and the start expression check option of the ENV component Terminal is everytimechange; thus just before tnow=1.0 is advanced the start event of Terminal is scheduled as shown in (16). In (17), we know that the SAS is executing the start event of Terminal and in (18), SAS prompts a '#' sign to indicate the SAS is in the terminal command mode. In (18), we type in the command to print the status of all the system components. The SAS in response prints the status of all the system components at our terminal. In (19), we type in the command to save the status of all the system components in the file, sysfile. In (20), we assign a pulsed output to Terminal. Unfortunately the pulse value, 'strt', is not an element of the pulsed output set. In (21), we type in the new pulse value 'start'; this time it is accepted by the SAS. In (22), we type in the command to exit from the terminal mode.

The 'start' pulse from the Terminal pulsed output is loaded into the ENV component Pulsegen. A start event of Pulsegen is scheduled in the event file as shown in (23). In (24), the start event of Pulsegen is

executed. Since Pulsegen has an execution time equal 1.0, an update event of Pulsegen is scheduled in the event file with event time equal 2.0 as shown in (25). At the end of execution of all the events in the current event set with event time equal 1.0, a start event of Terminal is scheduled into the event file as shown in (26). In (27), the start event of Terminal is executed to allow the user to examine the system status before the simulation time is advanced. In (28), we type the command to print the event file. The SAS prints the event file on the terminal indicating that there is an update event of Pulsegen to be executed at tnow=2.0. In (29), we exit from the terminal mode.

In (30), the SAS executes the update event of Pulsegen, which generates a pulse at its pulsed output and loads the pulsed output into its destination component, Counter. A start event of Counter is scheduled in the event file as shown in (31). In (32), the start event of Counter is executed. Since the execution time of Counter is zero, the update event of Counter is executed immediately following the start event of Counter. That is why there is no update event scheduled at the end of the execution of the start event of Counter. The component Terminal also has an execution time of zero, which implies no update event needs to be scheduled at the end of the execution of the start event of Terminal. In (33), a start event of Terminal is scheduled after all the events in the current event set are executed. In (34), the SAS executes the Terminal start event. In (35), we type a wrong command. In (36), we type the command to print the status of all the system component again. Notice that the current state of Counter has changed from '0' to '1' as

shown in (37), which indicates that a 'inc' pulse has been loaded into Counter from Pulsegen. In (38), we assign a new 'start' pulse to the Terminal pulsed output. In (39), we type in the command to exit from the terminal mode.

In (40), (41), (42), (43) and (44), the SAS repeats the same process in (23), (24), (25), (26) and (27) to execute the Pulsegen. In (45), we type the command to print the status of Pulsegen. Notice that a new 'inc' pulse is generated at the Pulsegen pulsed output. In (46), we type in the command to exit from the terminal mode. In (47), (48), (49), (50) and (51), the SAS repeats the same process in (30), (31), (32), (33) and (34) to load the pulse from Pulsegen into Counter and execute the start and update event of Counter. In (52), we type the command to print the status of Counter. Notice that the current simulation time is 3.0, and we had sent two 'start' pulses into Pulsegen which in turn had sent two 'inc' pulses into Counter. The current state of Counter is '2' as shown in (53).

From (54) to (69), we repeat the steps of sending 'start' pulses from the Terminal pulsed output, which are similar to the previous steps. In (70), the SAS is executing the Terminal start event at tnow=6.0. In (71), we type in the wrong component name. In (72), we type the command to print the status of Pulsegen. In (73), we type the command to change the start expression check option of Terminal to never, which means the start expression of Terminal will never be evaluated. In (74), we type the command to indicate a start event of

Terminal needs to be scheduled at tnow=9.0. In (75), we type the command to exit from the terminal mode.

Since the start expression check option of Terminal had been changed to never, the start event of Terminal will not be scheduled at the end of the execution of all the events in an event set. From (76) to (80), the SAS executes the simulated system until tnow=9.0 and schedules a start event of Terminal. In (81), we type the wrong command again. In (82), we assign a new 'start' pulse to the Terminal pulsed output. In (83), we type the command to exit from the terminal mode.

Since we initialize the simulation ending time, tend, equal 10.0, the SAS executes the simulated system until all the events in the event set with event time equal 10.0 and stops. SAS prints out a message 'THE SYSTEM HAS BEEN STOPPED' as shown in (84). In (85) and (86), SAS asks the user if the current simulated system status need to be saved. We type in 'N' to indicate no. SAS then prints the message 'DONE' to indicate the end of SAS execution.

The SAN model in the file simple.dat contains three performance trace instances; these were saved in the file datafile.dat, as listed in Figure 4.5. The first trace is the regular variable history instance, TraceCount. It traces the static output of Counter, which shows that the Counter counts up to five 'inc' pulses at time equal 5.0 and the static output stays at '5' until time equal 10.0; the static output is set to '0'. The second one is the conditional variable history instance, TracePulse. It traces the occurrence of the 'reset' pulse at the Pulsegen pulsed output. We notice the a 'reset' pulse occurs at

```
VARHIST  TraceCount                    :  regular
Counter.czs.1                           Time
0                                       1.000000000E+00
1                                       2.000000000E+00
2                                       3.000000000E+00
3                                       4.000000000E+00
4                                       5.000000000E+00
5                                       6.000000000E+00
5                                       7.000000000E+00
5                                       8.000000000E+00
5                                       9.000000000E+00
0                                       1.000000000E+01

VARHIST  TracePulse                    :  conditional
Pulsegen.czp.1                          Time
reset                                   1.000000000E+01

EXPHIST  Tracestatus                   :  regular
 FALSE                                  1.000000000E+00
  TRUE                                  2.000000000E+00
 FALSE                                  3.000000000E+00
 FALSE                                  4.000000000E+00
 FALSE                                  5.000000000E+00
 FALSE                                  6.000000000E+00
 FALSE                                  7.000000000E+00
 FALSE                                  8.000000000E+00
 FALSE                                  9.000000000E+00
 FALSE                                  1.000000000E+01
```

Figure 4.5. A listing of the datafile.dat file

time=10.0, which in turn sets the static output of Counter to '0'. The third one is the regular expression history instance, Tracestatus. The expression is true only at time=2.0 otherwise it is false as shown in Figure 4.5.

During the simulated system execution, we typed in the command to save the status of all the system components at simulation time equal 1.0. The status of all the system components were saved in the file, sysfile.dat as shown in Figure 4.6.

During the simulated system execution, the variables and expression traces are each temporarily stored in a linked list as described in the last chapter. The values in the linked list will be stored in the datafile.dat file on the disk at the end of the system execution. If the computer system breaks down or the user types in the character 'CNTL-C' to interrupt the current program execution during the simulation execution time, the simulation execution will be stopped immediately. If this happens, all the variable and expression traces of the current simulation run will be lost.

As for the system component status, once we type a command in the Terminal mode to save the system component status, the component statuses are stored into the sysfile.dat in the disk immediately. Unless the disk information is lost, the system component status can always be obtained from the sysfile.dat file.

```
SYSTEM STATUS AT TNOW := 1.000000000E+00

FSM     Counter
CS:= 0
cxp. 1:= -
czs. 1:= 0
Execution Delay:=  0.000000000E+00
Execstatus:=  IDLE

ENV     Terminal
czp. 1:= -
Execution time:=  0.000000000E+00
Execstatus:=  BUSY
STARTEXPCHECK:= EVERYTIMECHANGE
updatetime:= 1.000000000E+00
ntzp. 1:=-

ENV     Pulsegen
cxp. 1:= -
czp. 1:= -
Execution time:=  1.000000000E+00
Execstatus:=  IDLE
STARTEXPCHECK:=           NEVER
```

Figure 4.6. A listing of the sysfile.dat file

4. Execution of the SAN model executable image with execution errors

This sub-section illustrates the response of SAS to system execution errors. The same SAN model as in the previous sub-section is used. This time we will try to send a 'start' pulse to the component Pulsegen while Pulsegen is busy. Figure 4.7 presents the listing of an example run. In line (1), we invoke the SAS command procedure with the RUN step to execute the SAN model executable image. SAS asks the user if the initialization process is a New Initialization. In (2), we type in 'Y' to indicate yes. The system is stable. In the beginning, SAS schedules a start event of Terminal, which has a STARTEXPCHECK equal everytimechange. In (3), SAS indicates to the user that a start event of Terminal is being executed. In (4), we assign a 'start' pulse at the first pulsed output of Terminal, which is connected to the first pulsed input of Pulsgen. In (5), we type 'e' to exit from the terminal mode. The start pulse is then loaded into the pulsed input of Pulsegen at tnow=1.0 and a start event of Pulsegen is scheduled at tnow=1.0 (6). In (7), the start event of Pulsegen is executed and an update event of Pulsegen is scheduled at tnow=2.0 (8). At the end of execution of all the events in the event set with event set time=1.0, a start event of Terminal is scheduled (9). In (10), the start event of Terminal is executed and we type in the Terminal Mode Command Language to print out the current event file content (11). In (12), we type 'e' to exit from the terminal mode. SAS then advances the simulation time to 2.0 and executes the update event of Pulsegen, which generates a pulsed output. The pulsed output is loaded into its destination component, Counter, and a start

```
 (1) $ @sas counter.dat run
       PREVIOUS LOGICAL NAME ASSIGNMENT REPLACED
     tempsm=Terminal
     tempsm=Pulsegen
     tempsm=Counter
 (2) New Initialization ? Y=Yes or N=No. Y
     SYSTEM INITIALIZATION : CHECKING FOR UNSTABLE MACHINE
      **  SYSTEM IS STABLE **
     SYSTEM HAS BEEN STARTED
     ---- SCHEDULE ----  ENV  Terminal    START   1.000000000E+00
     EXECUTE :  ENV  Terminal    START   1.000000000E+00

 (3) ENV : Terminal    TNOW:=  1.000000000E+00
 (4) #a ZP.1:=start
     zp or zs number:=          1
 (5) #e
 (6) ---- SCHEDULE ----  ENV  Pulsegen    START   1.000000000E+00
 (7) EXECUTE :  ENV  Pulsegen    START   1.000000000E+00
 (8) ---- SCHEDULE ----  ENV  Pulsegen   UPDATE   2.000000000E+00
 (9) ---- SCHEDULE ----  ENV  Terminal    START   1.000000000E+00
(10) EXECUTE :  ENV  Terminal    START   1.000000000E+00

     ENV : Terminal    TNOW:=  1.000000000E+00
(11) #p eventfile
     Eventset.time:= 1.000000000E+00     Eventset.count:=          0

     Eventset.time:= 2.000000000E+00     Eventset.count:=          1
       UPDATE   ENV    Pulsegen
(12) #e
     EXECUTE :  ENV  Pulsegen   UPDATE   2.000000000E+00
(13) ---- SCHEDULE ----  FSM  Counter    START   2.000000000E+00
(14) EXECUTE :  FSM  Counter    START   2.000000000E+00
(15) ---- SCHEDULE ----  ENV  Terminal    START   2.000000000E+00
     EXECUTE :  ENV  Terminal    START   2.000000000E+00

     ENV : Terminal    TNOW:=  2.000000000E+00
(16) #a ZP.1:=start
     zp or zs number:=          1
     #e
(17) ---- SCHEDULE ----  ENV  Pulsegen    START   2.000000000E+00
(18) EXECUTE :  ENV  Pulsegen    START   2.000000000E+00
(19) ---- SCHEDULE ----  ENV  Pulsegen   UPDATE   3.000000000E+00
(20) ---- SCHEDULE ----  ENV  Terminal    START   2.000000000E+00
(21) EXECUTE :  ENV  Terminal    START   2.000000000E+00

     ENV : Terminal    TNOW:=  2.000000000E+00
(22) #a ZP.1:=start
     zp or zs number:=          1
```

Figure 4.7. Listing of an example run with system execution error

```
    *e
(23) hello execerr
(24) EXECUTION ERROR NUMBER :=       1015
(25) Messase : the destination component is in pend or busy state
(26) CURRENT SIMULATION TIME :  2.000000000E+00
(27) CURRENT EVENT :    START
(28) CURRENT MACHINE : Terminal
(29) DESTINATION MACHINE : Pulsesen
(30) Pulsed input Number :           1
    SYSTEM EXECUTION ERROR, CONTROL IS GIVEN TO UPM

    ENV : S$SYSTEM_MONITOR    TNOW:=  2.000000000E+00
    *p Terminal
    ENV      Terminal
    czp. 1:= start
    Execution time:=  0.000000000E+00
    Execstatus:=  IDLE
    STARTEXPCHECK:= EVERYTIMECHANGE
(31) *p Pulsesen
    ENV      Pulsesen
    cxp. 1:= start
    czp. 1:= inc
    Execution time:=  1.000000000E+00
(32) Execstatus:=  BUSY
    STARTEXPCHECK:=           NEVER
    updatetime:= 3.000000000E+00
    ntzp. 1:=inc
(33) *h
    ---- SCHEDULE ----  ENV  Pulsesen    START   2.000000000E+00
    SYSTEM HALT IS TRUE
    THE SYSTEM HAS BEEN STOPPED
(34) Do you want to save the current simulated system status?
(35) Y=Yes     N=No    ?N
(36) DONE
(37) $
```

event of Counter is scheduled (13). In (14), the start event of Counter is executed. At the end of execution of all the events in the current event set, the start event of Terminal is scheduled (15). When the start event of Terminal is executed, a 'start' pulse is assigned to the pulsed output of Terminal at tnow=2.0 (16). The pulsed output is loaded into the pulsed input of Pulsegen. From (17) to (20), SAS repeats the same process in (6) to (9) to execute the system components at tnow=2.0. At the end of execution of all events in the current event set, the start event of Terminal is also scheduled and executed (20) and (21). In (22), we assign another start pulse to the pulsed output of Terminal. After we type 'e' to exit from the terminal mode, the pulsed output is loaded into its destination component. At this point, SAS detects an execution error. SAS prints out the error message on the terminal as shown in (23) to (30). From the error message we notice that Terminal pulses out a start pulse to the first pulsed input of Pulsegen, while Pulsegen is busy. SAS then gives the control to the System-Monitor to allow the user to examine the current system status. In (31), we type in the command to point out the status of Pulsegen. The execution status of Pulsegen is busy (32). In (33), we type 'h' to halt the simulated system execution. In (34), SAS asks the user if he wants to save the current simulated system status. In (35), we type 'N' to indicate no. In (36), SAS prints out 'DONE' to indicate the end of SAS execution and returns to the VAX command mode indicated by the '$' (37).

5. Demonstration of the Restart Initialization process

The Restart Initialization process allows users to continue the execution of a SAN model simulation from the state where the simulation last stopped. This sub-section illustrates the Restart Initialization process using the simple SAN model in the previous sub-section. Figure 4.8 presents the listing of an example run. This example demonstrates that after having saved the total system status at the end of a simulation run, the simulation can be re-initialized to the same state where the simulation last stopped and continued.

In line (1), we invoke the SAS command procedure with the RUN step to execute the SAN model executable image. SAS then asks the user if the initialization process is a New Initialization. In (2), we type in 'Y' to indicate yes, the system is stable. In the beginning SAS schedules a start event of Terminal (which has a STARTEXPCHECK equal everytimechange). The steps from line (1) to line (21) are similar to those in the previous example (Figure 4.7). Readers may refer to the previous sub-section to understand the above steps.

At the end of execution of all the events in the event set with event set time = 2.0, a start event of Terminal is scheduled (20). In (21), the start event of Terminal is executed and we type in the command to change the check option of the start expression of Terminal to everyevent (22). This is done so that we can examine the system status after the execution of each event. In (23), we type 'e' to exit from the terminal mode. The simulator advances to tnow=3.0 and executes the update event of Pulsegen, which in turn schedules a start event of

```
(1) $ @sas simple.dat run
    Previous logical name assignment replaced
    tempsm=Terminal
    tempsm=Pulsegen
    tempsm=Counter
(2) New Initialization ? Y=Yes or N=No. Y
    SYSTEM INITIALIZATION : CHECKING FOR UNSTABLE MACHINE
     **  SYSTEM IS STABLE **
    SYSTEM HAS BEEN STARTED
    ---- SCHEDULE ----  ENV  Terminal    START   1.000000000E+00
    EXECUTE :  ENV  Terminal    START   1.000000000E+00

(3) ENV : Terminal    TNOW:=  1.000000000E+00
(4) #a ZP.1:=start
    zp or zs number:=          1
(5) #e
(6) ---- SCHEDULE ----  ENV  Pulsegen    START   1.000000000E+00
(7) EXECUTE :  ENV  Pulsegen    START   1.000000000E+00
(8) ---- SCHEDULE ----  ENV  Pulsegen   UPDATE   2.000000000E+00
(9) ---- SCHEDULE ----  ENV  Terminal    START   1.000000000E+00
(10) EXECUTE :  ENV  Terminal    START   1.000000000E+00

(11) ENV : Terminal    TNOW:=  1.000000000E+00
(12) #e
    EXECUTE :  ENV  Pulsegen   UPDATE   2.000000000E+00
(13) ---- SCHEDULE ----  FSM  Counter    START   2.000000000E+00
(14) EXECUTE :  FSM  Counter    START   2.000000000E+00
(15) ---- SCHEDULE ----  ENV  Terminal    START   2.000000000E+00
    EXECUTE :  ENV  Terminal    START   2.000000000E+00

    ENV : Terminal    TNOW:=  2.000000000E+00
(16) #a ZP.1:=start
    zp or zs number:=          1
(17) #e
    ---- SCHEDULE ----  ENV  Pulsegen    START   2.000000000E+00
(18) EXECUTE :  ENV  Pulsegen    START   2.000000000E+00
(19) ---- SCHEDULE ----  ENV  Pulsegen   UPDATE   3.000000000E+00
(20) ---- SCHEDULE ----  ENV  Terminal    START   2.000000000E+00
(21) EXECUTE :  ENV  Terminal    START   2.000000000E+00

(22) ENV : Terminal    TNOW:=  2.000000000E+00
    #a STARTCHECK:=everyevent
    CHECKOPT:=       EVERYEVENT
(23) #e
(24) EXECUTE :  ENV  Pulsegen   UPDATE   3.000000000E+00
(25) ---- SCHEDULE ----  FSM  Counter    START   3.000000000E+00
(26) ---- SCHEDULE ----  ENV  Terminal    START   3.000000000E+00
(27) EXECUTE :  ENV  Terminal    START   3.000000000E+00
```

Figure 4.8. Example run to demonstrate the Restart Initialization process

```
    ENV : Terminal     TNOW:=  3.000000000E+00
(28) #p all
    SYSTEM STATUS AT TNOW := 3.000000000E+00

    FSM      Counter
(29) CS:= 1
(30) cxp. 1:= inc
    czs. 1:= 1
    Execution Delay:=  0.000000000E+00
(31) Execstatus:=  PEND

    ENV      Terminal
    czp. 1:= -
    Execution time:=  0.000000000E+00
(32) Execstatus:=  BUSY
    STARTEXPCHECK:=       EVERYEVENT
    updatetime:= 3.000000000E+00
(33) ntzp. 1:=-

    ENV      Pulsegen
    cxp. 1:= -
    czp. 1:= inc
    Execution time:=  1.000000000E+00
(34) Execstatus:=  IDLE
    STARTEXPCHECK:=            NEVER
(35) #p eventfile
    Eventset.time:= 3.000000000E+00      Eventset.count:=          1
(36)   START    FSM     Counter
(37) #f 3.0
(38) #h
    SYSTEM HALT IS TRUE
    THE SYSTEM HAS BEEN STOPPED
    Do you want to save the current simulated system status?
(39) Y=Yes     N=No    ?Y
(40) DONE
(41) $ @sas simple.dat run
    Previous logical name assignment replaced
    Previous logical name assignment replaced
    tempsm=Terminal
    tempsm=Pulsegen
    tempsm=Counter
(42) New Initialization ? Y=Yes or N=No. N
    ---- SCHEDULE ----  FSM  Counter     START   3.000000000E+00
    SYSTEM HAS BEEN STARTED
(43) ---- SCHEDULE ----  ENV  Terminal     START   3.000000000E+00
(44) EXECUTE :  ENV  Terminal     START   3.000000000E+00

    ENV : Terminal     TNOW:=  3.000000000E+00
(45) #p all
    SYSTEM STATUS AT TNOW := 3.000000000E+00
```

Figure 4.8. (continued)

```
     FSM      Counter
     CS:= 1
     cxp. 1:= inc
     czs. 1:= 1
     Execution Delay:=  0.000000000E+00
     Execstatus:=  PEND

     ENV      Terminal
     czp. 1:= -
     Execution time:=  0.000000000E+00
     Execstatus:=  BUSY
     STARTEXPCHECK:=       EVERYEVENT
     updatetime:= 3.000000000E+00
     ntzp. 1:=-

     ENV      Pulsegen
     cxp. 1:= -
     czp. 1:= inc
     Execution time:=  1.000000000E+00
     Execstatus:=  IDLE
     STARTEXPCHECK:=            NEVER
(46) #p eventfile
     Eventset.time:= 3.000000000E+00      Eventset.count:=          1
        START    FSM     Counter
(47) #p
(48) EXECUTE :  FSM  Counter     START    3.000000000E+00
(49) ---- SCHEDULE ----  ENV  Terminal     START    3.000000000E+00
(50) EXECUTE :  ENV  Terminal     START    3.000000000E+00

     ENV : Terminal     TNOW:=  3.000000000E+00
(51) #p all
     SYSTEM STATUS AT TNOW := 3.000000000E+00

     FSM      Counter
     CS:= 2
     cxp. 1:= -
     czs. 1:= 2
     Execution Delay:=  0.000000000E+00
     Execstatus:=  IDLE

     ENV      Terminal
     czp. 1:= -
     Execution time:=  0.000000000E+00
     Execstatus:=  BUSY
     STARTEXPCHECK:=       EVERYEVENT
     updatetime:= 3.000000000E+00
     ntzp. 1:=-

     ENV      Pulsegen
     cxp. 1:= -
```

Figure 4.8. (continued)

```
    czp. 1:= inc
    Execution time:=  1.000000000E+00
    Execstatus:=  IDLE
    STARTEXPCHECK:=          NEVER
    #p eventfile
    Eventset.time:= 3.000000000E+00     Eventset.count:=          0
(52) #h
    SYSTEM HALT IS TRUE
    THE SYSTEM HAS BEEN STOPPED
    Do you want to save the current simulated system status?
    Y=Yes    N=No    ?N
    DONE
```

Figure 4.8. (continued)

Counter (25). In (26), the simulator schedules a start event of Terminal and executes the start event (27). In (28), we type in the command to print the status of all the components. We see that the current state of Counter is '1' (29), and an 'inc' pulse arrives at Counter's pulsed input (30), and Counter is Pending (31). Terminal is Busy (32) and the update record shows that no new pulsed output value is assigned to Terminal (33). Pulsegen is Idle (34). In (35), we type in the command to print the event file. The event file shows that a start event of Counter is pending to be executed at tnow = 3.0 (36). In (37), we type the command to schedule a start event of Terminal at tnow = 3.0 just before the system halt command is typed (38). This is done so that when the simulated system is re-initialized SAS will first execute the start event of Terminal and allow users to examine the system status before the simulation run continues. In (39), SAS asks if the current simulated system status is to be saved. We type in 'Y' to indicate yes (40). The current simulated system status is saved in the sysfile.dat file. The current simulation run is stopped. In (41), we invoke the SAS command procedure again. SAS asks the user if the initialization process is a New Initialization. In (42), we type in 'N' to indicate that it is not a New Initialization (which implies Restart Initialization). The Restart Initialization process is invoked to reload the previous system status from the sysfile.dat file. Since a start event of Terminal was scheduled at tnow = 3.0 in the previous run, a start event of Terminal is scheduled and executed at tnow = 3.0 (44). In (45), we type the command to print the simulated system status. The system status is exactly the same as

the system status when the simulation run last stopped. In (46), we type the command to print the event file. The event file is exactly the event file which the system had when the last simulation run was terminated. In (47), we type 'e' to exit from the terminal mode. The system execution is then continued.

The start event of Counter is executed (48). At the end of the execution of the start event of Counter, a start event of Terminal is scheduled (49) and executed (50). In (51), we examine the system status to check if the system execution proceeds properly or not. The current state of Counter changes from '1' to '2' due to the arrival of a second 'inc' pulse. The execution status of both Counter and Pulsegen are Idle, and that of Terminal is Busy. The system status shows that the simulated system continues to execute properly after the Restart Initialization process. In (52), we stop the system execution.

C. Start-Stop Link Simulation

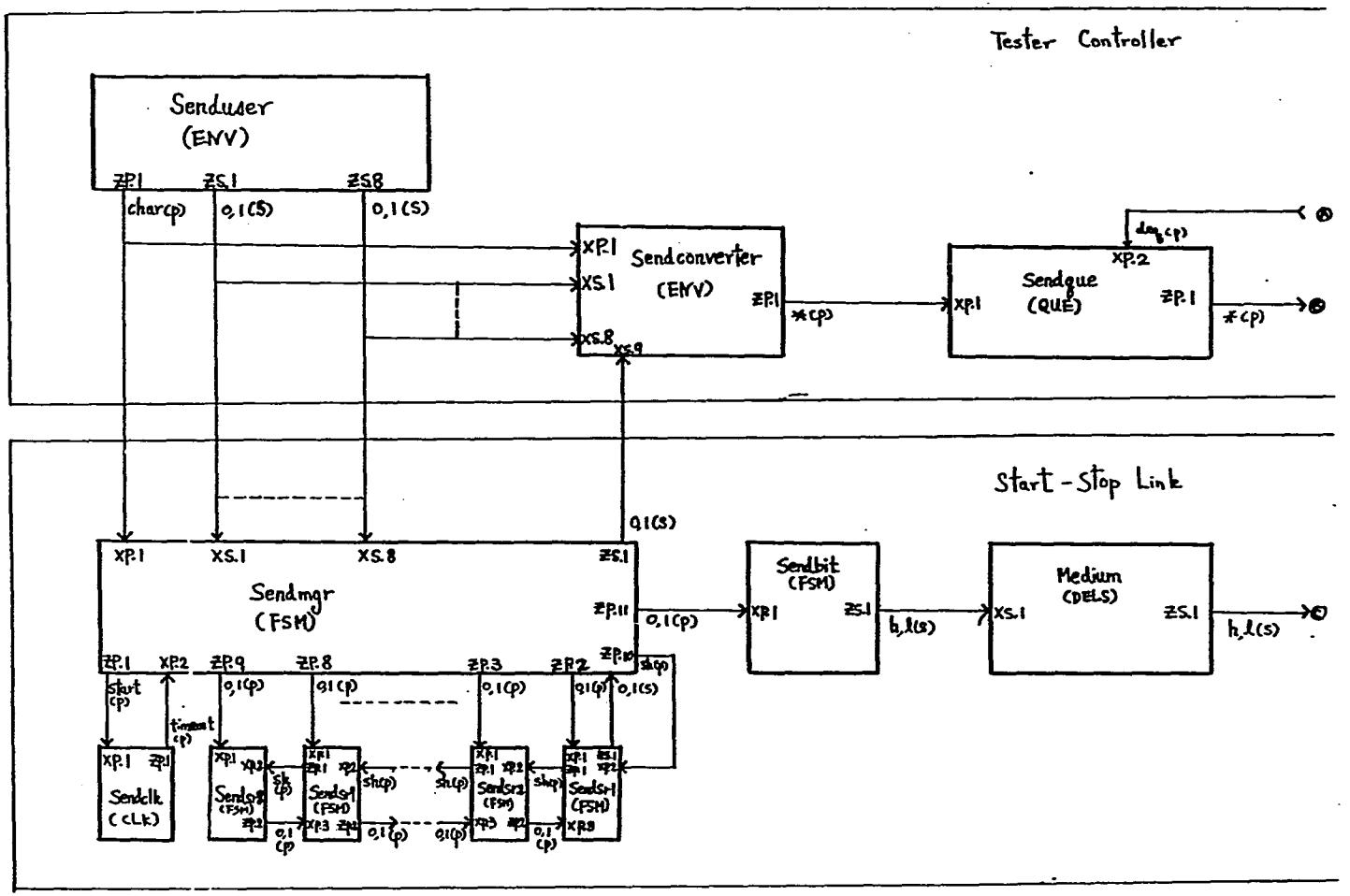
This section illustrates the use of six of the nine kinds of components (all except CFP, CFS, and DELS) of State Architecture Notation in simulating a Start-Stop link [Piatkowski 1981]. It also demonstrates the use of procedure and list options in representing the FNS, FOUTP, and FOUTS functions, the use of terminal and procedure options in the ENV FUNCTION specification, the use of different kinds of variable history trace specifications, and how to create a model subsystem to test a protocol. The Start-Stop link simulation consists

of the Start-Stop model as suggested by [Piatkowski 1981] and a test controller subsystem interconnected with the Start-Stop model to provide the test bed for the start-stop protocol. A block diagram of the Start-Stop link simulation is shown in Figure 4.9

1. Start-Stop model

The Start-Stop link without the test controller is a data communication link supporting a simplex (unidirectional) connection between a sender and a receiver in which a data stream of eight-bit blocks (bytes) is transmitted via the well-known start-stop protocol. The sender of the start-stop protocol basically takes the eight-bit byte and transmits it bit-serial by adding a start bit at the beginning of the transmission and two stop bits at the end. The receiver of the start-stop protocol strips off the start and two stop bits, assembles the eight-bit block into a byte and presents the byte to the next higher level.

Figure 4.10 presents the SAN model of the Start-Stop link simulation. The sender of the Start-Stop link is built-up from eleven basic components: Sendmgr (FSM), Sendclk (CLK), Sendbit (FSM), and an eight bit shift register built up of eight one-bit shift registers, Sendsr1, Sendsr2, ----, Sendsr8 (FSM). The sender connects to the receiver via a static delay (DELS) named Medium. The receiver is built-up from twelve basic components: Rcvdet (DER), Rcvmar (FSM), Rcvclk1 (CLK), Rcvclk2 (CLK), and an eight bit shift register built up of eight ripple connected one-bit shift registers, Rcvsr1, Rcvsr2, ----, Rcvsr8


Tester Controller
Senduser
(ENV)
char(p)
0,1(s)
0,1(s)
Sendconverter
(ENV)
Sendque
(QUE)
Start-Stop Link
Sendmgr
(FSM)
Sendbit
(FSM)
Medium
(DELS)
Sendclk
(CLK)

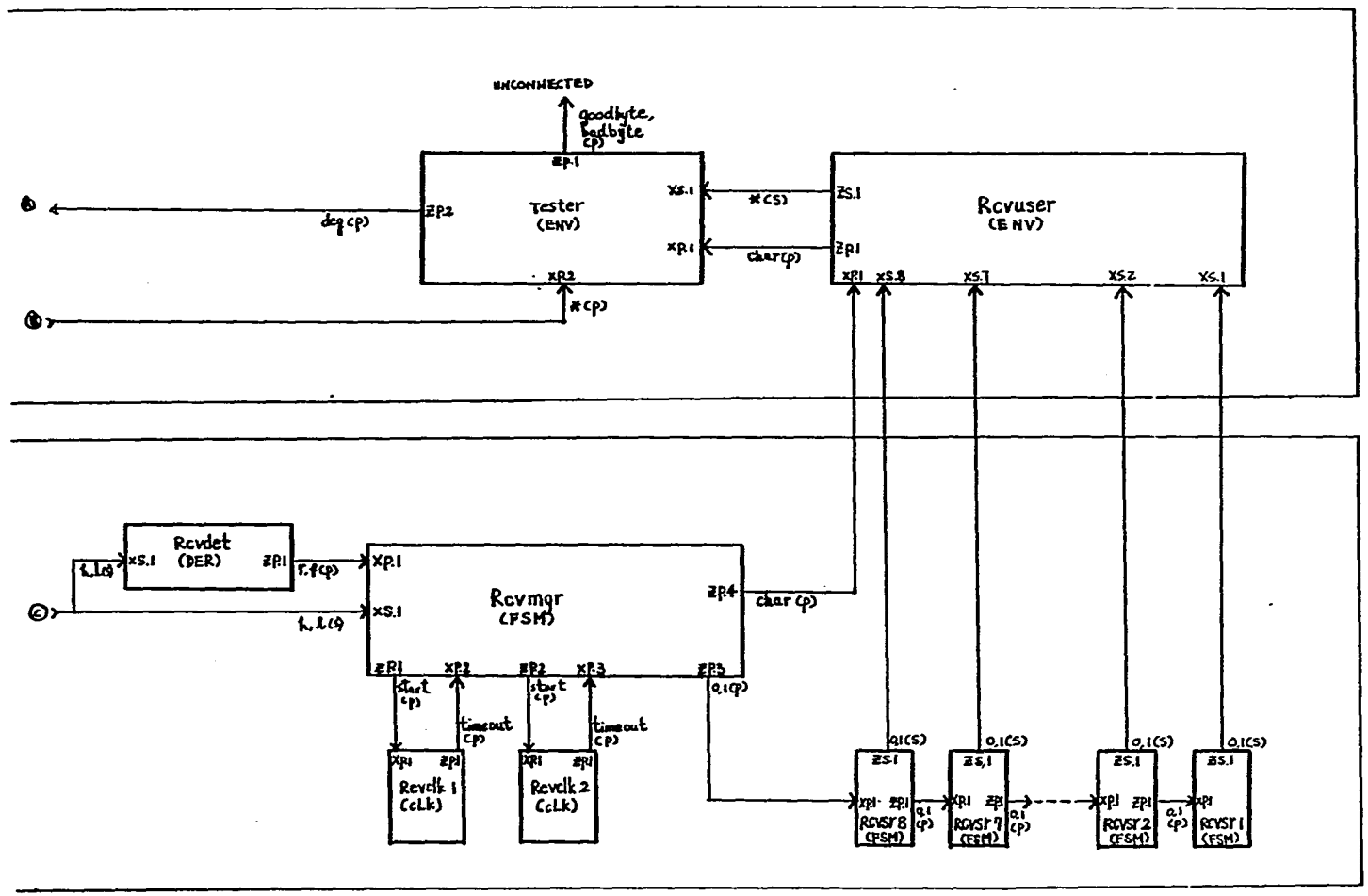


Figure 4.9. Block diagram of the Start-Stop link and Test Controller

```
{Start-Stop Link Model          July 20, 1982}

ENV Senduser : interlaced
  ZP.1 TO Sendmgr.XP.1,Sendconverter.XP.1 : char;
  ZS.1 TO Sendmgr.XS.1,Sendconverter.XS.1 : 0,1;
  ZS.2 TO Sendmgr.XS.2,Sendconverter.XS.2 : 0,1;
  ZS.3 TO Sendmgr.XS.3,Sendconverter.XS.3 : 0,1;
  ZS.4 TO Sendmgr.XS.4,Sendconverter.XS.4 : 0,1;
  ZS.5 TO Sendmgr.XS.5,Sendconverter.XS.5 : 0,1;
  ZS.6 TO Sendmgr.XS.6,Sendconverter.XS.6 : 0,1;
  ZS.7 TO Sendmgr.XS.7,Sendconverter.XS.7 : 0,1;
  ZS.8 TO Sendmgr.XS.8,Sendconverter.XS.8 : 0,1;

  FUNCTION: terminal;
  defzsinit : 0,1,0,1,0,1,0,1;
  deftexec: 0.0;
  STARTEXP : true;
  STARTEXPCHECK : everytimechange;

END;

ENV Sendconverter : interlaced
  XP.1 FROM Senduser.ZP.1 : char;
  XS.1 FROM Senduser.ZS.1 : 0,1;
  XS.2 FROM Senduser.ZS.2 : 0,1;
  XS.3 FROM Senduser.ZS.3 : 0,1;
  XS.4 FROM Senduser.ZS.4 : 0,1;
  XS.5 FROM Senduser.ZS.5 : 0,1;
  XS.6 FROM Senduser.ZS.6 : 0,1;
  XS.7 FROM Senduser.ZS.7 : 0,1;
  XS.8 FROM Senduser.ZS.8 : 0,1;
  XS.9 FROM Sendmgr.ZS.1 : Idle,Busy;
  ZP.1 TO Sendque.XP.1 : *;

  FUNCTION : procedure

    Procedure Sendconverterproc;
    begin
        with env[U$Sendconverter],updatestate do
        begin
            if(cxp[1]='char                                    ')and
              (cxs[9]='idle                                    ')then
            begin
                ntzp[1][1]:=cxs[1][1];
                ntzp[1][2]:=cxs[2][1];
                ntzp[1][3]:=cxs[3][1];
                ntzp[1][4]:=cxs[4][1];
                ntzp[1][5]:=cxs[5][1];
                ntzp[1][6]:=cxs[6][1];
                ntzp[1][7]:=cxs[7][1];
                ntzp[1][8]:=cxs[8][1];
            end
            else
                ntzp[1]:=null;
        end;
    end; {end of Sendconverterproc}
  END;

  deftexec:0.0;
  mulpulsecheck : false;
  STARTEXP:true;
  STARTEXPCHECK:never;
END;
```

Figure 4.10. The Start-Stop link SAN model

```
ENV Rcvuser : interlaced
  XP.1 FROM Rcvmgr.ZP.4 : char;
  XS.1 FROM Rcvsr1.ZS.1 : 0,1;
  XS.2 FROM Rcvsr2.ZS.2 : 0,1;
  XS.3 FROM Rcvsr3.ZS.3 : 0,1;
  XS.4 FROM Rcvsr4.ZS.4 : 0,1;
  XS.5 FROM Rcvsr5.ZS.5 : 0,1;
  XS.6 FROM Rcvsr6.ZS.6 : 0,1;
  XS.7 FROM Rcvsr7.ZS.7 : 0,1;
  XS.8 FROM Rcvsr8.ZS.8 : 0,1;
  ZP.1 TO Tester.XP.1 : char;
  ZS.1 TO Tester.XS.1 : #;

  FUNCTION : procedure

    Procedure Rcvuserproc;
    begin
        with env[U$Rcvuser],updatestate do
        begin
            if cxp[1]='char                                          ' then
            begin
                ntzp[1]:='char                                        ';
                ntzs[1][1]:=cxs[1][1];
                ntzs[1][2]:=cxs[2][1];
                ntzs[1][3]:=cxs[3][1];
                ntzs[1][4]:=cxs[4][1];
                ntzs[1][5]:=cxs[5][1];
                ntzs[1][6]:=cxs[6][1];
                ntzs[1][7]:=cxs[7][1];
                ntzs[1][8]:=cxs[8][1];
            end;
        end;
    end; {end of Rcvuserproc}
  END;

  defzsinit:00000000;
  deftexec:0.0;
  mulpulsecheck : false;
  STARTEXP:true;
  STARTEXPCHECK:never;
END;
```

Figure 4.10. (continued)

```
ENV Tester : interlaced
  XP.1 FROM Rcvuser.ZP.1 : char;
  XP.2 FROM Sendque.ZP.1 : *;
  XS.1 FROM Rcvuser.ZS.1 : *;
  ZP.1 UNCONNECTED : Goodbyte,Badbyte;
  ZP.2 TO Sendque.XP.2 : deq;

  FUNCTION : procedure

    Procedure Testerproc;
    var
        a : array[1..4] of string;

    begin
        a[1]:='Goodbyte                                          ';
        a[2]:='Badbyte                                           ';
        a[3]:='char                                              ';
        a[4]:='deq                                               ';
        with env[U$Tester],updatestate do
        begin
            if cxp[1]=a[3] then ntzp[2]:=a[4]
            else
            if cxp[2]<>null then
            begin
                if cxp[2]=cxs[1] then ntzp[1]:=a[1]
                                 else ntzp[1]:=a[2];
            end;
        end; {end with}
    end; {end Testerproc}

  END;

  deftexec:0.0;
  STARTEXP : true;
  STARTEXPCHECK : never;

END;

QUE Sendque : interlaced
  XP.1 FROM Sendconverter.ZP.1 : *;
  XP.2 FROM Tester.ZP.2: deq;
  ZP.1 TO Tester.XP.2 : *;
  ZS.1 UNCONNECTED : open,closedempty,closednotempty;
  deftenq : 0.0;
  deftdeq : 0.0;
END;
```

Figure 4.10. (continued)

```
FSM Sendmgr : interlaced
  S:Idle,Start,1,2,3,4,5,6,7,8,Stop1,Stop2;
  defsinit:Idle;
  XP.1 FROM Senduser.ZP.1 : char;
  XP.2 FROM Sendclk.ZP.1 : timeout;
  XS.1 FROM Senduser.ZS.1 :0,1;
  XS.2 FROM Senduser.ZS.2 :0,1;
  XS.3 FROM Senduser.ZS.3 :0,1;
  XS.4 FROM Senduser.ZS.4 :0,1;
  XS.5 FROM Senduser.ZS.5 :0,1;
  XS.6 FROM Senduser.ZS.6 :0,1;
  XS.7 FROM Senduser.ZS.7 :0,1;
  XS.8 FROM Senduser.ZS.8 :0,1;
  XS.9 FROM Sendsr1.ZS.1 : 0,1;
  ZP.1 TO Sendclk.XP.1 : start;
  ZP.2 TO Sendsr1.XP.1 : 0,1;
  ZP.3 TO Sendsr2.XP.1 : 0,1;
  ZP.4 TO Sendsr3.XP.1 : 0,1;
  ZP.5 TO Sendsr4.XP.1 : 0,1;
  ZP.6 TO Sendsr5.XP.1 : 0,1;
  ZP.7 TO Sendsr6.XP.1 : 0,1;
  ZP.8 TO Sendsr7.XP.1 : 0,1;
  ZP.9 TO Sendsr8.XP.1 : 0,1;
  ZP.10 TO Sendsr1.XP.2: sh;
  ZP.11 TO Sendbit.XP.1 :0,1;
  ZS.1 TO Sendconverter.XS.9 : Idle,Busy;

  FNS : list
    Idle/char,-/*,*,*,*,*,*,*,*,*=>Start;
    Start/-,timeout/*,*,*,*,*,*,*,*,*=>1;
    Start/char,-/*,*,*,*,*,*,*,*,*=>Start;
    1/-,timeout/*,*,*,*,*,*,*,*,*=>2;
    1/char,-/*,*,*,*,*,*,*,*,*=>1;
    2/-,timeout/*,*,*,*,*,*,*,*,*=>3;
    2/char,-/*,*,*,*,*,*,*,*,*=>2;
    3/-,timeout/*,*,*,*,*,*,*,*,*=>4;
    3/char,-/*,*,*,*,*,*,*,*,*=>3;
    4/-,timeout/*,*,*,*,*,*,*,*,*=>5;
    4/char,-/*,*,*,*,*,*,*,*,*=>4;
    5/-,timeout/*,*,*,*,*,*,*,*,*=>6;
    5/char,-/*,*,*,*,*,*,*,*,*=>5;
    6/-,timeout/*,*,*,*,*,*,*,*,*=>7;
    6/char,-/*,*,*,*,*,*,*,*,*=>6;
    7/-,timeout/*,*,*,*,*,*,*,*,*=>8;
    7/char,-/*,*,*,*,*,*,*,*,*=>7;
    8/-,timeout/*,*,*,*,*,*,*,*,*=>Stop1;
    8/char,-/*,*,*,*,*,*,*,*,*=>8;
    Stop1/-,timeout/*,*,*,*,*,*,*,*,*=>Stop2;
    Stop1/char,-/*,*,*,*,*,*,*,*,*=>Stop1;
    Stop2/-,timeout/*,*,*,*,*,*,*,*,*=>Idle;
    Stop2/char,-/*,*,*,*,*,*,*,*,*=>Stop2;
  END;


  FOUTP : list
    Idle/char,-/*,*,*,*,*,*,*,*,*=>start,cxs.1,cxs.2,cxs.3,cxs.4,cxs.5,
                                    cxs.6,cxs.7,cxs.8,-,0;
    Start/-,timeout/*,*,*,*,*,*,*,*,*=>start,-,-,-,-,-,-,-,-,sh,cxs.9;
    1/-,timeout/*,*,*,*,*,*,*,*,*=>start,-,-,-,-,-,-,-,-,sh,cxs.9;
    2/-,timeout/*,*,*,*,*,*,*,*,*=>start,-,-,-,-,-,-,-,-,sh,cxs.9;
```

Figure 4.10. (continued)

```
      3/-,timeout/*,*,*,*,*,*,*,*,*=>start,-,-,-,-,-,-,-,-,sh,cxs.9;
      4/-,timeout/*,*,*,*,*,*,*,*,*=>start,-,-,-,-,-,-,-,-,sh,cxs.9;
      5/-,timeout/*,*,*,*,*,*,*,*,*=>start,-,-,-,-,-,-,-,-,sh,cxs.9;
      6/-,timeout/*,*,*,*,*,*,*,*,*=>start,-,-,-,-,-,-,-,-,sh,cxs.9;
      7/-,timeout/*,*,*,*,*,*,*,*,*=>start,-,-,-,-,-,-,-,-,sh,cxs.9;
      8/-,timeout/*,*,*,*,*,*,*,*,*=>start,-,-,-,-,-,-,-,-,-,1;
      Stop1/-,timeout/*,*,*,*,*,*,*,*,*=>start,-,-,-,-,-,-,-,-,-,1;
    END;

    FOUTS : list
      Idle => Idle;
      *    => Busy;
    END;
    deftexec:0.0;
  END;

  FSM Sendbit : interlaced
    S:l,h;
    defsinit:h;
    XP.1 FROM Sendmgr.ZP.11 : 0,1;
    ZS.1 TO Medium.XS.1 : l,h;

    FNS : list
      */0=>l;
      */1=>h;
    END;

    FOUTS : list
      l=>l;
      h=>h;
    END;

    deftexec:0.0;
  END;

  FSM Sendsr1 : interlaced
    S:0,1;
    defsinit:0;
    XP.1 FROM Sendmgr.ZP.2 : 0,1;
    XP.2 FROM Sendmgr.ZP.10 : sh;
    XP.3 FROM Sendsr2.ZP.2 : 0,1;
    ZP.1 TO Sendsr2.XP.2 : sh;
    ZP.2 UNCONNECTED : 0,1;
    ZS.1 TO Sendmgr.XS.9 : 0,1;

    FNS :list
      */0,-,-/ =>0;
      */1,-,-/ =>1;
      */-,-,0/ =>0;
      */-,-,1/ =>1;
    END;
    FOUTP : list
      */-,sh,-/ =>sh,cs;
    END;
    FOUTS : procedure
      ntzs.1:=nts;
    END;
    deftexec:0.0;
  END;
```

Figure 4.10. (continued)

```
FSM Sendsr2 : interlaced
  S:0,1;
  defsinit:0;
  XP.1 FROM Sendmgr.ZP.3 : 0,1;
  XP.2 FROM  Sendsr1.ZP.1 : sh;
  XP.3 FROM Sendsr3.ZP.2 : 0,1;
  ZP.1 TO Sendsr3.XP.2 : sh;
  ZP.2 TO Sendsr1.XP.3 : 0,1;
  ZS.1 UNCONNECTED : 0,1;
  FNS :list
    */0,-,-/ =>0;
    */1,-,-/ =>1;
    */-,-,0/ =>0;
    */-,-,1/ =>1;
  END;
  FOUTP : list
    */-,sh,-/ =>sh,cs;
  END;
  FOUTS : procedure
    ntzs.1:=nts;
  END;

  deftexec:0.0;
END;

FSM Sendsr3 : interlaced
  S:0,1;
  defsinit:0;
  XP.1 FROM Sendmgr.ZP.4 : 0,1;
  XP.2 FROM  Sendsr2.ZP.1 : sh;
  XP.3 FROM Sendsr4.ZP.2 : 0,1;
  ZP.1 TO Sendsr4.XP.2 : sh;
  ZP.2 TO Sendsr2.XP.3 : 0,1;
  ZS.1 UNCONNECTED : 0,1;
  FNS :list
    */0,-,-/ =>0;
    */1,-,-/ =>1;
    */-,-,0/ =>0;
    */-,-,1/ =>1;
  END;
  FOUTP : list
    */-,sh,-/ =>sh,cs;
  END;
  FOUTS : procedure
    ntzs.1:=nts;
  END;

  deftexec:0.0;
END;
```

Figure 4.10. (continued)

```
FSM Sendsr4 : interlaced
  S:0,1;
  defsinit:0;
  XP.1 FROM Sendmgr.ZP.5 : 0,1;
  XP.2 FROM  Sendsr3.ZP.1 : sh;
  XP.3 FROM Sendsr5.ZP.2 : 0,1;
  ZP.1 TO Sendsr5.XP.2 : sh;
  ZP.2 TO Sendsr3.XP.3 : 0,1;
  ZS.1 UNCONNECTED : 0,1;
  FNS :list
    */0,-,-/ =>0;
    */1,-,-/ =>1;
    */-,-,0/ =>0;
    */-,-,1/ =>1;
  END;
  FOUTP : list
    */-,sh,-/ =>sh,cs;
  END;
  FOUTS : procedure
    ntzs.1:=nts;
  END;

  deftexec:0.0;
END;

FSM Sendsr5 : interlaced
  S:0,1;
  defsinit:0;
  XP.1 FROM Sendmgr.ZP.6 : 0,1;
  XP.2 FROM  Sendsr4.ZP.1 : sh;
  XP.3 FROM Sendsr6.ZP.2 : 0,1;
  ZP.1 TO Sendsr6.XP.2 : sh;
  ZP.2 TO Sendsr4.XP.3 : 0,1;
  ZS.1 UNCONNECTED : 0,1;
  FNS :list
    */0,-,-/ =>0;
    */1,-,-/ =>1;
    */-,-,0/ =>0;
    */-,-,1/ =>1;
  END;
  FOUTP : list
    */-,sh,-/ =>sh,cs;
  END;
  FOUTS : procedure
    ntzs.1:=nts;
  END;
  deftexec:0.0;

END;
```

Figure 4.10. (continued)

```
FSM Sendsr6 : interlaced
  S:0,1;
  defsinit:0;
  XP.1 FROM Sendmgr.ZP.7 : 0,1;
  XP.2 FROM  Sendsr5.ZP.1 : sh;
  XP.3 FROM Sendsr7.ZP.2 : 0,1;
  ZP.1 TO  Sendsr7.XP.2 : sh;
  ZP.2 TO Sendsr5.XP.3 : 0,1;
  ZS.1 UNCONNECTED : 0,1;
  FNS :list
    */0,-,-/ =>0;
    */1,-,-/ =>1;
    */-,-,0/ =>0;
    */-,-,1/ =>1;
  END;
  FOUTP : list
    */-,sh,-/ =>sh,cs;
  END;
  FOUTS : procedure
    ntzs.1:=nts;
  END;

  deftexec:0.0;
END;

FSM Sendsr7 : interlaced
  S:0,1;
  defsinit:0;
  XP.1 FROM Sendmgr.ZP.8 : 0,1;
  XP.2 FROM  Sendsr6.ZP.1 : sh;
  XP.3 FROM Sendsr8.ZP.2 : 0,1;
  ZP.1 TO Sendsr8.XP.2 : sh;
  ZP.2 TO Sendsr6.XP.3 : 0,1;
  ZS.1 UNCONNECTED : 0,1;
  FNS :list
    */0,-,-/ =>0;
    */1,-,-/ =>1;
    */-,-,0/ =>0;
    */-,-,1/ =>1;
  END;
  FOUTP : list
    */-,sh,-/ =>sh,cs;
  END;
  FOUTS : procedure
    ntzs.1:=nts;
  END;

  deftexec:0.0;
END;
```

Figure 4.10. (continued)

```
FSM Sendsr8 : interlaced
  S:0,1;
  defsinit:0;
  XP.1 FROM Sendmgr.ZP.9 : 0,1;
  XP.2 FROM Sendsr7.ZP.1 : sh;
  XP.3 UNCONNECTED : 0,1;
  ZP.1 UNCONNECTED : sh;
  ZP.2 TO Sendsr7.XP.3 : 0,1;
  ZS.1 UNCONNECTED : 0,1;
  FNS :list
    #/0,-,-/ =>0;
    #/1,-,-/ =>1;
    #/-,-,0/ =>0;
    #/-,-,1/ =>1;
  END;
  FOUTP : list
    #/-,sh,-/ =>sh,cs;
  END;
  FOUTS : procedure
    ntzs.1:=nts;
  END;

  deftexec:0.0;
END;

CLK Sendclk : interlaced
  XP.1 FROM Sendmgr.ZP.1 : start,reset;
  ZP.1 TO Sendmgr.XP.2 : timeout;
  ZS.1 UNCONNECTED : reset,running,expired;
  deftclk:1.0;
END;

DELS Medium :interlaced
  XS.1 FROM Sendbit.ZS.1 : l,h;
  ZS.1 TO Rcvmgr.XS.1,rcvdet.XS.1 : l,h;
  defzsinit: h ;
  deftdel : 3.0;
END;

FSM Rcvmgr : interlaced
  S:idle,start,1,2,3,4,5,6,7,8;
  defsinit:idle;
  XP.1 FROM rcvdet.ZP.1 :r,f;
  XP.2 FROM Rcvclk1.ZP.1 : timeout;
  XP.3 FROM Rcvclk2.ZP.1 : timeout;
  XS.1 FROM Medium.ZS.1 : l,h;
  ZP.1 TO Rcvclk1.XP.1 : start;
  ZP.2 TO Rcvclk2.XP.1 : start;
  ZP.3 TO Rcvsr8.XP.1 : 0,1;
  ZP.4 TO Rcvuser.XP.1 :char;

  FNS : list
    idle/f,-,-/* => start;
    start/*,timeout,-/* => 1;
    1/*,-,timeout/* => 2;
    2/*,-,timeout/* => 3;
    3/*,-,timeout/* => 4;
    4/*,-,timeout/* => 5;
    5/*,-,timeout/* => 6;
    6/*,-,timeout/* => 7;
    7/*,-,timeout/* =>8;
    8/*,-,timeout/* =>idle;
  END;
```

Figure 4.10. (continued)

```
    FOUTP : list
      idle/f,-,-/* =>start,-,-,-;
      start/*,timeout,-/h =>-,start,1,-;
      start/*,timeout,-/l =>-,start,0,-;
      1/*,-,timeout/h =>-,start,1,-;
      1/*,-,timeout/l =>-,start,0,-;
      2/*,-,timeout/h =>-,start,1,-;
      2/*,-,timeout/l =>-,start,0,-;
      3/*,-,timeout/h =>-,start,1,-;
      3/*,-,timeout/l =>-,start,0,-;
      4/*,-,timeout/h =>-,start,1,-;
      4/*,-,timeout/l =>-,start,0,-;
      5/*,-,timeout/h =>-,start,1,-;
      5/*,-,timeout/l =>-,start,0,-;
      6/*,-,timeout/h =>-,start,1,-;
      6/*,-,timeout/l =>-,start,0,-;
      7/*,-,timeout/h =>-,start,1,-;
      7/*,-,timeout/l =>-,start,0,-;
      8/*,-,timeout/* => -,-,-,char;
    END;
    deftexec : 0.0;
  END;

  FSM Rcvsr8 : interlaced
    S:0,1;
    defsinit:0;
    XP.1 FROM Rcvmgr.ZP.3 : 0,1;
    ZP.1 TO Rcvsr7.XP.1 : 0,1;
    ZS.1 TO Rcvuser.XS.8 :  0,1;

    FNS :  procedure
      if (cxp.1='0') then nts:='0';
      if (cxp.1='1') then nts:='1';
    END;

    FOUTP : procedure
       if (cxp.1<> '-') then
           if (cs='0' ) then ntzp.1:='0'
                        else ntzp.1:='1';
    END;

    FOUTS : procedure
      ntzs.1:=nts;
    END;

    deftexec:0.0;
  END;

  FSM Rcvsr7 : interlaced
    S:0,1;
    defsinit:0;
    XP.1 FROM Rcvsr8.ZP.1 : 0,1;
    ZP.1 TO Rcvsr6.XP.1 : 0,1;
    ZS.1 TO Rcvuser.XS.7 :  0,1;

    FNS :  procedure
      if (cxp.1='0') then nts:='0';
      if (cxp.1='1') then nts:='1';
    END;
```

Figure 4.10. (continued)

```
    FOUTP : procedure
       if (cxp.1<> '-') then
           if (cs='0') then ntzp.1:='0'
                       else ntzp.1:='1';
    END;

    FOUTS : procedure
      ntzs.1:=nts;
    END;

    deftexec:0.0;
  END;

  FSM Rcvsr6 : interlaced
    S:0,1;
    defsinit:0;
    XP.1 FROM Rcvsr7.ZP.1 : 0,1;
    ZP.1 TO Rcvsr5.XP.1 : 0,1;
    ZS.1 TO Rcvuser.XS.6 :  0,1;

    FNS :  procedure
      if (cxp.1='0') then nts:='0';
      if (cxp.1='1') then nts:='1';
    END;

    FOUTP : procedure
       if (cxp.1<> '-') then
           if (cs='0') then ntzp.1:='0'
                       else ntzp.1:='1';
    END;

    FOUTS : procedure
      ntzs.1:=nts;
    END;

    deftexec:0.0;
  END;

  FSM Rcvsr5 : interlaced
    S:0,1;
    defsinit:0;
    XP.1 FROM Rcvsr6.ZP.1 : 0,1;
    ZP.1 TO Rcvsr4.XP.1 : 0,1;
    ZS.1 TO Rcvuser.XS.5 :  0,1;

    FNS :  procedure
      if (cxp.1='0') then nts:='0';
      if (cxp.1='1') then nts:='1';
    END;

    FOUTP : procedure
       if (cxp.1<> '-') then
           if (cs='0') then ntzp.1:='0'
                       else ntzp.1:='1';
    END;

    FOUTS : procedure
      ntzs.1:=nts;
    END;

    deftexec:0.0;
  END;
```

Figure 4.10. (continued)

```
FSM Rcvsr4 : interlaced
  S:0,1;
  defsinit:0;
  XP.1 FROM Rcvsr5.ZP.1 : 0,1;
  ZP.1 TO Rcvsr3.XP.1 : 0,1;
  ZS.1 TO Rcvuser.XS.4 :  0,1;

  FNS :  procedure
    if (cxp.1='0') then nts:='0';
    if (cxp.1='1') then nts:='1';
  END;

  FOUTP : procedure
     if (cxp.1<> '-') then
         if (cs='0') then ntzp.1:='0'
                     else ntzp.1:='1';
  END;

  FOUTS : procedure
    ntzs.1:=nts;
  END;

  deftexec:0.0;
END;

FSM Rcvsr3 : interlaced
  S:0,1;
  defsinit:0;
  XP.1 FROM Rcvsr4.ZP.1 : 0,1;
  ZP.1 TO Rcvsr2.XP.1 : 0,1;
  ZS.1 TO Rcvuser.XS.3 :  0,1;

  FNS :  procedure
    if (cxp.1='0') then nts:='0';
    if (cxp.1='1') then nts:='1';
  END;

  FOUTP : procedure
     if (cxp.1<> '-') then
         if (cs='0') then ntzp.1:='0'
                     else ntzp.1:='1';
  END;

  FOUTS : procedure
    ntzs.1:=nts;
  END;

  deftexec:0.0;
END;
```

Figure 4.10. (continued)

```
FSM Rcvsr2 : interlaced
  S:0,1;
  defsinit:0;
  XP.1 FROM Rcvsr3.ZP.1 : 0,1;
  ZP.1 TO Rcvsr1.XP.1 : 0,1;
  ZS.1 TO Rcvuser.XS.2 :  0,1;

  FNS :  procedure
    if (cxp.1='0') then nts:='0';
    if (cxp.1='1') then nts:='1';
  END;

  FOUTP : procedure
     if (cxp.1<> '-') then
         if (cs='0') then ntzp.1:='0'
                     else ntzp.1:='1';
  END;

  FOUTS : procedure
    ntzs.1:=nts;
  END;

  deftexec:0.0;
END;

FSM Rcvsr1 : interlaced
  S:0,1;
  defsinit:0;
  XP.1 FROM Rcvsr2.ZP.1 : 0,1;
  ZP.1 UNCONNECTED: 0,1;
  ZS.1 TO Rcvuser.XS.1 :  0,1;

  FNS :  procedure
    if (cxp.1='0') then nts:='0';
    if (cxp.1='1') then nts:='1';
  END;

  FOUTP : procedure
     if (cxp.1<> '-') then
         if (cs='0') then ntzp.1:='0'
                     else ntzp.1:='1';
  END;

  FOUTS : procedure
    ntzs.1:=nts;
  END;

  deftexec:0.0;
END;
```

Figure 4.10. (continued)

```
DER rcvdet  : interlaced
  XS.1 FROM Medium.ZS.1 :l,h;
  ZP.1 TO Rcvmgr.XP.1 : r,f;
END;

CLK Rcvclk1 : interlaced
  XP.1 FROM Rcvmgr.ZP.1 : start;
  ZP.1 TO Rcvmgr.XP.2  : timeout;
  ZS.1 UNCONNECTED : reset,running,expired;
  deftclk:1.5;
END;

CLK Rcvclk2 : interlaced
  XP.1 FROM Rcvmgr.ZP.2 : start;
  ZP.1 TO Rcvmgr.XP.3  : timeout;
  ZS.1 UNCONNECTED : reset,running,expired;
  deftclk:1.0;
END;


VARHISTORY TraceMedium : regular
  VARIABLES : Medium.cxs.1;
  DTHISTORY : 1.0;
  CHECKOPT : everytimechange;
END;

VARHISTORY TraceSendconverter : conditional
  VARIABLES : Sendconverter.czp.1;
  CONDITION : Sendconverter.czp.1 <> '-';
  CHECKOPT : everytimechange;
END;

VARHISTORY TraceRcvuser : conditional
  VARIABLES : Rcvuser.czs.1;
  CONDITION : Rcvuser.czp.1<>'-';
  CHECKOPT : everytimechange;
END;

VARHISTORY TraceTester : conditional
  VARIABLES : Tester.czp.1;
  CONDITION : Tester.czp.1<>'-';
  CHECKOPT : everytimechange;
END;

INIT
  tbeg : 1.0;
  tend : 150.0;
  mulpulsecheck : true;
END;
```

Figure 4.10. (continued)

(FSM). The functions of the components in the sender and receiver of the Start-Stop link have been explained by [Piatkowski 1981]. Here we highlight the use of component features in simulating the Start-Stop link and its Test Controller.

Sendmgr, a finite-state machine (FSM), incorporates the serial sending of the start bit, the eight-bit user data and two stop bits to the receiver. Sendmgr has twelve different states as specified in the S line of the Sendmgr specification. It has two pulsed inputs, XP.1 and XP.2, connected to Senduser.ZP.1 and Sendclk.ZP.1, respectively. There are nine static inputs with XS.1 to XS.8 connected to the static outputs of Senduser(ZS.1 to ZS.8); the static input, XS.9, connects to the static output of the shift register (Sendsrl.ZS.1). The first pulsed output (ZP.1) connects to the pulsed input of Sendclk(XP.1). The next eight pulsed outputs (ZP.2 to ZP.9) connect to the XP.1 of Sendsrl, Sendsr2, ---- and Sendsr8, respectively. The tenth pulsed output, ZP.10, connects to Sendsrl.XP.2. The eleventh pulsed output, ZP.11, connects to Sendbit.XP.1. The static output, ZS.1, connects to Sendconverter.XS.9 of the test model to indicate whether Sendmgr is in the Idle or Busy state. The list option of the FNS enumerates possible state transitions based on the mapping of cs, xp's and xs's to nts. Similarly, the list option of the FOUTP enumerates all possible combinations of cs, xp's and xs's to map into the array of next pulsed outputs, ntzp's. The first line of the FOUTP list demonstrates the use of current local static input variables, cxs.1, cxs.2 and so on, as the value of the ntzp's. The list option of FOUTS enumerates the mappings

of the current state into the current static output. The default execution time is zero.

Further on in the specification of the shift registers in the receiver of the Start-Stop link, the procedure option is used in FNS, FOUTP and FOUTS. For example, Rcvsr8 (FSM), a one-bit shift register, has a pulsed input, XP.1, connected to Rcvmgr.ZP.3, a pulsed output, ZP.1, connected to Rcvsr7.XP.1, and a static output, zs.1, connected to Rcvvuser.XS.8. The FOUTP procedure indicates that if the current pulsed input is '0', then the next state value, nts, is '0', else if the current pulsed input is '1', then the next state value, nts, is '1'. The FOUTP and FOUTS are also specified via a local PASCAL procedure.

Sendclk a clock (CLK), will pulse out a timeout pulse to Sendmgr 1.0 time unit after the arrival of a start pulse to its pulsed input. Sendclk has a pulsed input, XP.1, connected to Sendmgr.ZP.1, a pulsed output, ZP.1, connected to Sendmgr.XP.2. The static output of Sendclk is not connected. The clock period is 1.0 time unit.

Medium, a static delay (DELS), acts as a delay between the sender and the receiver of the Start-Stop link. The static input, XS.1, connects to the Sendbit.ZS.1; and the static output, ZS.1, connects to both Rcvmgr.XS.1 and Rcvdet.XS.1. The default initial static output value of Medium is h and its delay time is arbitrarily 3.0 time units.

Rcvdet, a derivative (DER), senses the change of the static output of Medium from low (l) to high (h) or high (h) to low (l) to pulse out a rising (r) or falling (f) pulse to Sendmgr respectively. Rcvdet has a

static input, XS.1, connected to Medium.ZS.1 and a pulsed output, ZP.1, connected to Rcvmgr.XP.1.

2. Test controller

The test controller interfaces the user with the Start-Stop model and automates much of the testing of the link. Figure 4.10 presents the SAN specification of the Test Controller, which is built up from five components: Senduser(ENV), Sendconverter(ENV), Sendque(QUE), Rcvuser(ENV) and Tester(ENV). The Test controller makes it possible for the user to send a stream of eight-bit bytes for transmission on the link.

Since the sender accepts an eight-bit block (data) only if the Sendmgr is in Idle state, the Sendconverter allows the data from the Senduser to load into the FIFO queue, Sendque, only if the Sendmgr is in Idle state. The Sendconverter also converts the eight static inputs of the eight-bit byte into a pulsed output containing the eight-bit information as a single string. In this case, one FIFO queue is needed to save the eight bits of data; otherwise eight FIFO queues will be needed. The pulsed output of Sendconverter is loaded into the FIFO queue, Sendque. Sendque saves all the data sent out by Senduser; the data will be used by the Tester to check if the data received by the Rcvuser is good or bad.

Rcvuser latches the eight-bit byte from the eight-bit shift register in the receiver and converts the eight-bit byte into a single string. When Rcvuser receives data from the receiver of the

Start-Stop link, it sends out a char pulse to the Tester, which in return pulses out a deq pulse to Sendque for the oldest data in the queue. The data from Sendque is compared with the data received by the Rcvuser. If they are equal, Tester pulses out a goodbyte pulse; else, it pulses out a badbyte pulse.

3. Performance traces

The Start-Stop link simulation model has four variable history instances to trace the execution of the Start-Stop link. The first trace, named TraceMedium, is a regular variable history instance. It records the value of the static input of Medium at every 1.0 time unit interval and only one sample is recorded just before the current simulation time is advanced. The second trace is named TraceSendconverter; it is a conditional variable history instance. The pulsed output of Sendconverter is checked just before simulation time is advanced. If a pulse appears at Sendconverter.czp.1, then the value of the pulse and the current simulation time is recorded. Similarly, the third and the fourth conditional variable history instances named TraceRcvuser and TraceTester trace the pulse being received at Rcvuser and the pulse generated by Tester, respectively.

Figure 4.11 presents the traces of the TraceSendconverter, TraceRcvuser, and TraceTester instances. From the instance TraceSendconverter, we notice that an eight-bit byte, '01010101', is sent out from Senduser at tnow=1.0, '01010001' is sent out from Senduser at tnow=13.0, and so on. From the instance TraceRcvuser we notice that

an eight-bit byte, '01010101' is received by Rcvuser at tnow=13.5, '01010001' is received by Rcvuser at tnow=25.5, and so on. From the instance TraceTester, we notice that a good byte is received by the Rcvuser at tnow=13.5, 25.5, 37.5 and so on. We also notice that there is a 12.5 time unit delay before the Rcvuser received a byte sent from the Senduser. The 12.5 time unit delay is the result of time needed to send a start pulse, which takes 1.0 time unit; the eight-bit bytes takes 8.0 time units; the Medium has a delay of 3.0 time unit; and also the 0.5 time unit delay for the receiving clock to sample the middle of the valid data.

Figure 4.12 presents the output of TraceMedium. It shows the value of the static input of Medium at every unit time interval between the simulation time period 1.0 to 150.0. From the trace, we can observe the serial bit pattern generated by the Sendmgr. As indicated by the instance TraceSendconverter an eight-bit byte '01010101' is sent out from Senduser at tnow=1.0. From the output of TraceMedium, we notice that a start bit, l, is first sent out to the Medium at tnow=1.0. It is then followed by eight bits of data, '01010101', as denoted by the value of the static input of Medium from tnow=2.0 to 9.0, which is l h l h l h l h. At the end of sending the eight-bit byte, two stop bits, h, are sent out at tnow=10.0 and 11.0 as indicated in Figure 4.11. The static input of Medium stays at h until tnow=13.0. At tnow=13.0, a new eight-bit byte is sent out by Senduser as indicated by the output of TraceSendconverter. And so on.

```
VARHIST  TraceSendconverter                : conditional
Sendconverter.czp.1                         Time
01010101                                    1.000000000E+00
01010001                                    1.300000000E+01
11011001                                    2.500000000E+01
10010001                                    3.700000000E+01
00000001                                    4.900000000E+01
10001011                                    6.100000000E+01
10011111                                    7.300000000E+01
11010011                                    8.500000000E+01
11010011                                    9.700000000E+01
11010000                                    1.090000000E+02
11110000                                    1.210000000E+02
11110001                                    1.330000000E+02
11110111                                    1.450000000E+02

VARHIST  TraceRcvuser                      : conditional
Rcvuser.czs.1                               Time
01010101                                    1.350000000E+01
01010001                                    2.550000000E+01
11011001                                    3.750000000E+01
10010001                                    4.950000000E+01
00000001                                    6.150000000E+01
10001011                                    7.350000000E+01
10011111                                    8.550000000E+01
11010011                                    9.750000000E+01
11010011                                    1.095000000E+02
11010000                                    1.215000000E+02
11110000                                    1.335000000E+02
11110001                                    1.455000000E+02

VARHIST  TraceTester                       : conditional
Tester.czp.1                                Time
Goodbyte                                    1.350000000E+01
Goodbyte                                    2.550000000E+01
Goodbyte                                    3.750000000E+01
Goodbyte                                    4.950000000E+01
Goodbyte                                    6.150000000E+01
Goodbyte                                    7.350000000E+01
Goodbyte                                    8.550000000E+01
Goodbyte                                    9.750000000E+01
Goodbyte                                    1.095000000E+02
Goodbyte                                    1.215000000E+02
Goodbyte                                    1.335000000E+02
Goodbyte                                    1.455000000E+02
```

Figure 4.11. Traces of the instances, TraceSendconverter, TraceRcvuser and TraceTester

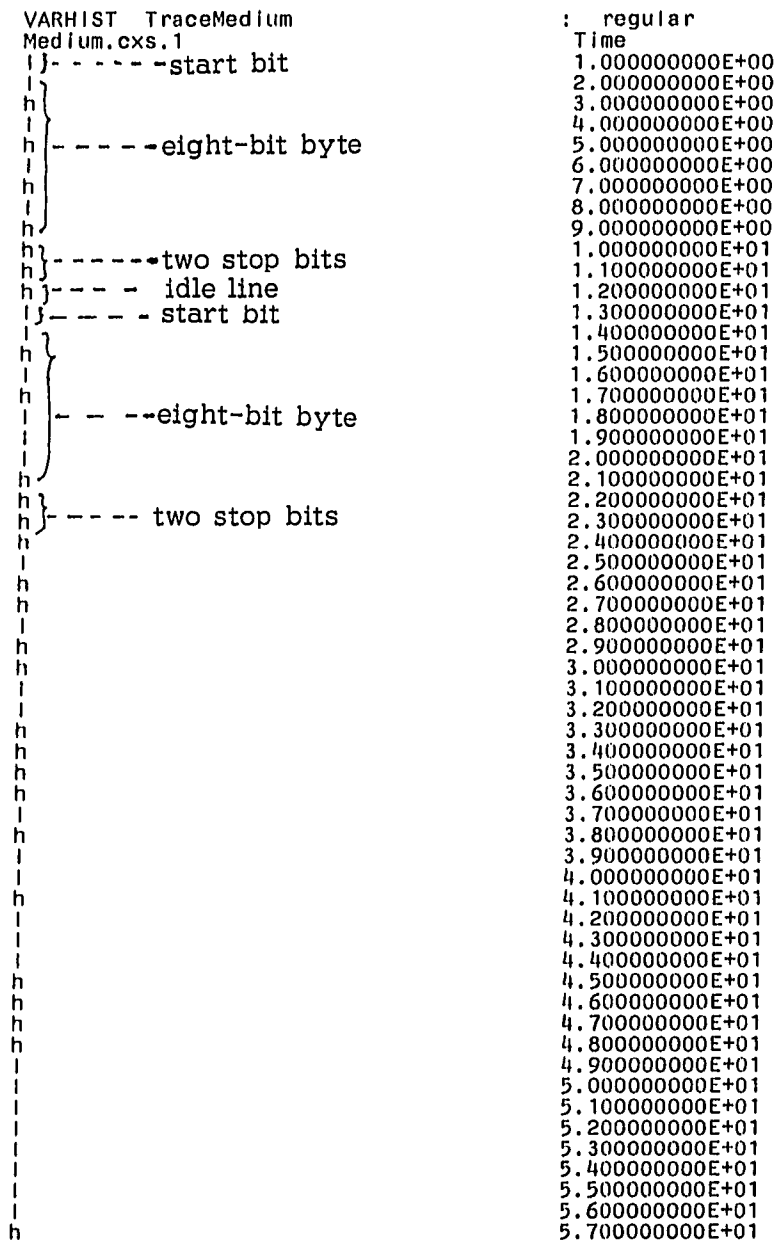


Figure 4.12. Traces of the instance, TraceMedium

```
h                                5.800000000E+01
h                                5.900000000E+01
h                                6.000000000E+01
h                                6.100000000E+01
h                                6.200000000E+01
h                                6.300000000E+01
h                                6.400000000E+01
h                                6.500000000E+01
h                                6.600000000E+01
h                                6.700000000E+01
h                                6.800000000E+01
h                                6.900000000E+01
h                                7.000000000E+01
h                                7.100000000E+01
h                                7.200000000E+01
h                                7.300000000E+01
h                                7.400000000E+01
h                                7.500000000E+01
h                                7.600000000E+01
h                                7.700000000E+01
h                                7.800000000E+01
h                                7.900000000E+01
h                                8.000000000E+01
h                                8.100000000E+01
h                                8.200000000E+01
h                                8.300000000E+01
h                                8.400000000E+01
h                                8.500000000E+01
h                                8.600000000E+01
h                                8.700000000E+01
h                                8.800000000E+01
h                                8.900000000E+01
h                                9.000000000E+01
h                                9.100000000E+01
h                                9.200000000E+01
h                                9.300000000E+01
h                                9.400000000E+01
h                                9.500000000E+01
h                                9.600000000E+01
h                                9.700000000E+01
h                                9.800000000E+01
h                                9.900000000E+01
h                                1.000000000E+02
h                                1.010000000E+02
h                                1.020000000E+02
h                                1.030000000E+02
h                                1.040000000E+02
h                                1.050000000E+02
h                                1.060000000E+02
h                                1.070000000E+02
h                                1.080000000E+02
h                                1.090000000E+02
h                                1.100000000E+02
h                                1.110000000E+02
h                                1.120000000E+02
h                                1.130000000E+02
h                                1.140000000E+02
h                                1.150000000E+02
h                                1.160000000E+02
h                                1.170000000E+02
```

Figure 4.12. (continued)

```
h                1.180000000E+02
h                1.190000000E+02
h                1.200000000E+02
l                1.210000000E+02
h                1.220000000E+02
h                1.230000000E+02
h                1.240000000E+02
h                1.250000000E+02
l                1.260000000E+02
l                1.270000000E+02
l                1.280000000E+02
l                1.290000000E+02
h                1.300000000E+02
h                1.310000000E+02
h                1.320000000E+02
l                1.330000000E+02
h                1.340000000E+02
h                1.350000000E+02
h                1.360000000E+02
h                1.370000000E+02
l                1.380000000E+02
l                1.390000000E+02
l                1.400000000E+02
h                1.410000000E+02
h                1.420000000E+02
h                1.430000000E+02
h                1.440000000E+02
l                1.450000000E+02
h                1.460000000E+02
h                1.470000000E+02
h                1.480000000E+02
h                1.490000000E+02
l                1.500000000E+02
```

Figure 4.12. (continued)

D. Advanced Data Communication Control Procedures Simulation

This section illustrates the use of the SAN and the SAS in simulating a fairly complicated data communication protocol, namely the Advanced Data Communication Control Procedures (ADCCP); ADCCP is the American National Standard version of the High Level Data Link (HDLC). The ADCCP is a set of algorithms for passing arbitrary bit sequences (or messages) between devices connected by a bit serial communication medium.

The work presented in this section is based on the work of Dayun He in using the SAN and the SAS to simulate and test an ADCCP system. The SAN model of the ADCCP system is based on the formal ADCCP station model developed by Piatkowski [Piatkowski 1979]. The details of the ADCCP SAN model and the results of the simulation testing are presented in a paper by Piatkowski, Ip and He [Piatkowski et al. 1982]. Part of the following paragraphs are taken from this paper.

In summary, a SAN model of a one-way point to point channel consisting of the lowest three levels of two ADCCP stations were simulated and tested. A block diagram of the channel is shown in Figure 4.13. The ADCCP channel components are shown enclosed in the dashed boundary; the additional external components are used to control and monitor the system for exercising and testing. The kind of SAN component(s) comprising each block in Figure 4.13 is indicated in the figure by the expressions within the parentheses; e.g., ClockMgr a CFP, FCSSend a subsystem of one CFP and eighteen FSMs. There are four ENVs, one CFP and a CLK for the

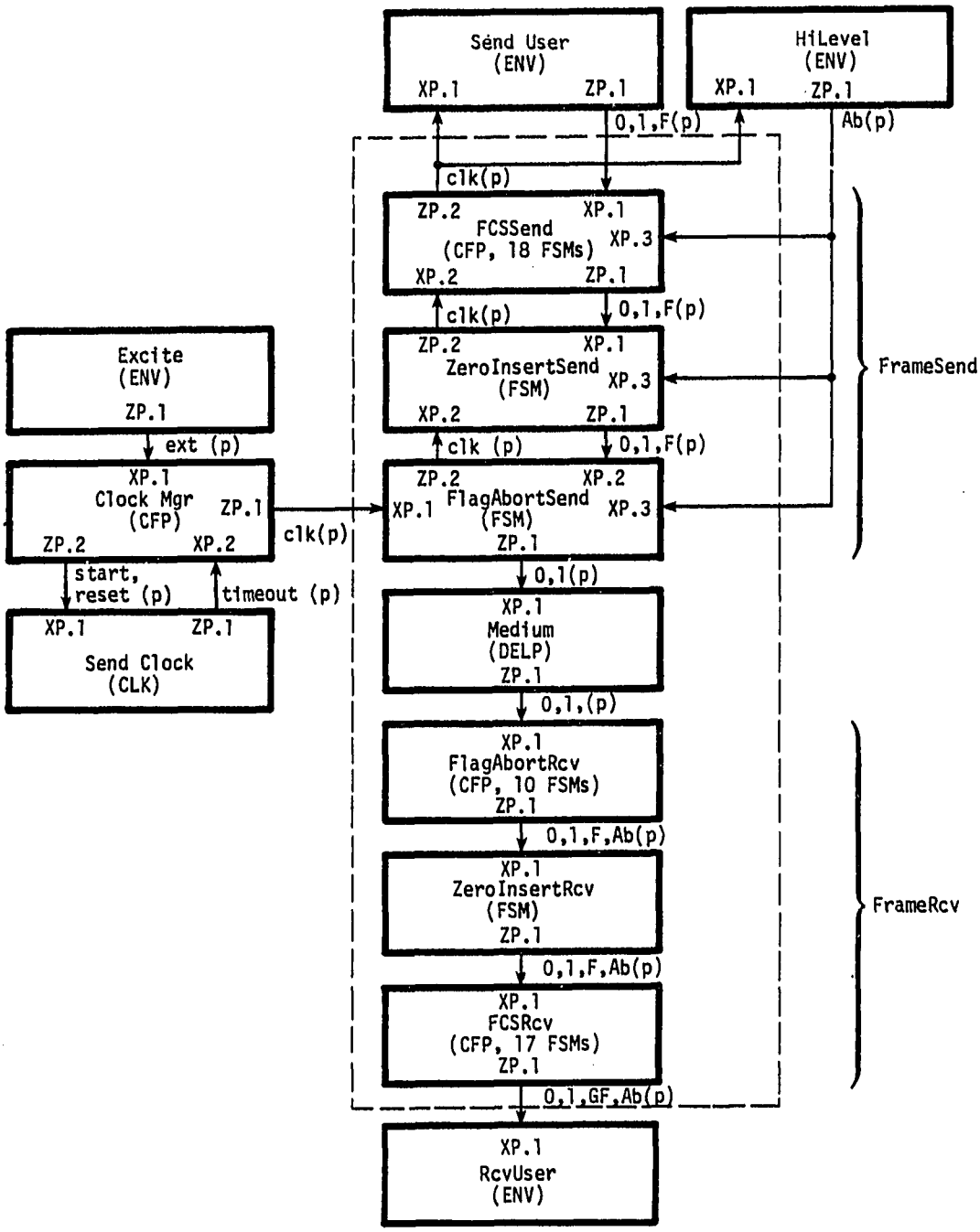


Figure 4.13. Overview structure of a one-way point-to-point ADCCP channel with manual tester

control and monitoring components (those components reside outside the dashed boundary). There are twenty FSMs and one CFP in the send station, a DELP in the Medium, and twenty-eight FSMs and two CFPs in the receive station of the channel.

The model specifies an automatic repeating clock for channel timing and three terminal environments for manual data input and output monitoring. A formatted listing of the variable history traces for a sample run is shown in Figure 4.14. In it, we see that the sending side (represented by SendUser.czp.1) transmitted the sequence

1 1 (FCS) F 0 1 1 1 1 1 1 0 (FCS) F 1 1 Ab

The receiving side (represented by FCSRcv.czp.1) correctly ignored the bits preceeding the initial flag, checked the FCS (Frame Check Sequence) for the first frame, passed the first frame up to the receiving user, and detected the Abort. Note the proper timing for the flag bits, zero insertion and deletion, and FCS handling.

In the second simulation run, a modified version of the same model with automated (procedural) environment control and monitoring was created by replacing the external components and variable history of the previous version. This version provides for automatic high-speed long-length random testing of the channel model. In this case, the sending sequence was generated randomly and a single test monitoring ENV determined that the receiver correctly delivered messages as sent by the sender. The block diagram of the ADCCP channel with automatic tester is shown in Figure 4.15. A formatted listing of the variable history traces for a sample run is shown in Figure 4.16.

ClockMgr.czp.1	FCSSend.czp.2	SendUser.czp.1	FCSRcv.czp.1	HiLevel.czp.1	Time
-	-	-	-	-	0
clk	clk	1	-	-	1
clk	clk	1	-	-	2
clk	clk	F	-	-	3
clk	-	-	-	-	4
clk	-	-	-	-	5
clk	-	-	-	-	6
clk	-	-	-	-	7
clk	-	-	-	-	8
clk	-	-	-	-	9
clk	-	-	-	-	10
clk	-	-	-	-	11
clk	-	-	-	-	12
clk	-	-	-	-	13
clk	-	-	-	-	14
clk	-	-	-	-	15
clk	-	-	-	-	16
clk	-	-	-	-	17
clk	-	-	-	-	18
clk	-	-	-	-	19
clk	-	-	-	-	20
clk	-	-	-	-	21
clk	-	-	-	-	22
clk	-	-	-	-	23
clk	-	-	-	-	24
clk	-	-	-	-	25
clk	-	-	-	-	26
clk	clk	0	-	-	27
clk	clk	1	-	-	28
clk	clk	1	-	-	29
clk	clk	1	-	-	30
clk	clk	1	-	-	31
clk	clk	1	-	-	32
clk	-	-	-	-	33
clk	clk	1	-	-	34

Figure 4.14. Traces for a sample run of the ADCCP channel with manual tester

clk	clk	0	0	-	35
clk	clk	F	1	-	36
clk	-	-	1	-	37
clk	-	-	1	-	38
clk	-	-	1	-	39
clk	-	-	1	-	40
clk	-	-	-	-	41
clk	-	-	1	-	42
clk	-	-	0	-	43
clk	-	-	1	-	44
clk	-	-	0	-	45
clk	-	-	0	-	46
clk	-	-	0	-	47
clk	-	-	0	-	48
clk	-	-	0	-	49
clk	-	-	0	-	50
clk	-	-	1	-	51
clk	-	-	0	-	52
clk	-	-	1	-	53
clk	-	-	0	-	54
clk	-	-	1	-	55
clk	-	-	0	-	56
clk	-	-	1	-	57
clk	-	-	1	-	58
clk	-	-	0	-	59
clk	clk	1	GF	-	60
clk	clk	1	-	-	61
clk	clk	-	-	Ab	62
clk	-	-	-	-	63
clk	-	-	-	-	64
clk	-	-	-	-	65
clk	-	-	-	-	66
clk	-	-	-	-	67
clk	-	-	Ab	-	68

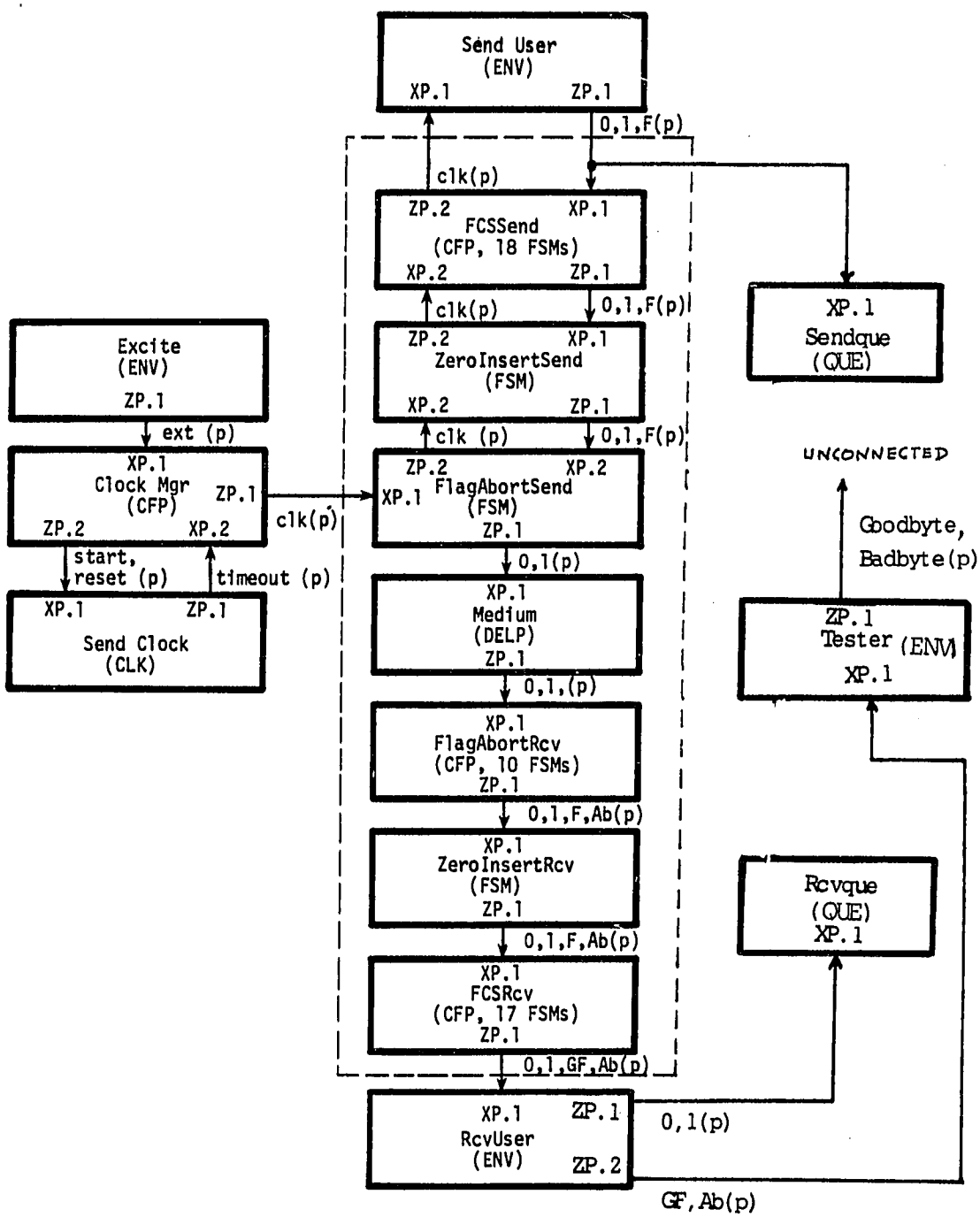


Figure 4.15. Overview structure of a one-way point-to-point ADCCP channel with automatic tester

Tester.czp.1	Time
Goodframe	4.100000000E+01
Goodframe	9.400000000E+01
Goodframe	1.620000000E+02
Goodframe	1.870000000E+02

Figure 4.16. Traces for a sample run of the ADCCP channel with automatic tester

The simulation and testing of the ADCCP system demonstrates that the SAN and the SAS are capable of simulating and testing complex protocol systems and with the help of the procedural environment, random sequences of bit pattern, can be generated to test the simulated system. In the process of coding the SAN model of the ADCCP protocols, Dayun He found several errors in the graphical ADCCP model [Piatkowski 1979]; these errors were verified by SAS simulation runs. Equally important, an unknown error in the ADCCP was discovered through a SAS run.

V. DISCUSSION AND CONCLUSIONS

A. SAN and SAS Model Size

The State Architecture Simulator (SAS) is running on a VAX 11/780 at Iowa State University using the VAX VMS V2.0 operating system. The current version of SAS consists of approximately 10K lines of PASCAL source code using 800 blocks of memory with 512 bytes per block. The source code takes 5 cpu minutes to compile. This is a one-time operation. The cpu time required to transform, compile, and link the simple SAN model (Figure 4.2) with 47 lines of specification code into the executable simulation file is about 30 seconds. Whereas for the ADCCP model with 1625 lines of specification code, the total cpu time required is about one minute.

SAS is dimensioned to handle up to 50 FSMs, 20 each of CFPs, CFSs, and ENVs, and 10 each of DELPs, DELSs, QUEs, CLKs, and DERs. Each component can contain up to 20 input/output streams of each kind (XP, XS, ZP, or ZS). Each pulsed and static output can have up to 10 different destinations (fanout). The maximum number of values that a state set can have is 100 and the maximum number of different string values used in the local FNS, FOUTP, and FOUTS procedures in the SAN model is 1000.

The above dimensions have been chosen so that a fairly complicated protocol, such as the ADCCP channel, can be represented in SAN and exercised in the SAS environment, while at the same time the amount of memory space needed is under control. For example the ADCCP channel needs 67 blocks for the SAN model and 350 blocks for the executable

simulation file. There is no reason that the SAS dimensions can not be extended to allow simulation of more complicated protocols, provided the VAX 11/780 system has enough memory and the PASCAL constraints are not violated.

In the current SAS implementation, all the component names are mapped into corresponding enumerated names. This is done so that each component can be referenced via an array indexed by the enumerated name. In PASCAL the enumerated name has a range of 256 different values. This implies that the maximum number of components allowed in the SAN model is 256. However, if the component names are mapped into an integer type instead of enumerated type, the above mentioned constraints can be extended to allow $2^{31} - 1$ components.

B. Experience with SAN and SAS

SAS has been in operation since February 1982. It has been used as a teaching tool in a graduate course in Formal Methods of Protocol Design at Iowa State University. Students were using the SAN and the SAS to simulate and exercise some simple discrete components. Subsequently they formed three groups with two persons in a group to simulate and exercise a simple unidirectional IEEE 488 system with a source and three acceptors linked together on a IEEE 488 Bus. After two 90 minute lectures on the SAN and the SAS, the students felt comfortable enough in using the SAN and the SAS to simulate and test some simple discrete systems. The students also made a few remarks about the

SAN semantics and the interactive SAS messages which were confusing to them; for example, several unnecessary SAS messages which appeared on the terminal have been taken out. Right now the only messages shown on the terminal are those necessary to inform the users how to respond.

We have simulated and tested a SAN model of the start-stop communication protocol and an ADCCP channel as mentioned in the previous chapter. In the above experiments, we used all of the kinds of primitive SAN components and performance trace instances in the simulations. During the simulation process, we can observe the status of each component at different simulation times. The SAS also records traces of different component statuses (identified by the component's state, input, and output variables). The above experiments demonstrated that the SAN and the SAS can be used to simulate and test a fairly complicated protocol. On the other hand, some SAS programming bugs and design philosophy deficiencies were discovered during the experiments; for instance, the SAN syntax error messages were improved to provide better understanding for users, the update event of a component with execution time equal to zero has been changed to execute immediately after its associated start event, the random number generator was modified to provide better uniform distributions, and the performance trace was extended to record variables until the end of the simulation.

C. Reliability of SAS

There are two aspects concerning the reliability of SAS. Initially, one is concerned with how closely the SAS simulation is related to the

SAN model. Secondly, the robustness of the SAS program is considered.

SAS, a general purpose simulator, was designed to directly execute the SAN specification. In other words, the SAN model is the simulation model, unlike some modelling methods where the model has to be translated into a programming language before the simulation can be exercised. In using the SAN and the SAS to simulate and exercise the start-stop protocol and the ADCCP channel, we found that the SAS simulation does totally reflect the behavior of the SAN model. The SAS gives a higher priority to the ENV component (the ENV components are executed first for a given simulation time). This was not a property of the SAN model until SAS was built.

As for the robustness of the SAS program, each major procedure had been tested before it was incorporated in the SAS main program. Before the SAS was released, the SAS was tested by simulating a system with all kinds of components and performance trace instances. Since then, SAS has been used as a teaching tool in a graduate course in Formal Methods of Protocol Design at Iowa State University as mentioned in the previous section. I have also used the SAN and the SAS to simulate and exercise a unidirectional start-stop link. Dayun He, a Visiting Scholar from the People's Republic of China, has used the SAN and the SAS to simulate and test the ADCCP channel. Through the use by different groups of people, the SAN and the SAS were debugged and upgraded. In general, the groups who used the SAN and the SAS did not find any major problem in simulating and testing their protocol systems. However, some modifications were made as mentioned in the previous

paragraphs. Presently, Dayun He is simulating a complete ADCCP system with a primary and a secondary station. In this case, the dimensions of the SAS have been significantly extended to be able to simulate the whole ADCCP system.

D. Limitations of SAN and SAS

The experience in using the SAN and the SAS revealed to us that the currently supported version of SAN and SAS has a number of limitations. The following discussion reflects the author's opinion, in the order of significance, of the limitations that he would like to see removed. The changes which would result in major improvements for the SAN and the SAS are presented first. They are as follows:

a) The current version allows only string type SAN variables, which is very cumbersome in modelling operations with natural numeric value; e.g., sequence numbers. We propose to have integer or real type in addition to the string type SAN variable to allow PASCAL arithmetic operations on SAN variables. Internal to the SAS, the string type SAN variables would be interpreted to be either integers or reals. This is done to have a uniform data type for all SAN variables as far as SAS is concerned.

b) The current ENV type components do not have any state variable. We propose to introduce an explicit state variable into the ENV type component. The state variable is used as a

local memory to retain partial input history information of an ENV component after the component is executed.

c) The current local FNS, FOUTP, and FOUTS procedures do not allow any user declared local variables. The addition of such local variables to the FNS, FOUTP, and FOUTS procedures will greatly enhance the convenience in using local procedures. For instance, a local integer variable, i, can be used in a for loop to reference a set of outputs (for i:=1 to n do ntzp[i]:='0'), instead of writing out all the output assignments one by one.

d) The current SAN state and input/output variables are declared as simple strings. We propose to add arrays of strings to the SAN state and input/output variables so that it will be more convenient to model components with parallel data paths; e.g., defining an eight-bit byte as an array [1..8] of 0 or 1 [Piatkowski 1981].

e) The current system components are defined individually with explicit input and output connections. If a subsystem with identical components is used, like a shift register made up of a set of interconnected flip-flops, the current SAN and SAS force the user to define each individual component and their interconnections one by one. We propose to allow such built-up systems to be defined as "arrays of components," e.g., defining a shift register as an array of rippling flip-flops [Piatkowski 1981].

f) The current version has only nine primitive components. We propose to allow the user to define new types of system components built-up from the primitive component types. This idea was kept in mind in the design of the SAS so that SAS can be upgraded to allow the above new feature.

g) The current Data Input process will stop the SAS run whenever a SAN syntax error is encountered. We propose to allow the Data Input process to detect multiple SAN syntax errors before the SAS run is terminated.

h) The current version has a multiple pulsed input check option (mulpulsecheck) for each ENV component and a global mulpulse-check for the rest of the components with pulsed inputs. It would be convenient to have individual multiple pulsed input check options for each component instance; e.g., some FSM components may allow multiple pulses arriving simultaneously and others not.

i) The current initialization instance can not initialize the components' status e.g., Texec or defsinit. We propose to allow the initialization instance to have an option to re-initialize the components' status without changing the default value of the components.

j) We propose to add an additional command to the Terminal Mode Command Language to interactively suppress and invoke the trace of the SAS messages.

The above limitations are presented in order of significance: (a) the most significant and (j) the least significant. The modification steps for some of the limitations stated will affect the status of the other limitations. Hence it appears advantageous to the author to solve the afore-mentioned limitations in the following order: (b), (c), (g), (h), (j), (i), (a), (d), (e), and (f).

E. Conclusions

The author would like to point out some of his research contributions. The major contribution is the design and implementation of the general purpose State Architecture Simulator (SAS) that accepts system specifications written in the State Architecture Notation (SAN), compiles them into an executable PASCAL simulation program, and executes them. In the process of developing the SAS, he refined the SAN language proposed by Piatkowski [Piatkowski 1981] and developed or helped develop some data communication SAN models (the start-stop link and the ADCCP channel) to illustrate SAN and SAS.

The above contributions made it possible to have a teaching tool in a graduate course in Formal Methods of Protocol Design at Iowa State University. The fact that SAS allows users to observe and interact with simulations of SAN models of protocol system is a valuable tool in informal design validation.

VI. BIBLIOGRAPHY

Alfonzetti, S.; Casale, S.; and Faro, A. "A Formal Description of the DTE Packet Level in the X.25 Recommendation." Alta Frequenze 48, No. 8 (August 1979): 339-348.

Bjorner, D. "Finite State Automaton - Definition of Data Communication Line Control Procedures." Proceedings of the Fall Joint Computer Conference. Montvale: AFIPS Press, 1970.

Bochmann, Gregor V. "Logical Verification and Implementation of Protocols." Pp. 8.5-8.20 in Proceedings of the Fourth Data Communication Symposium. New York, N.Y.: IEEE, 1975.

Bochmann, Gregor V.; and Chung, R. J. "A Formalized Specification of HDLC classes of Procedures." Pp. 03A:2.1-03A:2.11 in Proceedings of the National Telecommunications Conference. New York, N.Y.: IEEE, December 1977.

Bochmann, Gregor V. "Finite State Description of Communication Protocols." Computer Networks 2 (October 1978): 362-372.

CCITT. Provisional Recommendation X.25, Interface Between Data Terminal Equipment (DTE) and Data Circuit Terminating Equipment (DCE) for Terminals Operating in the Packet Mode on Public Data Networks. Geneva: Consultive Committee for International Telephone and Telegraph, 1977.

Danthine, A. S.; and Bremer, J. "Modelling and Verification of End-to-End Transport Protocols." Computer Networks 2 (October 1978): 381-395.

Diaz, Michel. "Modelling and Analysis of Communication and Cooperation Protocols Using Petri Net Based Models." Proceedings of the Second International Workshop on Protocol Specification, Testing, and Verification. Idyllwild, CA: n.p., May 1982.

Digital Equipment. Digital Data Communications Message Protocol (DDCMP). DIGITAL NETWORK ARCHITECTURE (DECNET). Maynard, MA: Digital Equipment Corp., March 1978.

Gardner, Robert I. "State of the Implementation of SARA." Pp. 82-83 in Proceedings of the Symposium on Design Automation and Microprocessors. New York, N. Y.: IEEE, February 1977.

Gouda, M. G.; and Manning, E. G. "Protocol Machines: A Concise Format Model and its Automatic Implementation." Pp. 346-350 in Proceedings of the Third International Conference on Computer Communications. Washington, D.C.: International Council for Computer Communication, August 1976.

Green, P. E., Jr. Computer Network Architecture and Protocols. New York, N.Y.: Plenum Press, 1982.

Hailpern, B.; and Owicki, S. "Verifying Network Protocols Using Temporal Logic." Pp. 18-28 in Proceedings of NBS Trends and Applications Symposium. New York, N.Y.: IEEE, 1980.

Harangozo, J. "An Approach to Describe a Link Level Protocol with a Formal Lanauage." Pp. 4.37-4.49 in Proceedings of the Fifth Data Communication Symposium. New York, N.Y.: IEEE, September 1977.

Hewlett-Packard. Condensed Description of the Hewlett-Packard Interface Bus. --hp--part No. 59401-90030. Loveland, Colorado: Hewlett-Packard Co., 1975.

IEEE. IEEE Digital Interface for Programmable Instrumentation, IEEE Standard 488-1978. Long Beach, CA: IEEE, 1978.

Jensen, Kathleen; and Wirth, Nicklaus. PASCAL User Manual and Report. New York, N.Y.: Springer-Verlag, 1978.

Kawashima, H.; Futami, K.; and Kand, S. "Functional Specification of Call Processing by State Transition Diagrams." IEEE Transactions on Communications COM-19 (October 1971): 581-587.

Merlin, P. M.; and Farber, D. J. "Recoverability of Communication Protocols Implications of a Theoretical Study." IEEE Transactions on Communications COM-24 (September 1976): 1036-1043.

Piatkowski, T. F. "Finite-State Architecture." IBM Technical Report TR29.0133. IBM Corp., Research Triangle Park, North Carolina, July 1975.

Piatkowski, T. F. "A Formal Model of the Advanced Data Communication Control Procedures (ADCCP)." A report. Institute for Computer Science and Technology, National Bureau of Standards, Washington, D.C., September 1979.

Piatkowski, T. F. "An Engineering Discipline for Distributed Protocol Systems." Pp. 177-215 in Proceedings of the First International Workshop on Protocol Specification, Testing, and Verification. Middlesex, UK: National Physical Laboratory, May 1981.

Piatkowski, T. F.; Ip, Lap-Kin; and He, Dayun. "STATE ARCHITECTURE NOTATION AND SIMULATION: A Formal Technique for the Specification and Testing of Protocol Systems." Proceedings of the Second International Workshop on Protocol Specification, Testing and Verification. Idyllwild, CA: n.p., May 1982.

Postel, J.; and Farber, D. "Graph Modelling of Computer Communications Protocols." Pp. 66-67 in Proceedings of the Fifth Texas Conference on Computing Systems. Long Beach, CA: IEEE, 1976.

Proceedings of the First International Workshop on Protocol Specification, Testing and Verification. Middlesex, UK: National Physical Laboratory, May 1981.

Proceedings of the Second International Workshop on Protocol Specification, Testing and Verification. Idyllwild, CA: n.p., May 1982.

Razouk, Rami R.; and Estrin, Gerald. "The Graph Model of Behavior Simulator." Pp. 89-98 in Proceedings of the Symposium on Design Automation and Microprocessors. New York, N.Y.: IEEE, February 1977.

Remes, Antero. "Simulation Techniques in Network Design." in Computer Networks and Simulation, pp. 85-100. Edited by S. Schoemaker. Amsterdam: North-Holland Publishing Company, 1978.

Rockstrom, Anders; and Saracco, Roberto. "SDL - CCITT Specification and Description Language." IEEE Transactions on Communications COM-24 (June 1982): 1310-1318.

Schultz, G. D.; Rose, D. B.; West, C. H.; and Gray, J. P. "Executable Description and Validation of SNA." IEEE Transactions on Communications COM-28 (April 1980): 661-677.

SNA. SNA Format and Protocol Reference Manual: Architecture Logic. SC 30-3112. White Plains, N.Y.: IBM Corp., 1976.

SNA. SNA Format and Protocol Reference Manual: Architecture Logic. SC 30-3112-01. White Plains, N.Y.: IBM Corp., 1978.

Stenning, V. N. "A Data Transfer Protocol." Computer Networks 1 (September 1976): 99-110.

Sundstrom, R. J. "Formal Definition of IBM's System Network Architecture." Pp. 03A:1.1-03A:1.7 in Proceedings of the National Telecommunications Conference. New York, N.Y.: IEEE, December 1977.

Sunshine, Carl A. "Survey of Protocol Definition and Verification Techniques." Pp. F1.1-F1.4 in Proceedings of the Computer Network Protocols Symposium. Liege, Belgium: University of Liege, February 1978.

Sunshine, Carl A. Communication Protocol Modelling. Dedham, MA: Artech House Inc., 1981.

Sussenguth, E. "System Network Architecture." Interface '76. Miami, Florida: n.p., 1976.

Symons, F. J. W. "Introduction to Numerical Petri Nets, a General Model of Concurrent Processing Systems." Australian Telecommunication Research 14, No. 1 (1980a): 28-32.

Symons, F. J. W. "The Verification of Communication Protocols using Petri Nets." Australian Telecommunication Research 14, No. 1 (1980b): 34-38.

Teng, Albert Y.; and Liu, Ming T. "A Formal Approach to the Design and Implementation of Network Communication Protocol." Proceedings of the COMPSAC. Long Beach, CA: IEEE, 1978.

Yeh, Jeffy W. "Simulation of Local Computer Networks." Pp. 56-66 in Proceedings of the 4th Conference on Local Computer Networks. New York, N.Y.: IEEE, October 1979.

VII. ACKNOWLEDGMENTS

The author would like to thank his major professors Dr. Thomas F. Piatkowski and Dr. Arthur V. Pohm. In particular, Dr. Piatkowski provided valuable suggestions, guidance, and encouragement throughout the work. The Affiliates Program in Electronics Science and Technology, Iowa State University, provided financial support in the form of a graduate research assistantship.

Thanks are also due to Mr. Dayun He for the discussion of his experience in using SAN and SAS, to Mr. Marcus Jobe for proofreading the dissertation, and to Mrs. Sherry Smay for typing the dissertation.

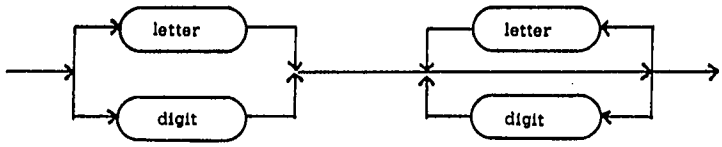
This work is dedicated to the author's wife, Esther, for her love, patience, and support and to the author's parents and brother for always encouraging the author in his study.

Finally, the author thanks God for His abundant blessing.

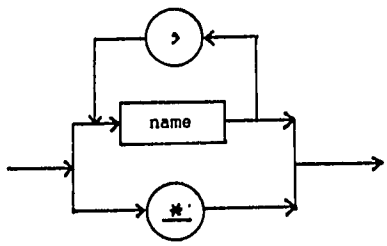
VIII. APPENDIX: STATE ARCHITECTURE NOTATION SYNTAX DIAGRAMS

This appendix summarizes the syntax of State Architecture Notation (SAN) with syntax diagrams in the format commonly used for PASCAL [Jensen 1978]. The elements such as integer and real, which are the same as PASCAL syntax, are not shown in the following syntax diagrams. In case a PASCAL procedure is needed, the user should refer to the syntax in the PASCAL User Manual [Jensen 1978]. The syntax diagrams do not reflect a direct implementation of the SAS syntax checking process; however, the syntax diagrams do provide a correct format for the users to specify their SAN models. Users are also urged to follow the syntax diagrams to specify their SAN model.

name

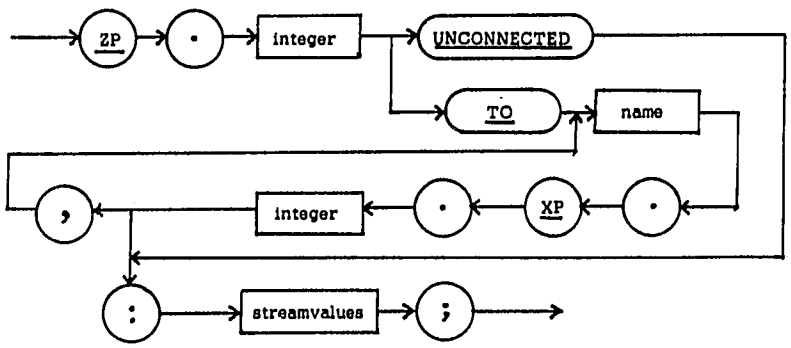


streamvalues

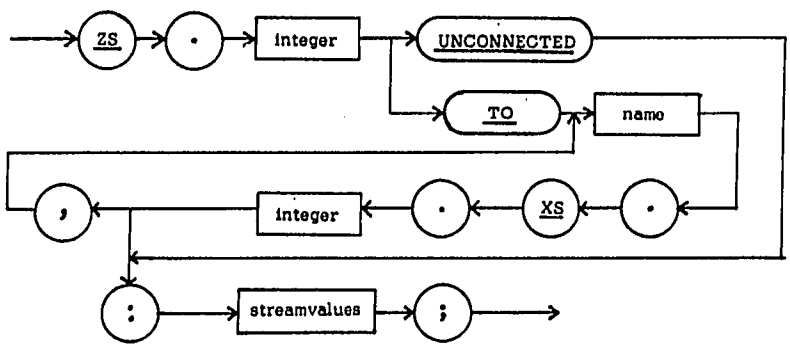


Sline

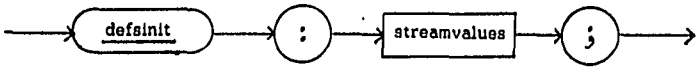
ZPline



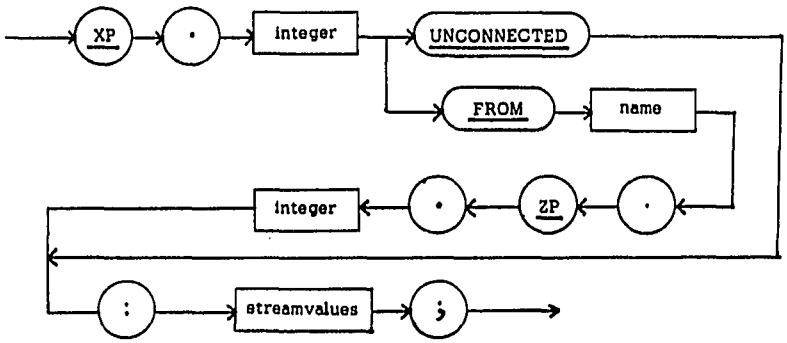
ZSline



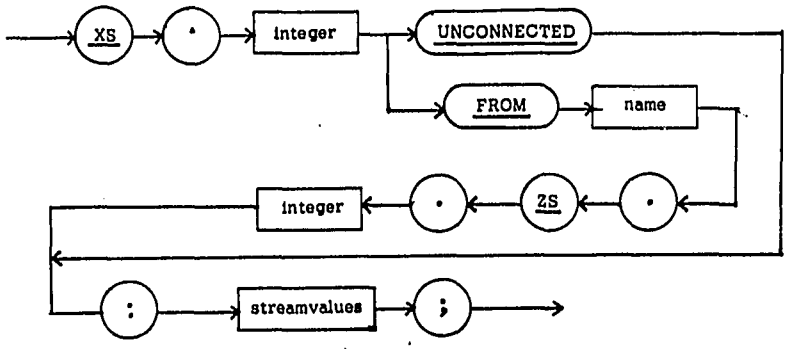
defsinitline



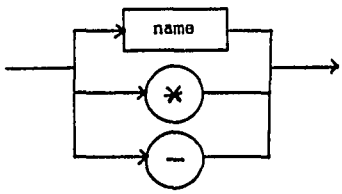
XPline



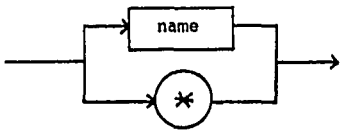
XSline



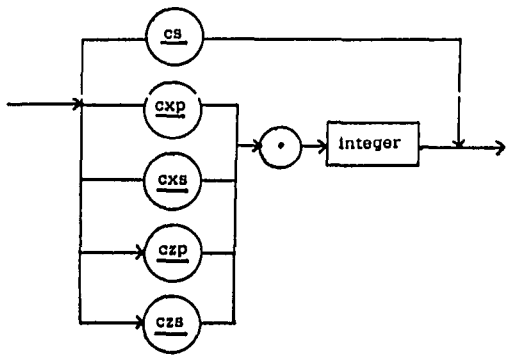
xp



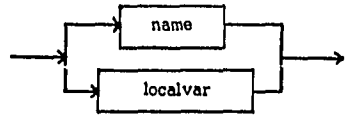
xs



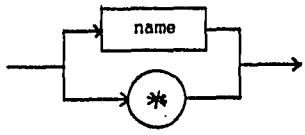
localvar



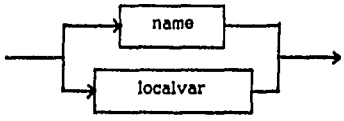
ntzs



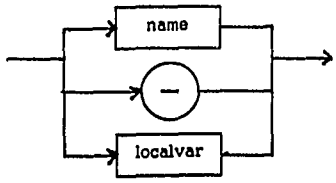
cs



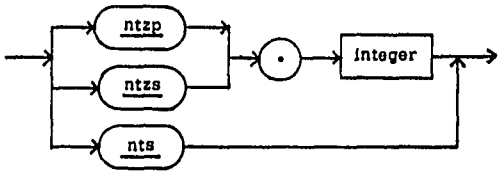
nts



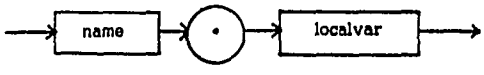
ntzp



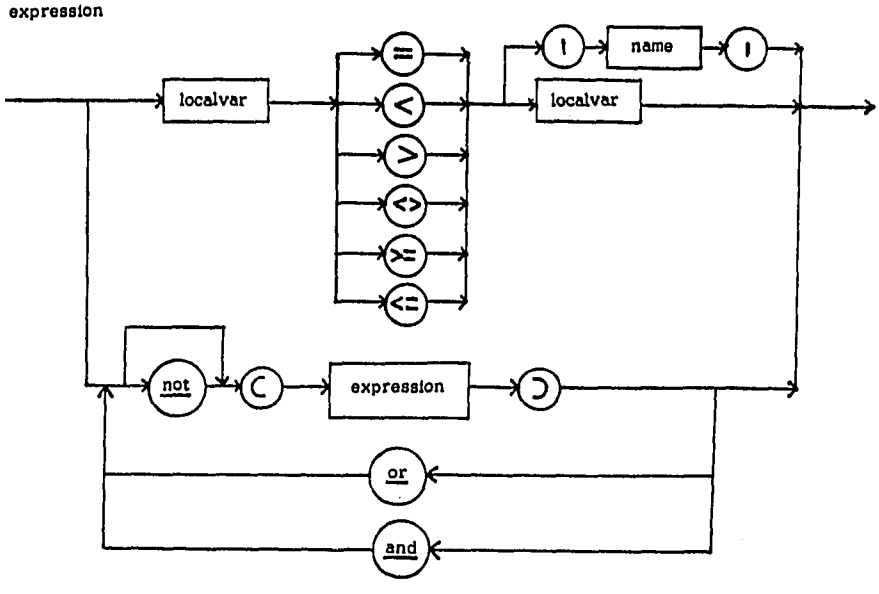
updatevar



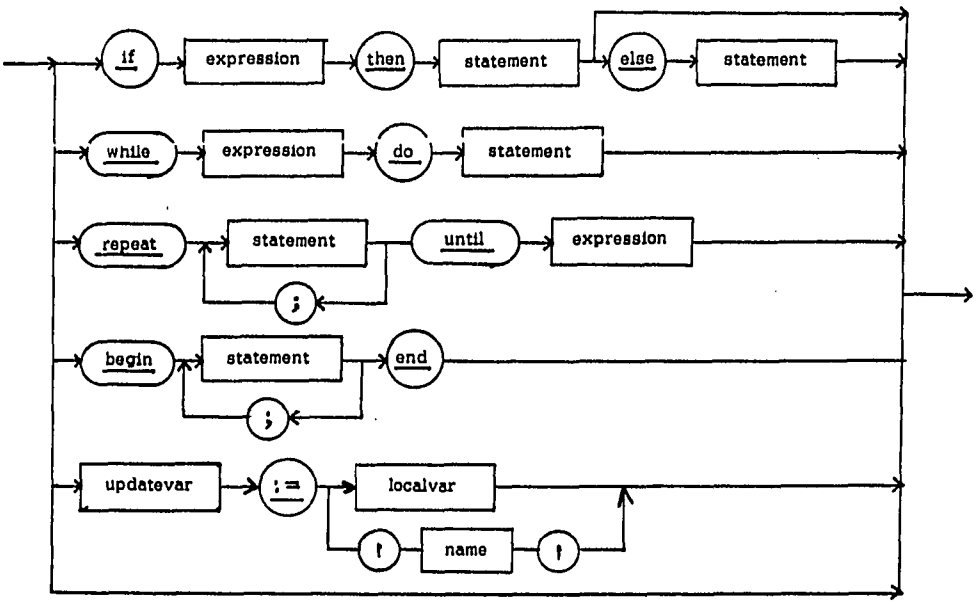
gobalvar



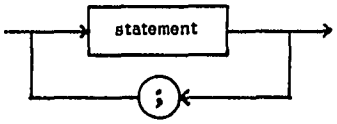
expression



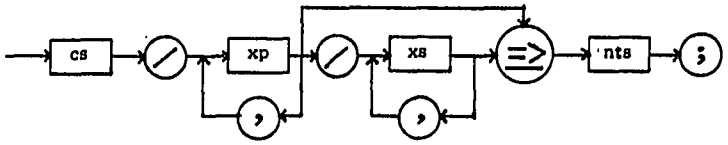
statement



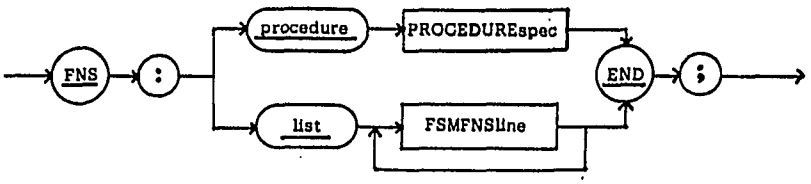
PROCEDUREspec



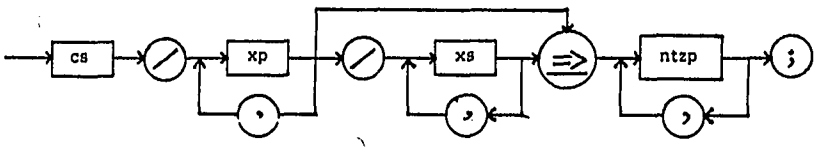
FSMFNSline



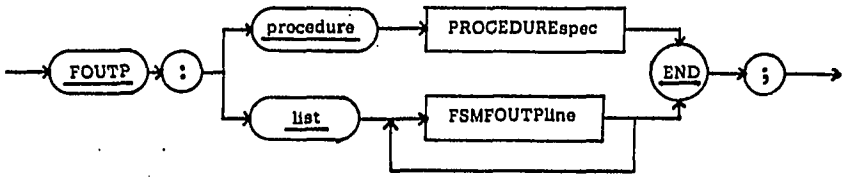
FSMFNSspec



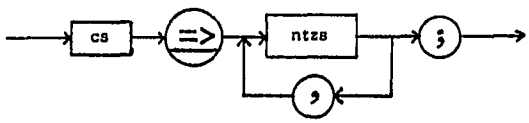
FSMFOUTPline



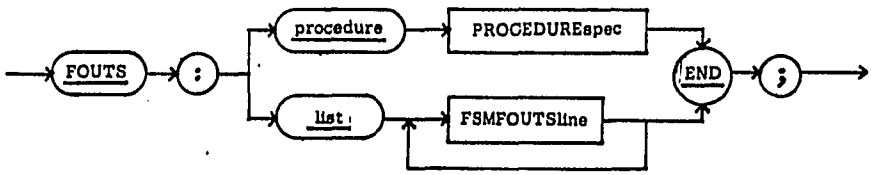
FSMFOUTPspec



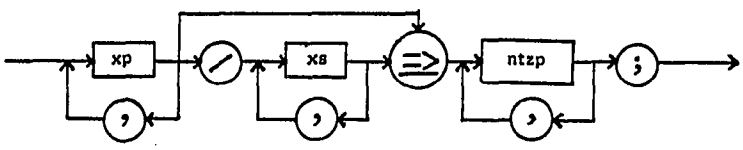
FSMFOUTSline



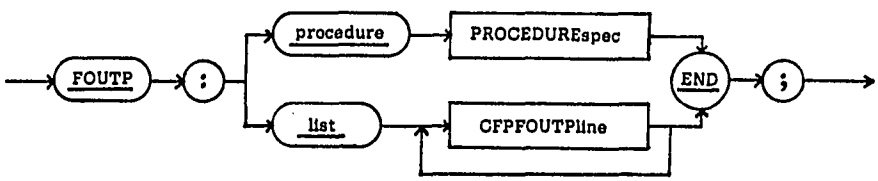
FSMFOUTSspec



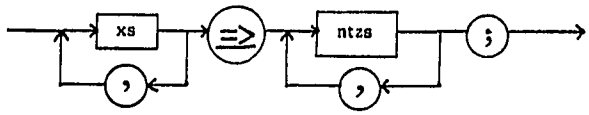
CFPFOUTPline



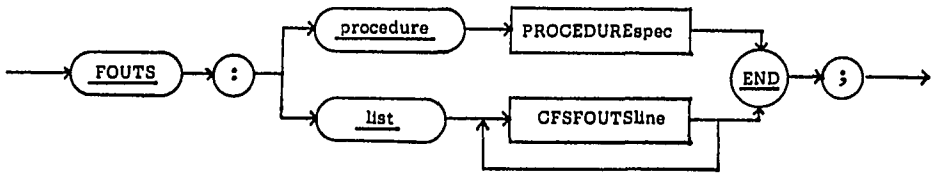
CFPFOUTPspec



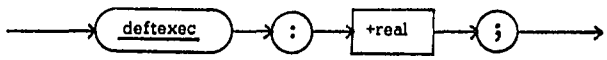
CFSFOUTSline



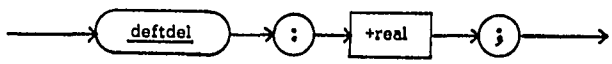
CFSFOUTSspec



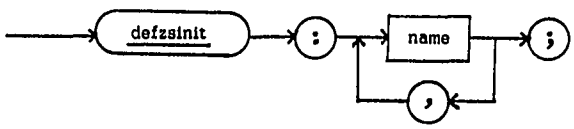
deftexecline



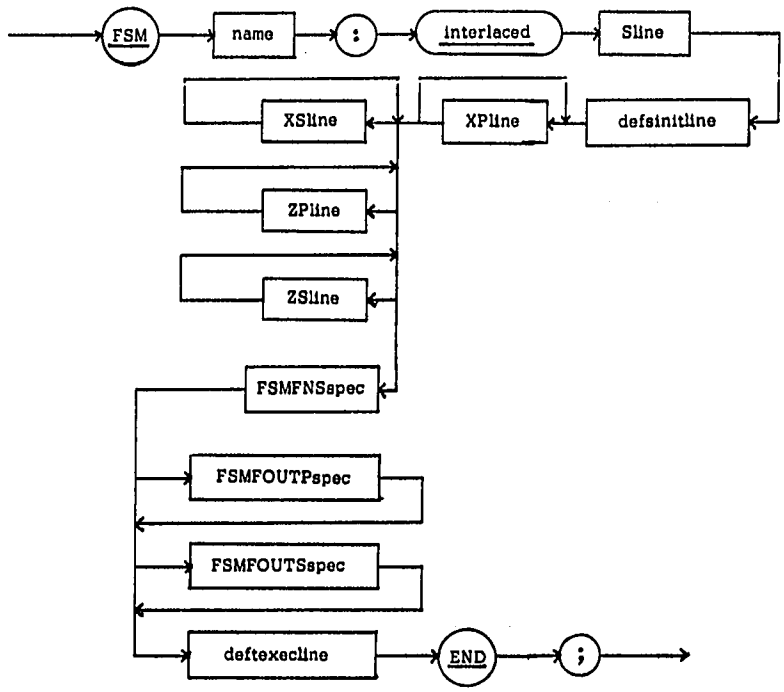
deftdelline



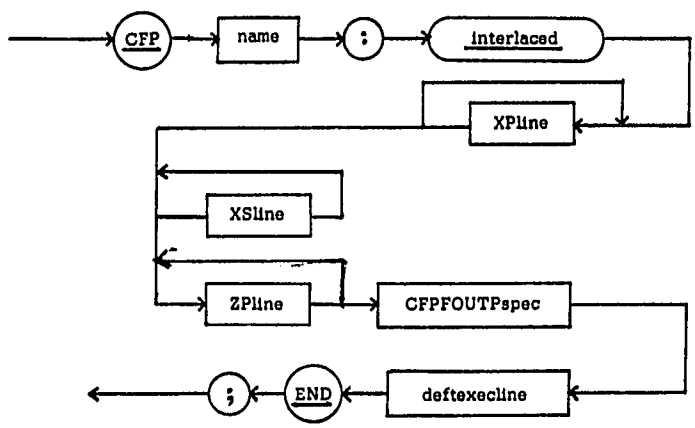
defzsinitline



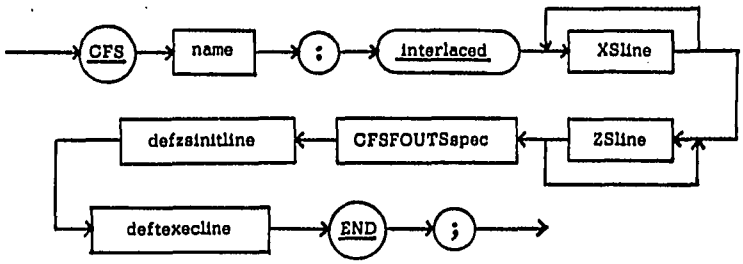
FSMspec



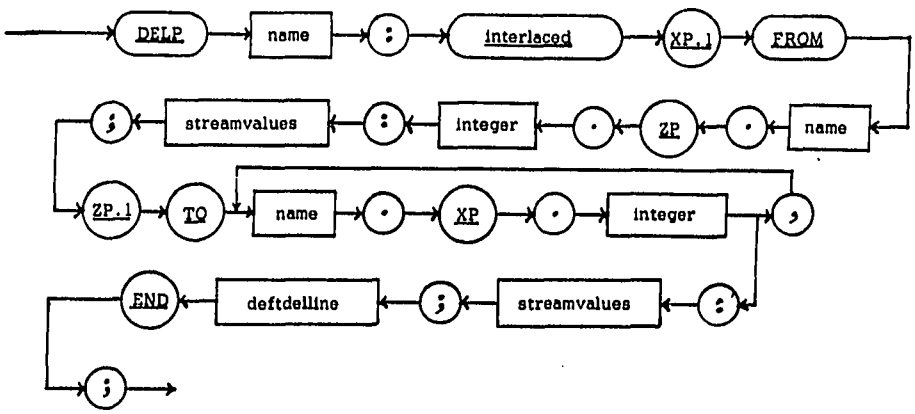
CFPspec



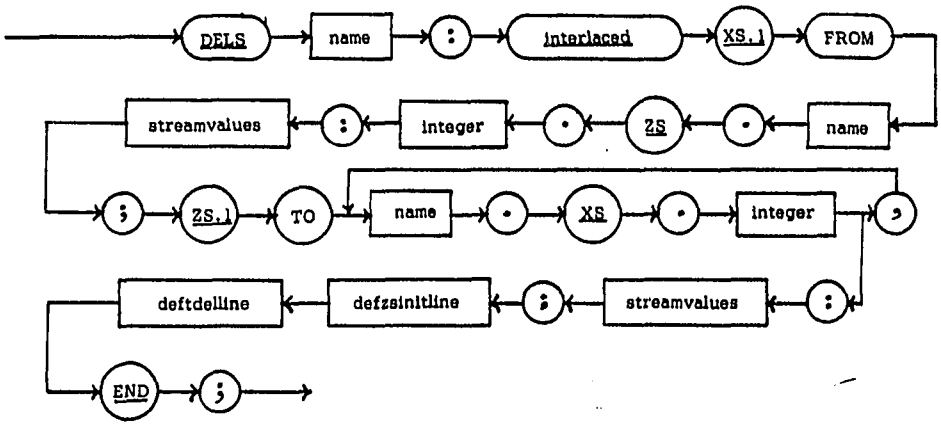
CFSspec



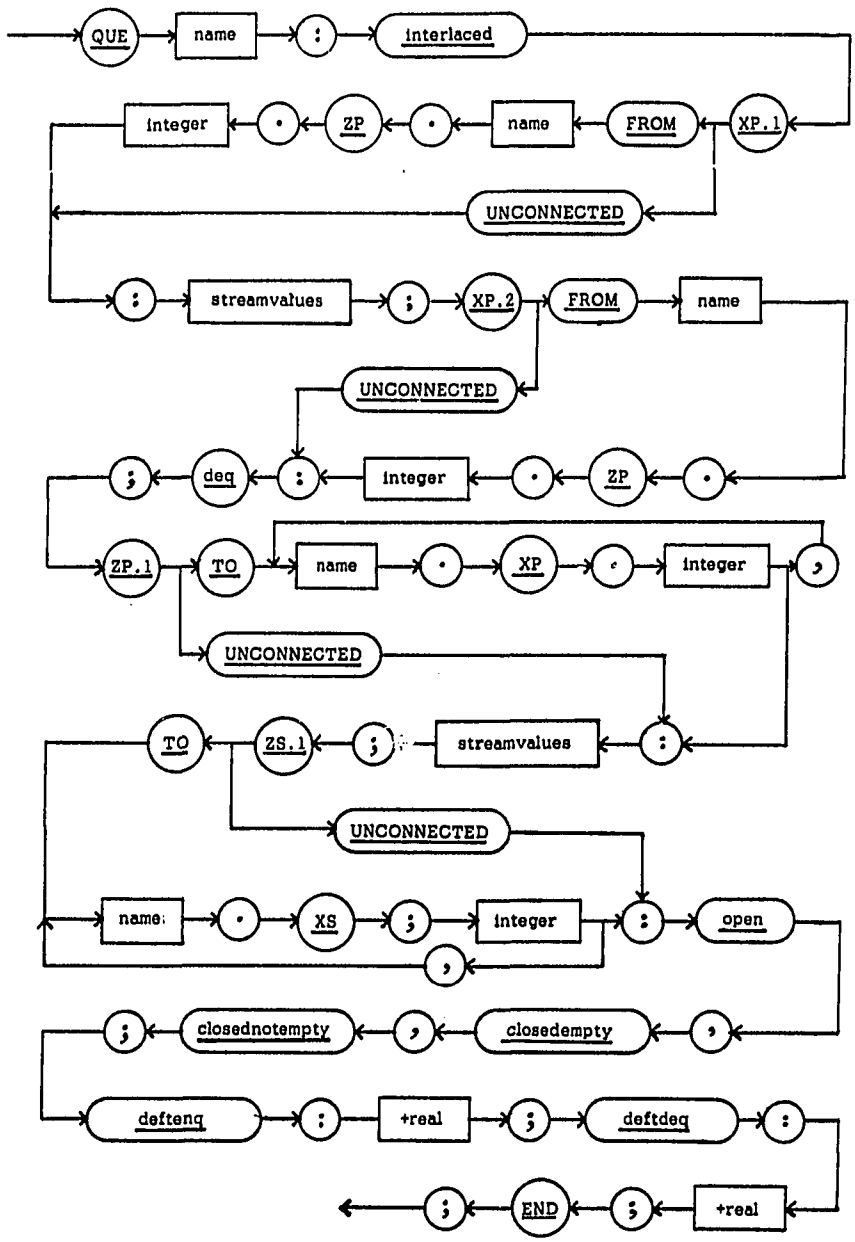
DELPspec



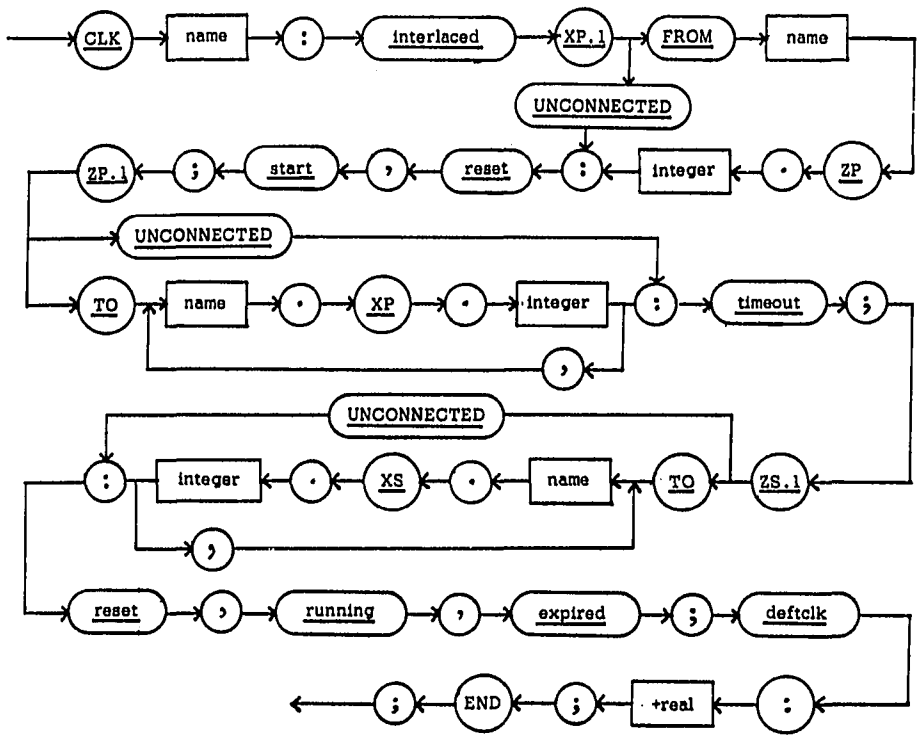
DELSspec



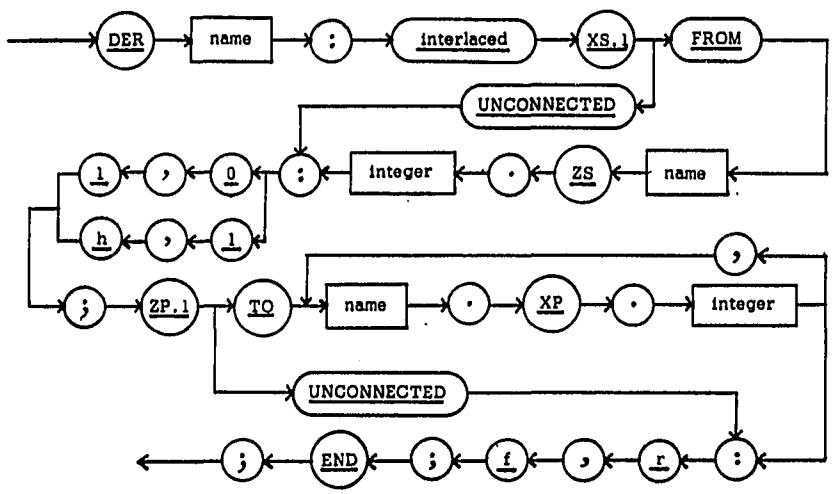
QUEspec

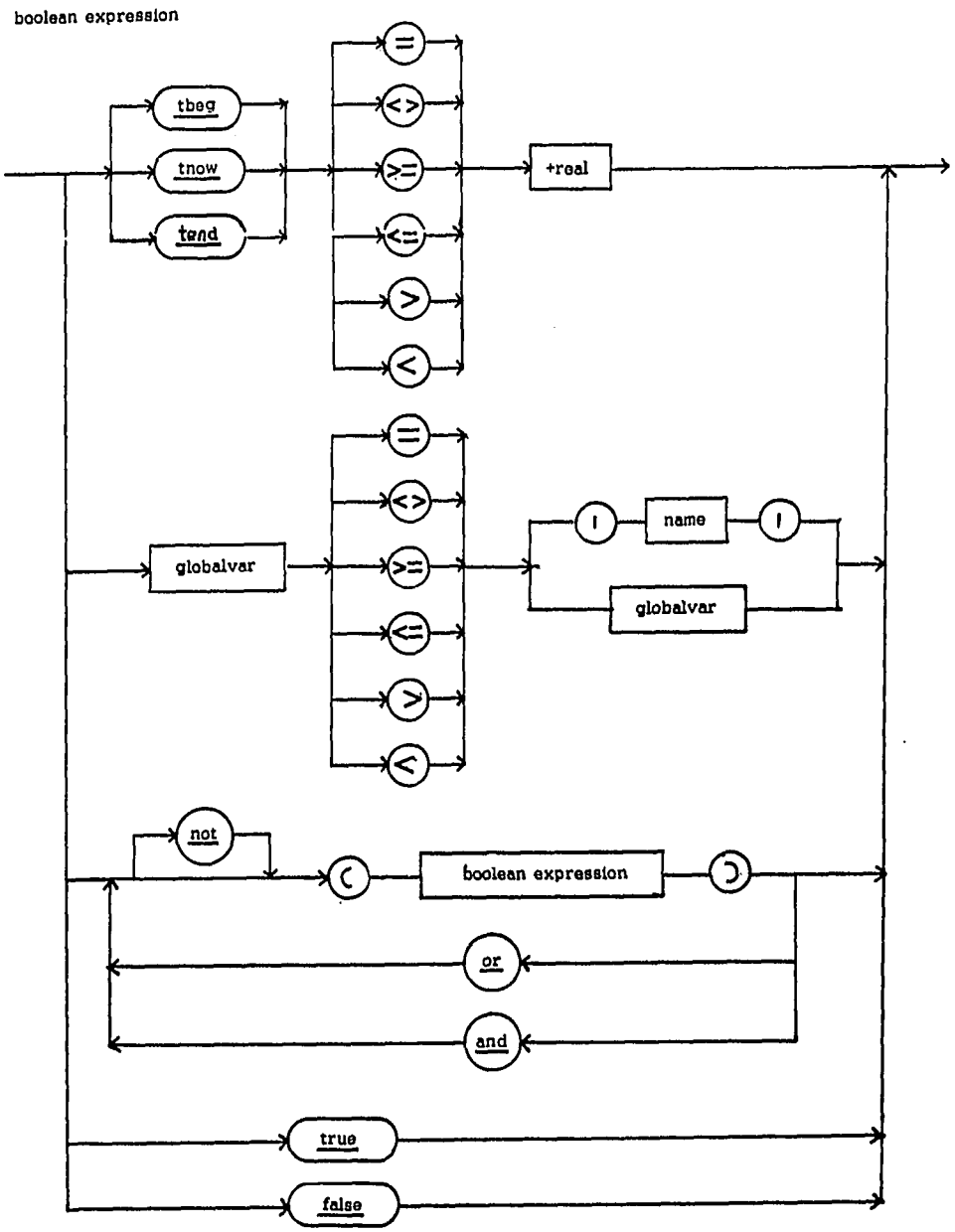

QUE
name
:
interlaced
integer
.
ZP
.
name
FROM
XP.1
UNCONNECTED
:
streamvalues
;
XP.2
FROM
name
UNCONNECTED
;
deq
:
integer
.
ZP
.
ZP.1
TO
name
.
XP
.
integer
,
UNCONNECTED
TO
ZS.1
;
streamvalues
:
UNCONNECTED
name:
.
XS
;
integer
:
open
,
;
closednotempty
,
closedempty
,
deftenq
:
+real
;
deftdeq
:
;
END
;
+real

CLKspec

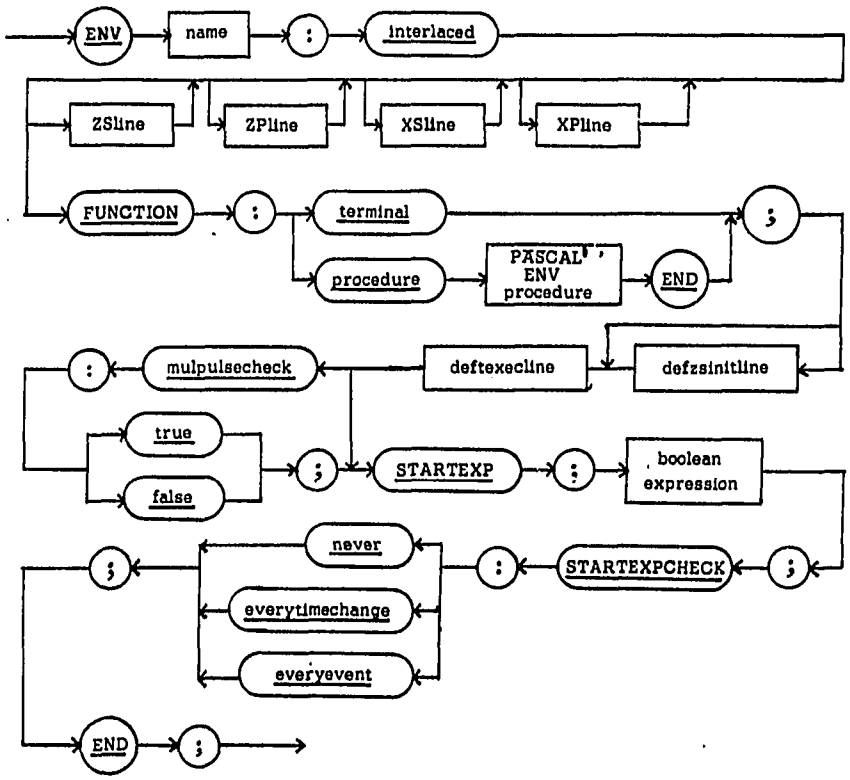


DERspec

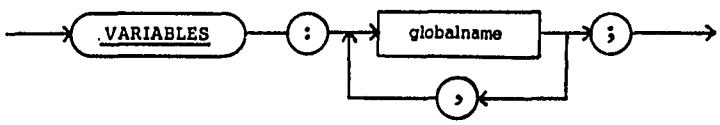



boolean expression
tbeg
tnow
tend
=
<>
>=
<=
>
<
+real
globalvar
=
<>
>=
<=
>
<
(
name
)
globalvar
not
(
boolean expression
)
or
and
true
false

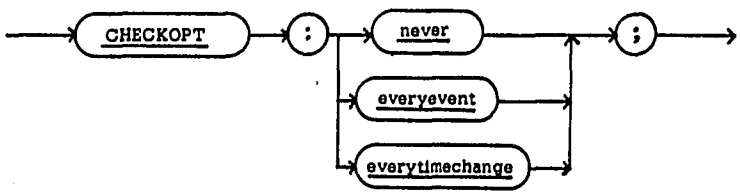
ENVspec



VARIABLESline



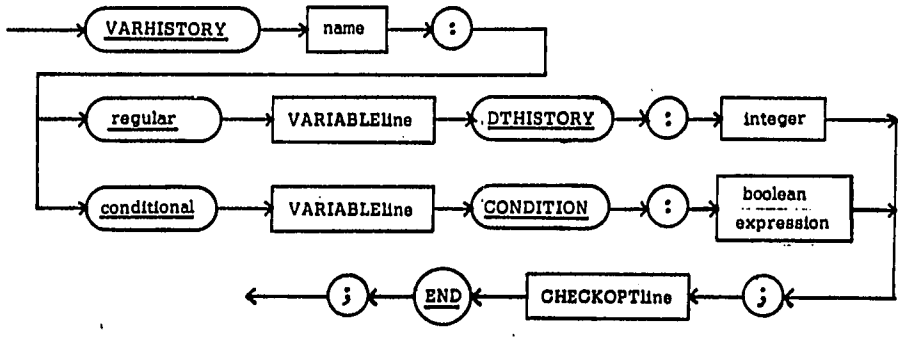
CHECKOPTline



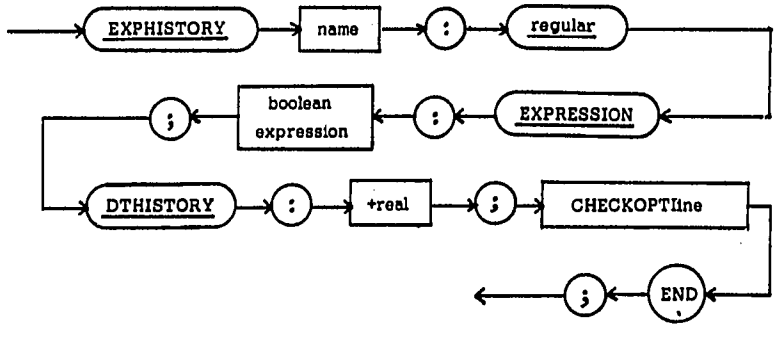
---

[1] A PASCAL ENV procedure as described in Chapter II.J

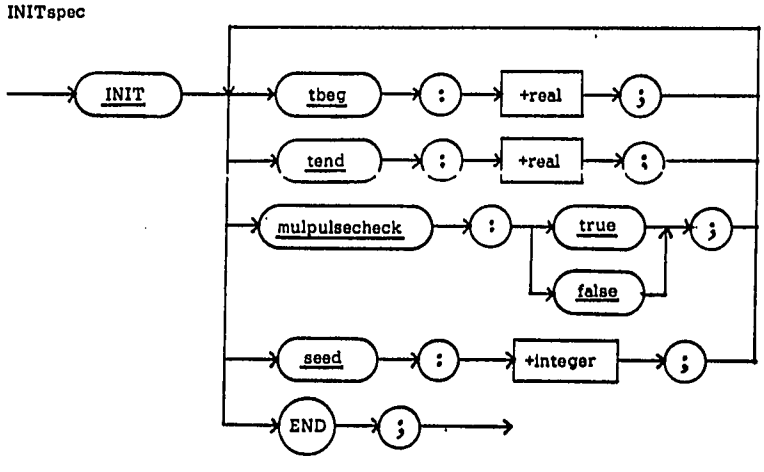
VARHISTspec



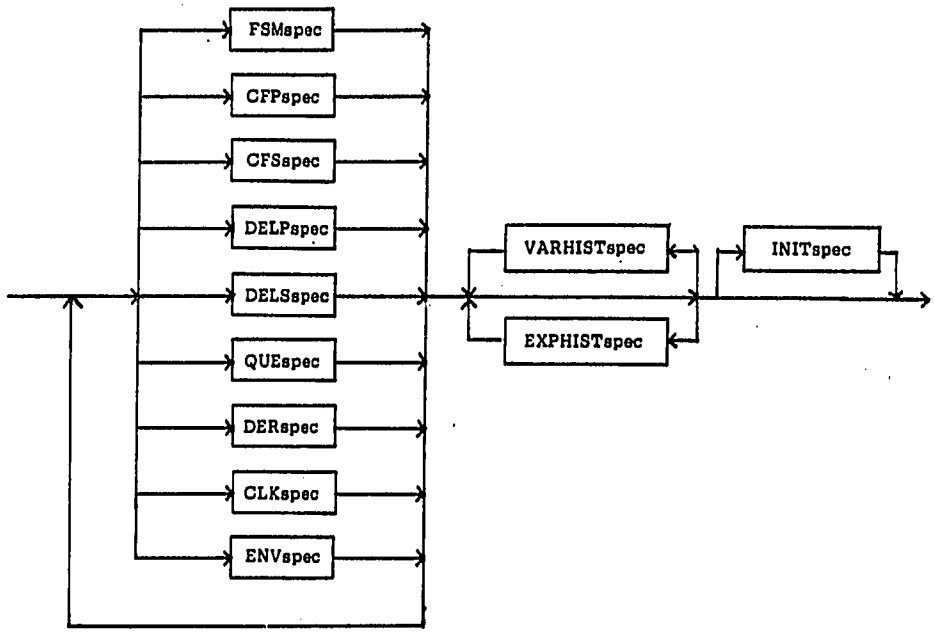
EXPHISTspec



INITspec



SANmodel


FSMspec
CFPspec
CFSspec
DELPspec
DELSspec
QUEspec
DERspec
CLKspec
ENVspec
VARHISTspec
EXPHISTspec
INITspec