NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

The SLIDE Simulator: A Design and Evaluation Tool For I/O and Interfacing Strategies

by

Arthur H. Altman

DRC-01-3-80

December, 1979

The SLIDE Simulator : A Design and Evaluation Tool For I/O and Interfacing Strategies

Arthur H. Altman
Department of Electrical Engineering
Carnegie-Mellon University
Pittsburgh, Pa. 15213

Master's Research Project Report

December 2, 197S

This research has been supported by the U.S. Army Research Office, **under grants** «DAAG29-76-G-0221, «DAAG29-78-G-0070, and »DAAG29-79-C-O197, **and** by **the** Department of Electrical Engineering, Carnegie-Mellon University.

UNIVERSITY LIBRARIES
;.VLV;.:-;SA 15213

Acknowledgements

There are many people who in one way or another aided and abetted the author in making this project a success. My advisor, Professor Alice Parker, was an inexhaustable source of support, comments and criticism, not to mention premature terminations of employment ; she has had a profound influence on my outlook as an engineer. As for the other members of my committee, Dr. Mario Barbacci and Dr. Don Thomas, their doors were never closed, and they have provided useful suggestions and encouragement during the past year and a quarter. Rich Bollingcr produced the preprocessor software, and suffered through countless bull sessions helping this hardware type crawl through the software maze. John Wallace and Karem Sakallah provided valuable information and insight into, respectively, the SLIDE language and simulation in general. Finally, I thank my friends and my family in Montreal, Pittsburgh, and elsewhere, for their love and support - for making it worth doing.

Table of Contents

1. Introduction 1

2. Motivation and Background 4

- 2.1 The Need For SLIDE Simulation 4
- 2.2 Inclusion Of the Mulli-Level Simulator 5
- 2.3 A Hardware Simulator Taxonomy 6
- 2.4 Discussion of SARA and SABLE **10**

3. SLIDE Simulator : Implementation and Executing Environment 13

- 3.1 Implementation 13
- 3.2 The Multi-Level Simulator 16
  - 3.2.1 Requirements for Multi-Level Simulation 17
  - 3.2.2 SIMULA 17
  - 3.2.3 Data and Control Structures of the Simulator Core 18
  - 3.2.4 The Module Interconnections 21
- 3.3 SLIDE Simulation Overview 23
  - 3.3.1 Overall Relationship to the Core 23
  - 3.3.2 SLIDE Device Functioning During Simulation **24**

4. Mapping SLIDE to SIMULA 26

- 4.1 A Review of SIMULA 26
- 4.2 SLIDE Device Implementation 27
- 4.3 SLIDE Processes 27
  - 4.3.1 Issues 27
  - 4.3.2 Implementation 28
- 4.4 Inside The SLIDE Process 34
  - 4.4.1 Hardwnro 34
  - 4.4.2 Subroutines 37
  - 4.4.3 Statements 38
  - 4.4.4 Timing 41
  - 4.4.5 Expressions **42**

5. SLIDE Code Generator 47

- 5.1 Context Of Operation 47
  - 5.1.1 Pre-Compiled Code 47
  - 5.1.2 Preprocessor 47
  - 5.1.3 Final Simulator Program 49
- 5.2 Discussion of Operation 51
  - 5.2.1 Uniqueness of Variable Names 51
  - 5.2.2 Procedure Interaction and Related Comments 51
- 5.3 Points Of Interest 54

6. The SLIDE Simulator Tost Case 58

- 6.1 Peripheral Dovico Description 58
- 6.2 Summary Of Simulation Test Results 61

7. Conclusions **63**

Reforences 67

I. Simulator Commands 70

II. Test Traces And Commentary 73

List of Figures

Figure 1: Simulator Structure 14
Figure 2: Producing a Runnable SLIDE Simulator 15
Figure 3: Partitioning of a Digital Device in the Multi-level Simulator 19
Figure 4: Representation of an Inverter Module 20
Figure 5: Representation of Device Interconnection by Wire 22
Figure 6: Static Data Structures of SLIDE Processes 30
Figure 7: Structures of the SLIDE Scheduler 31
Figure 8: SLIDE Expression Implementation 44
Figure 9: Procedure Interaction In The Code Generator 53
Figure 10: SLIDE Simulator Test Case 59
Figure 11: SLIDE Process Structure of the Peripheral Device Model 60
Figure I-1: Results of example ADD commands 71
Figure II-1: Daisychaining of UNIBUS NPG Line 74
Figure II-2: First Test Run : Trace 1 76
Figure II-3: First Test Run : Trace 2 77
Figure II-4: First Test Run : Trace 3a 78
Figure II-5: First Test Run : Trace 3b 79
Figure II-6: First Test Run : Trace 3c 80
Figure II-7: First Test Run : Trace 3d 81
Figure II-8: First Test Run : Trace 3e 82
Figure II-9: Second Test Run : Trace 1 83
Figure II-10: Third Test Run : Trace 1 84
Figure II-11: Third Test Run : Trace 2 85

1. Introduction

With the advent of LSI technology digital system designers have had to search for structured approaches to the design task, because the cost of using traditional design techniques for LSI-based systems has become prohibitive. These highly complex designs are simply not amenable to the last minute patch-up, the ambiguous documentation, and **the** premature attention to detail that characterised the "good old days[11] of a thousand or so transistors on a chip.

The notion Of top-down digital system design [6] has been promulgated by various engineers for a number of years as the best alternative to *ad hoc* design. The main thrust of this approach is the orderly progression from a high-level design specification for **a** system to an equivalent low-level design specification, using computer simulation of designs in moving from one level of abstraction to another. References [6] and [5] describe this approach, in which the design specification of a system and the simulation model of that system are in fact equivalent. What should be remembered is that

1. this design methodology implies that digital design specifications must be *formally* described at a number of levels, and that

2. each level of abstraction to be dealt with directly by the designer needs a simulation capability at that level for the designer to use.

In light of the above, it is not surprising that formal descriptive systems (and their associated simulators) for digital hardv/are have been fruitful and multiplied, so to speak. What is surprising is the lack of attention that has been paid to the area of interfacing and I/O hardware. After all, it is clear that interconnection design has a significant effect on the price and performance of complex digital systems such as multiprocessors. Nonetheless, until recently no complete register-transfer level hardware descriptive language existed to address the nocd to describe, for example, the behaviour of peripheral devices on an I/O bus. Currently existing general purpose hardware descriptive languages , as well as past I/O hardware descriptive proposals, have been found unsuitable in one way or another for describing complicated interconnection strategics [22]. With current simulation techniques, one is limited to goto and circuit- level simulations of I/O hardware, unless one is willing to construct a simulation program from a programming language such as APL or a simulation language such as SIMULA. An approach which allows a cleaner user interface is to compile from a hardware descriptive language to a programming language for simulation purposes.

It is evident that a real gap has existed in the set of hardware descriptive languages. SLIDE, a Structured Language for Interface Design and Evaluation, has been developed at Carnegie-Mellon to serve as a stand-alone behavioural description language for I/O **and**

interfacing strategies *[22].* It is intended Unit SLIDE will serve as a specification, verification, and simulation vehicle within the larger context of the design automation effort at CMU, filling the I/O descriptive gap.

Not surprisingly in the context of this discussion, the most immediate technical task arising from the existence of SLIDE was simulation. The goal of this Master's Research Project was to provide, as a first application of SLIDE, an I/O and interfacing hardware simulation facility for the CMU DA community, subject to these constraints :

- SIMULA-67 [1] , because of its rich repertoire of discrete event simulation primitives, was the programming language of choice.

- SLIDE simulations wore to be incorporated into a developing multi-level simulation facility.

It should bo noted that these two problem constraints came about at different points in time. It was indeed decided a *priori* to simulate SLIDE using SIMULA-67 as the simulation vehicle. However, the other main constraint to the problem , to embed SLIDE device descriptions in a "multi-level[11] simulator [8], was only introduced in the spring term. This point is addressed in the next chapter.

The above task was simplified by the existence of a SLIDE compiler [23]. This compiler does a syntactic check on SLIDE descriptions and produces a Global Data Base parse tree similar to that used by ISPS [2], [3]. Even so, the programming problem as stated gave rise to these sub-problems :

- A suitable mapping from SLIDE to SIMULA had to be devised such that the resulting code would be compatible with the multi-level simulator.

- A code generation program had to be written that would produce the required mappings.

- Software had to be designed that v/ould glue together the translated SLIDE descriptions and the multi-level simulator.

The purpose of this report is to describe the nature of the problems encountered , and the solutions produced, during the past 15 months of this project.[1] Chapter 2 provides some motivation and background related to the problem of SLIDE simulation. Chapter 3 discusses the implementation and executing environment of the SLIDE simulator. Chapter 4 describes

---

[1] Throughout this report, a certain clogroo of familiarity will bo assumed on the reador'o part with SLIDE. Reference [22] IB a good introduction to tho language.

the way SLIDE w.r; mapped into SIMULA, and Chapter 5 illustrates the code generator program that produces these mappings. Finally, an example test case of the SLIDE simulator is presented in Chapter 6, and the report is completed with conclusions in Chapter 7.

2. Motivation and Background

The SLIDE simulator is a tool that will answer a number of questions a systems designer might raise with respect to an interface or interconnection strategy. These questions include:

- Does the design function as planned
- Is the design sensitive to wirelengths and other timing dependencies
- What is the effect of occurrence of exception conditions on operation of the proposed design
- What are the bottlenecks which limit speed of operation of the design
- What is the average device latency and is the maximum allowable latency exceeded
- Is resource allocation with the proposed strategy vulnerable to starvation and/or deadlock problems, and does it add an "acceptable" amount of overhead or not

It should be noted here that some properties of interconnection strategies (deadlocks, for example) might only be discovered through formal verification procedures, such as that proposed by Wallace *[24].* However, simulation does provide a design aid which, if properly constructed and used, can provide a designer significant assistance in answering the above questions.

The purpose of this chapter is to elaborate on the research problem that was solved by this project, and on why this problem was unique in light of previous work. As the first step in doing this some clarification is provided in the next section as to the motives that were served in this research.

2.1 The Need For SLIDE Simulation

To the reader familiar with the present body of work in digital hardware simulation, it may seem unnecessary to have built this SLIDE simulation package. After all, behavioural descriptions of I/O hardware at higher levels than SLIDE have been written and simulated using special purpose programming languages such as ASPOL [5]; the simulation problem is "already solved". The answer to this is two-fold. First, production of an I/O hardware simulator using cither special purpose or general purpose programming languages is not a trivial task. It requires the user to manually produce code to handle the I/O primitives and related semantics which SLIDE already describes and which the SLIDE simulator will automatically translate and execute. So the distance from conception to simulation is considerably shortened with the SLIDE simulator. Second, SLIDE is being proposed as a formal descriptive and formal verification tool as well as a descriptive language for a

simulator. In l his connection, the development of SLIDE itself as a useful new hardware descriptive language v/ili be enhanced greatly if "real world" user feedback on the language can bo encouraged. The quality and quantity of such feedback from the users of a SLIDE simulator will almost certainly be superior to that from the users of a sterile paper-and-pencil language. It is clear that the simulation of SLIDE, then, was a research problem that was indeed worth pursuing.

2.2 Inclusion Of the Multi-Level Simulator

Originally, it was thought that a dedicated simulator for SLIDE would be built along the lines of the ISPS simulator at CMU [3]. Thus it would accept and simulate one SLIDE description at a time. SLIDE nesting capabilities would be used to to describe inter-device structuring as well as intr.i-device structuring. However, it was subsequently decided that SLIDE descriptions of devices would instead be translated into functional modules for a multi-level simulator [8] which was being constructed at the same time. This multi-level simulator is designed to allow a ur.cr to simulate digital systems whose components may be gate, register-transfer, or system level modules. It has as its heart a uniform interconnection and representation mechanism for digital devices , and an interactive command language to allow the user to build tost configurations from a library of device modules . From a simple conceptual viewpoint, adding SLIDE to this simulator meant allowing SLIDE-described devices to be added to the library.

There are a number of advantages to the above decision. For instance, one must recognise that a given interconnection strategy (i.e., protocol, resource allocation, addressing , etc.) assumes one or more topologies for device interconnection, e.g. star, loop. Since a particular topology can have a wide variety of instantiations, it is desirable (even imperative) for the designer of the interconnection strategy to test out that strategy using as many different configurations of devices as possible. Having to rewrite the SLIDE description for each such test would be inconvenient and slow. As it is, the SLIDE nesting constructs do not convey topological information as well as they do behavioural information. If the SLIDE description of a device could be fixed for *each* interconnection scheme, i.e., if the SLIDE nesting constructs were reserved for *internal* device structuring, which is less variable than inter-device connections, then once the member components of an interconnection strategy had been specified in SLIDE , they would not need to be recompiled while various system configurations were being tested out. The multi-level simulation environment provides exactly such a capability.

Another advantage is increased utility for SLIDE as a design tool. This comes as a natural by-product of being in a multi-level simulation environment, because general digital

components, not just their interfaces, can be interconnected and simulated at various levels of detail.[1] Thus, more general tests of interconnection schemes can be provided than simply those involving hooking up SLIDE-described devices to each other; for example a SLIDE I/O interface for a CPU could be connected to a high level functional description of the CPU, which could in turn be interfaced to some gate-level device structure. This capability would not have existed in a dedicated SLIDE simulator.

The principal disadvantage was in terms of project logistics. Because the implementation of the multi-level simulator was already well underway when the decision to incorporate SLIDE was made, this author was faced with a software *fait accompli.* This is one of the reasons why the project was especially challenging : SLIDE simulation had to be designed to conform to the multi-level simulator structure[2] .

In summary, it can be said that the sum of the following points made this project "unique" :

- A new and different HDL was to be simulated

- It was to be set inside a multi-level simulator

Additionally, due to project timing problems, the burden of software compatibility was unidirectional, which made the project task more difficult than originally anticipated.

Before moving on to discuss the SLIDE simulator itself, it may be helpful to try and quantify the relationship this simulator has to existing hardware simulators. In the next section, an informal taxonomy for digital hardware simulators is presented, and it is used as a framework for comparison of some currently existing digital hardware simulators.

2.3 A Hardware Simulator Taxonomy

The following taxonomy for digital hardware simulators was developed during informal discussions with some members of the CMU DA group. No claims are made for it other than as a simple basis for discussion of the hardware simulator "space". This taxonomy is based upon eight design decisions for this class of software. These design decisions are not uncorrelated, and it is debatable whether a "complete" set of uncorrelated decisions exists at

[1] References [1] and [20] discuss the advantages of being ablo to go from high **level,** low detail **functional descriptionB to** low level, high rlolail structural descriptions within one simulation as **A way of gotting around the drawbacks of purely** high level / purely low lovel simulation.

[2] Some software extensions **woro** made to **the** multi-lovel simulator **after the fact , however.**

all for digital hardware simulator design. It should be noted that the set of all possible outcomes of these decisions does not constitute a total ordering on the set of possible simulators.

The eight design decisions are :

1. Types of descriptive languages
   a. Abstract Models (e.g. Petri net)
   b. Hardware Descriptive Languages
   c. Special Purpose Programming Languages (e.g. GPSS)
   d. General Purpose Programming Languages
   e. General Purpose Programming Languages extended for hardware description (e.g. AHPL [15])
2. Types of levels of abstraction
   a. Circuit Level
   b. Gate Level
   c. Register Transfer Level
   d. System Lovel
3. Number of Levels of Abstraction / Number of Languages
   a. 1 / 1
   b. >1 / 1
   c. >1 / >1
4. Characterisation of Behaviour
   a. Stochastic
   b. Deterministic
5. Characterisation of Structure
   a. Formal (e.g. SL/1 [10], SDL [21])
   b. Informal (i.e. interactive commando)

6. Degree of Separation of Structure from Behaviour

    a. None (e.g. CAP [18])

    b. Weak (most simulators)

    c. Strong (e.g. SARA [9])

7. Binding of Structure and/or Behaviour

    a. Before Simulation (batch)

    b. During Simulation (interactive)

8. Characterisation of Operation

    a. Orientation

        i. Event Oriented

        ii. Activity Oriented

        iii. Process Oriented

    b. Execution

        i. Table-driven

        ii. Interpretive

        iii. Compiled

Each of these design decisions will now be briefly described and discussed.

"Types of descriptive languages" refers to the category of structural and/or behavioural descriptive vehicle that the user provides to the simulator as input. It is hoped that the definitions of the five listed categories are intuitively obvious.

"Types of levels of abstraction" relates to the choice of descriptive focus that the simulator will give to the user. The four generally agreed upon levels of abstraction are enumerated ; *gain it is expected that the definitions are well Known lo the reader.

"Number of Levels of Abstraction / Number of Languages" refers to the intertwined decisions about, first, whether the simulator will support modelling of components at different levels of abstraction, and second, the number of descriptive languages that will be used to

describe that modelling. If one chose single-level modelling, it would imply a single descriptive language. But multi-level modelling could be supported by a single descriptive language such as ADLIB, as well as by assigning one language per level of abstraction.

"Characterisation of Behaviour" attempts to make a distinction between the probablistic style of behavioural description (e.g. data arrives at a terminal in an exponentially distributed fashion) and the more common deterministic style, wherein the actions of components are described (to at least some degree) so that their behaviour at a given point in time may be precisely known.

"Characterisation of Structure" refers to the dichotomy between methods of structure description in simulators. On the one hand, structure or topology may be described in an informal, interactive fashion by the user. On the other hand, some simulators insist on the use of formal methods to describe system structuring.

"Degree of Separation of Structure from Behaviour" refers to the emphasis that is placed on the uses of the available descriptive languages. By extension, this also refers to the degree to which separation of structure from behaviour is promoted by the simulator itself. Some simulators want to separate the use of languages along strict structural / behavioural lines ; others want a single language to do everything. Most simulators , though, allow their descriptive languages to cross the boundaries to some degree.

"Binding of Structure and/or Behaviour" is a design decision that relates to the point in time when a user finalises the description of the system to be simulated. In a "batch" simulator this binding must be done before simulation is to begin, and is fixed thereafter. In an "interactive" simulator the binding can be modified after simulation has begun.

"Characterisation of Operation" is broken down into two sub-decisions. The first is orientation, which describes the overall outlook of the simulator as it executes. The second is execution, which describes the method by which the actual running simulation is achieved.

The three orientation categories are event , activity, and process oriented[1] . These categories grow out of the notion of "discrete event simulation", whereby the work that is done by the model being simulated is broken down into discrete units of work, and each of these work units has a certain execution time associated with it. An *activity* is the chosen fundamental unit of work in a simulation for a given viewpoint or level of abstraction. Whether or not it can be further dissected is of little concern ; it is *selected* to be a single

---

[1]These categories were derived from the discussion of simulation languages found in reference [16].

work step for the particular level of abstraction. A *process* is an ordered collection of activities, and like activities, processes arc discrete entities that occur dynamically. An *event* is an instantaneous change of system state that can cauoc activities or processes to initiate or suspend execution. One can say that a simulating system is characterised by the dynamic interactions of processes, and these interactions are governed by the occurrence of events. *Activity oriented* and *process oriented* simulators are those that are concerned with the dynamic scheduling and rescheduling of activities and processes, respectively. The viewpoint in process oriented simulators (and to a lesser degree activity oriented simulators) is biased in a vertical, multi-level direction, in that a given activity can be expanded into a process at a lower level of abstraction, and of course a process can be expanded into one or more activities. In contrast, *event driven* simulators are concerned with the scheduling of events. The outlook in an event oriented simulator is horizontal, i.e., it ranges over the entire simulated system, and is typically restricted to a single level of abstraction.

The three execution modes arc table-driven, interpretive, and compiled. Given a system description, a table-driven simulator will build tables that will be used to drive a special program. The execution of this program in accordance with what is in the tables constitutes the simulation of the description. An interpretive simulator will scan the description, executing what it seer, as it goer, without any sort of intermediate translation step. A compiled simulator will actually translate the system description into an executable program, and then run that program to simulate the system.

To reiterate, this hardware simulator taxonomy is ^\^ informal attempt to provide a discussion base. Each reader will probably be able to punch his or her own set of holes through it, and encouraging just that was part of the reason for presenting it.

Examples of various leaves on the above taxonomy tree may be found in the general body of design automation and hardware descriptive literature. Among multi-level simulators such as N.mPc [17], Multi-Sim [7], and MODAL [12] are found the leaves having the closest relationship to I he SI IDF: simulator. Two leaves in particular, SARA [9] and SABLE [13], will be discussed in the following section.

2.4 Discussion of SARA and SABLE

SARA embodies an extremely general design tool of which behavioural simulation is only a part. One of the fundamental concepts of SARA is the required separation of structure from behaviour to enforce intended design modularity. In this regard, two sets of modelling primitives are provided in SARA, one for structure and one for behaviour. The structural primitives are embodied in a language called SL/1 [10], and the behavioural primitives exist in

the form of Graph Models of Behaviour (GMG) [19].

An SL/1 structural model is built up with "modules" whose internal structures are not visible from the outside. Modules interface with each other through "sockets" which can be connected to other modules' sockets. Since one can substitute the unseen internal structure of a system module for an explicit structure of interconnected modules without affecting other parts of the system, hierarchical modelling becomes possible.

These structural models are empty shells in the sense that there is nothing in particular to simulate in the absence of a behavioural model. GMB's provide behavioural specification, and are simulated with a GMD simulator [19]. They are divided into control primitives and data primitives; thus the designer is required to be able to make this separation in describing his design. The control primitives are in the style of Petri nets, having control nodes, control arcs, and tokens. Additional control primitives are input and output control "logics", which control the flow of tokens according to logical relationships among the associated control arcs. The data primitivrs consist of data sets (which arc passive collections of data), data processors - controlled (by a control node) and ync_qnlrqllrd -, and data arcs to glue together sets and processors.[1] Iking these primitives the user can create a hierarchical model of the behaviour of the system. Before it can be simulated though, the behaviour of the processors must be specified. PUP, a modification of PL/1 , is used to describe the actions of the processors.

After a behavioural model of control and data graphs has been specified the GMO simulator can execute it on a "token machine". The capabilities of this simulator include interactive commands to start and end simulation, set breakpoints, specify initial token distributions, and examine the graph states.

A more recent development in the hardware simulator domain is SABLE [14]. SABLE stands for Structure And Behaviour Linking Environment , and as its name suggests, it represents a more integrated approach to structural and behavioural description. As in SARA, different description mechanisms are used to specify behaviour and structure. System components, called complyper., are modelled as self-contained units that communicale through nets. Unlike SARA, components are not separated as to data and control; it is up to the user to enforce any such distinction. The structural nesting of components and component interconnections are described in a structural description language called SDL [21]. The behaviour of a component is described in ADLIB [13], an extension of PASCAL. SABLE will pull together

[1] A CPU would bo an oxamplo of a controlled thin processor; combinational logic would be an example of an uncontrolled data proencnor.

ADLIB-specificcl components according lo wh.it it sees in the SDL structural model for a given system, so that a r.yr;lem thus modelled can be simulated. Now, since a given system can be described at various *structural* levclr. with SDL, i.e., levels of structural refinement in a hierarchically modelled system, and at various *data* levels with ADLIB, i.e., layers of data structures that approximate other data structures, multi-level simulation is indeed achieved in SABLE.

There are two modes of behaviour specification for SABLE components. One is characterised by reaction to events at component nets, the other by clocked behaviour. These are the only mechanisms provided to activate component actions. Components are not allowed to access nets that are not their ov/n. What results is a high degree of modularity in the simulated model, since the internal details of components have no effect other than where they are immediately employed.

At this point, one can go back to the taxonomy tree and locate SARA and SABLE. On *an* item-by-item basis, it is found that

- SARA uses an abstract language (GMB) and a slightly modified general purpose programming language (PL1P) to describe behaviour; SABLE uses an extended general purpose programming language (ADLIB).

- SABLE and SARA cover levels of abstraction from system level to gate level, but SABLE uc.es only 1 language while SARA uses two.

- SABLE *and* SARA characterise behaviour detcrministically, and structure formally (SDL and 5L/1).

- By providing separate descriptive mechanisms for behaviour and structure both SABLE and SARA promote a strong degree of separation between the two.

- In both SARA and SABLE the binding of structure and behaviour takes place before simulation begins.

- SARA is best described as an event oriented simulator ; SABLE is a mix of event oriented and process oriented actions.

- The execution of SABLE can bo characterised as compiled (into PASCAL) whereas that of SARA is interpretive.

Having gone through lhir» exercise, the degree of similarity that exists between SARA and SABLE should be more evident. Comparison and contrast along topological lines with the SLIDE simulator will bo delayed until some notion of its structure and function has been related. The next chapter will provide such an overview.

3. SLIDE Simulator : Implementation and Executing Environment

The purpose of this chapter is to describe how the user goes about producing a runnable simulator from SLIDE, as well as the nature of the resulting simulation environment.

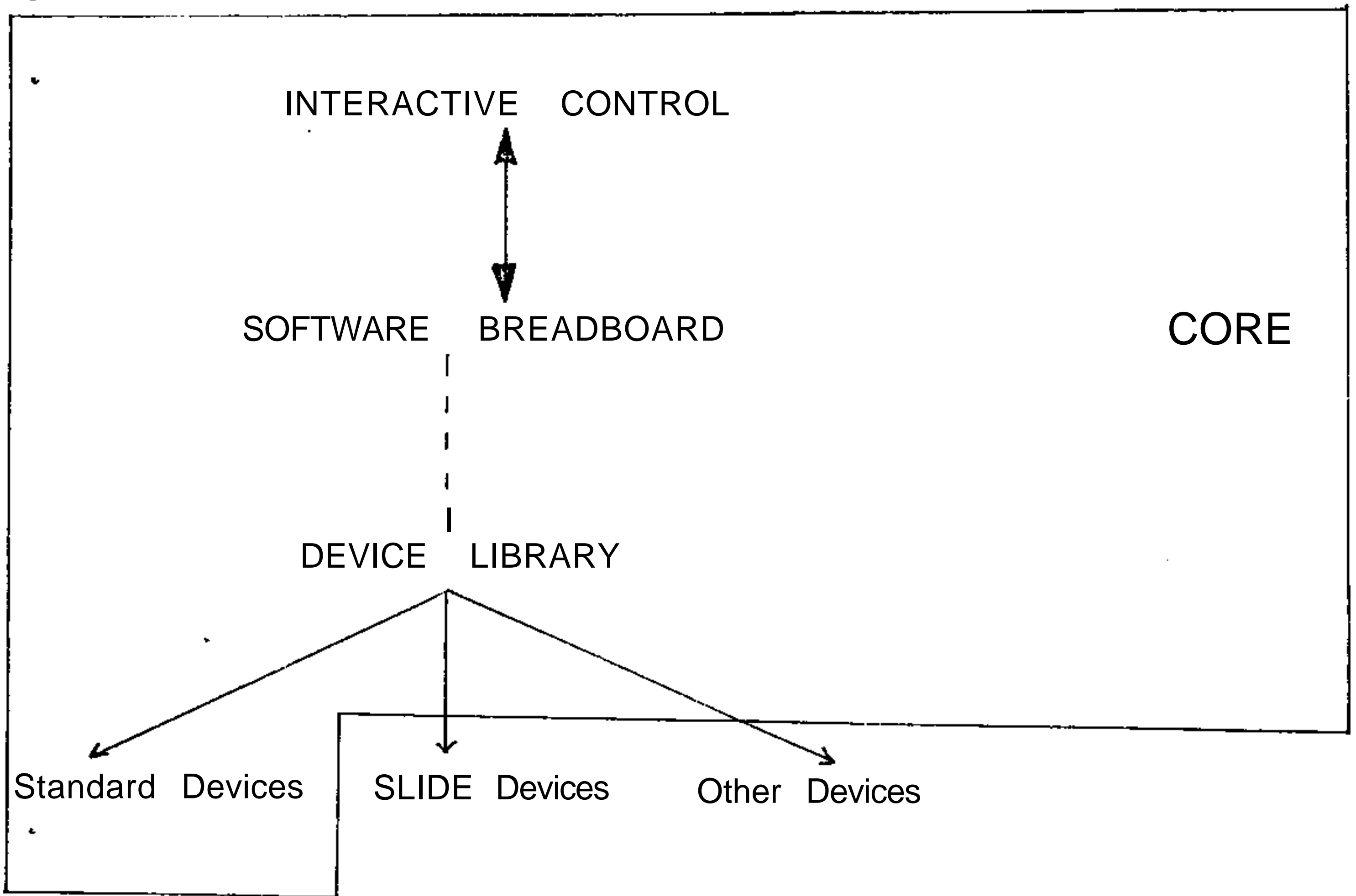
3.1 Implementation

As has been previously mentioned, SLIDE devices are simulated inside a simple multi-level simulator. SLIDE devices use this simulator as a software breadboarding faclility, allowing the user to interconnect interfaces and simulate them. Figure 1 illustrates the relationship that SLIDE modules have with this multi-level simulator core,and Appendix 1 provides a summary of the interactive simulator commands that arc available to the user.

By way of explanation of this implementation, consider an example scheme in which n different kinds of devices are to be interconnected. The communication strategy specified for the scheme dictator, tho interfacing behaviour of each device type. So the first step in realising a simulation of the communicating system is to write a SLIDE behavioural description for each type of component. Note that only one description per device *type* need be written. Once the n device types have been so specified, a number of processing steps are done on those SLIDE descriptions, the end result of which is a runnable simulation environment. This environment contains a library of devices that now includes executable models of each of the n original dovic n typos. These models exist as dynamic data and control structures, so that unprcdctcrminecl numbers of these devices can be interactively created and connected. In this way, the designer can proceed to put together a variety of sample configurations and simulate them using the software breadboard of the simulator core.

Various properties of each SLIDE description may be parameterised. This is done by allowing numbers in a SLIDE description to be replaced by special identifiers called *simulation time parameters* (STP). STP's *sre* bound interactively, by the user,for each instance of a SLIDE functional module in the simulator.

Figure 1: Simulator Structure



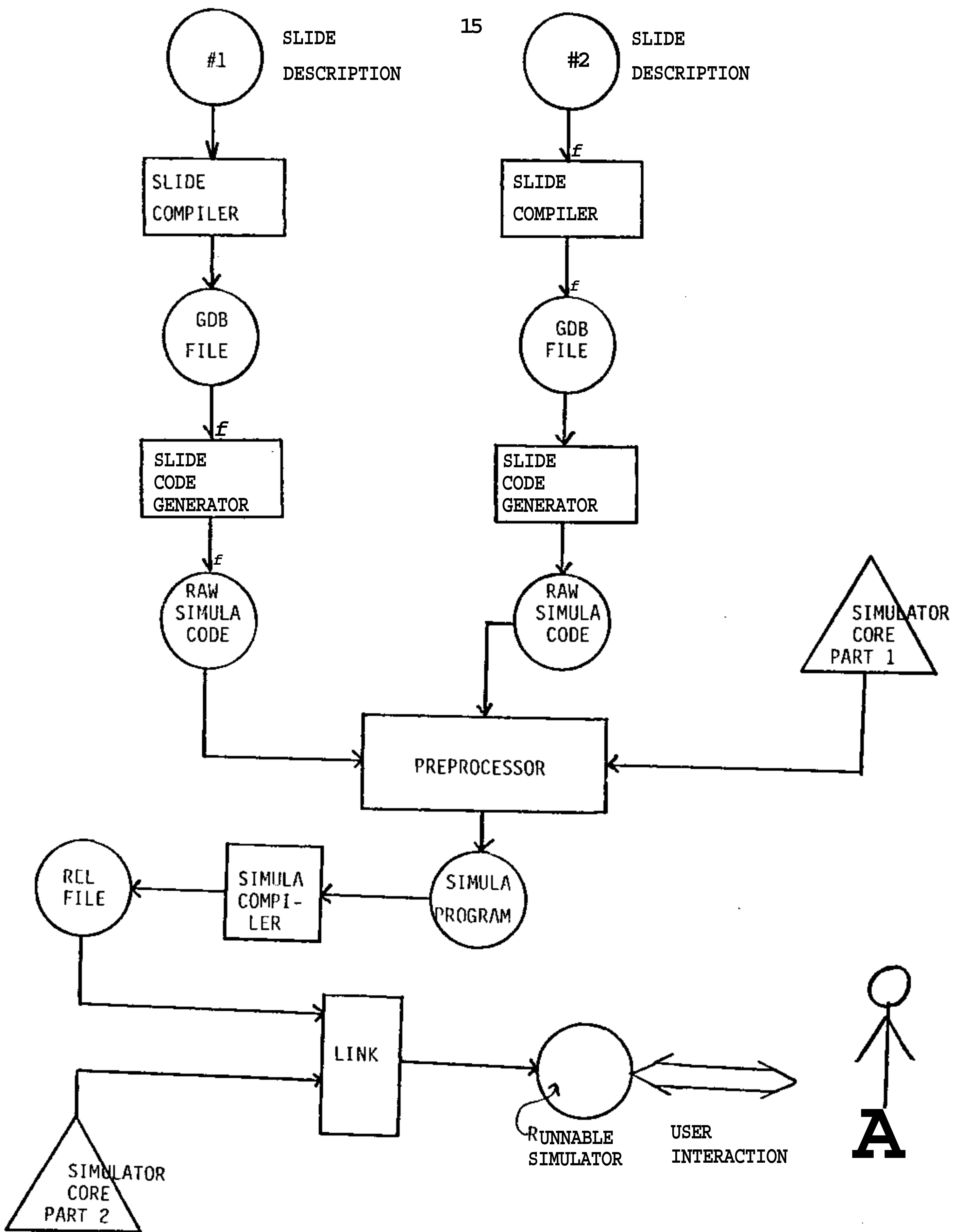


Figure 2: Producing a Runnnble SLIDE Simulator

It has been mentioned that there are corlain processing steps that the user performs to produce a runnable SLIDE simulator. These processing steps are illustrated in Figure 2. To begin, the SLIDE descriptions must be checked for syntactic correctness, and so the first step is to put each one through the SLIDE compiler. Once the descriptions have been syntactically cleaned up, the compiler will produce a GDQ parse tree file for each of them. A Global Data Base (GDB) is an ASCII representation of the parse tree of a SLIDE description. The mapping from SLIDE source to GDD is a reversible transformation. For details of the SLIDE compiler and the GDD it produces see reference [23].

The next step involves performing the mapping for each GDB file from parse tree to SIMULA code. The SLIDE-to-SIMULA code generator is responsible for performing this function. The form of its output is not a monolithic SIMULA program, however. It is a collection of SIMULA code fragments, where each fragment has a label associated with it that marks where that particular fragment is to go within the eventual simulation program. Now, after each GDB file has been put through the code generator, it is time to combine the various SIMULA fragments into a single program. The preprocessor, which was designed and written by Rich Bollinger, is used to perform this function. The preprocessor operates on the labelling information provided in the fragment files to piece together one syntactically correct SIMULA program.* Notice that in addition to whatever SLIDE files are provided by the user, the preprocessor uses as input a pnrt of the multi-level simulator core. This part contains a library of standard devices such as gates and counters , plus the simulator monitor. This part of the core provides the necessary skeletal code to guarantee that the output of the preprocessor is indeed a complete SIMULA program.

The rest of the multi-level simulator core consists of support code for both SLIDE simulation and general simulation. This code has been precompiled ; it is linked with the compiled preprocessor output to produce the final runnable simulator.

3.2 The Multi-Level Simulator

---

[1] Ono in!oror.tinp nnporl of thin in that llm SIMill A code ^onoraled from tho SI IDT modulon doon nol havo lo bo exactly nlructurrd on defined by SIMULA r.ynlnx ruler. For oxample, declarations can bo gonoratod and inGorfod on tho fly by tho codo ^onor.ilor m the GDR filo in proconnod; the preprocessor will clean *up* and roordor tho SIMULA code for tho SIMULA compiler. Thin dtntribution of tho tankn of a complex problem among different software look simplified the programming problem grontly

3.2.1 Requirements for Multi-Level Simulation

The multi-level simulator provides a decentralized, dynamically alterable environment for the interconnection and simulation of digital systems. It embodies the basic vertical communication[1] requirements described by MacDougall [16] for multi-level simulation. In a nutshell, MacDougall asserts that standardised vertical communications are the fundamental prerequisite to multi-level simulation. An invariant interface for inter-component communication should exist that all components both can and must use to communicate with each other, no matter what their relative levels of abstraction may be.

This standard interface must be general enough to support the varying amounts of data detail that will be transmitted through it. The design of the interface itself and of the communications that it supports should be done with the emphasis on efficiency, since this is clearly an area where a little extra overhead one way or the other will have a significant effect on simulation execution speed.

A side effect of this interfacing requirement is that a certain amount of separation between structure and behaviour is encouraged on the user's part. Having to define a standard vertical communications interface pre-supposes the ability to draw a dotted line around an entity and label it a "component", which in turn needs to be interfaced to other "components". Thus, the problem of describing a collection of hardware is broken down into one of defining components (having behaviour and internal structure) and defining their interconnections (external structure or topology). So software that supports multi-level simulation must by extension encourage the partitioning of the digital system description problem into behavioural and structural description sub-problems.

3.2.2 SIMULA

The implementation of the simulator core is based on the coroutining and discrete event simulation primitives provided by SIMULA-67 [4]. A simulation program written in SIMULA uses special coroutines called *processes*. For a *process* to execute, it must be scheduled by placing it in a special linked list called the *event list*. The list is ordered by the simulation time associated with each process. The process at the front of the list is due to resume execution, and the time associated with this process is considered to be the "current" simulation time. A process can be removed from the list and rescheduled at a later time. Thus, time moves ahead in discrete jumps, and the simulation is process oriented .

---

[1] Communication from one level of abstraction to another.

3.2.3 Data and Control Structures of the Simulator Core

The structure of the simulator core is characterized by three classes of dynamic data structures and the operations performed on them. These are: The *Element^* the *Chain* and the *Simulation Process* (SP). Together, the Element, Chain and the SP completely specify the functioning of a given digital device module.

The Element and the Chain constitute the common external characteristics of the device modules; the internal structural and behavioural differences have been abstracted away. Together, the Element and the Chain form the standardised vertical communication interface that MacDougall describes, enabling multi-level modelling and simulation.

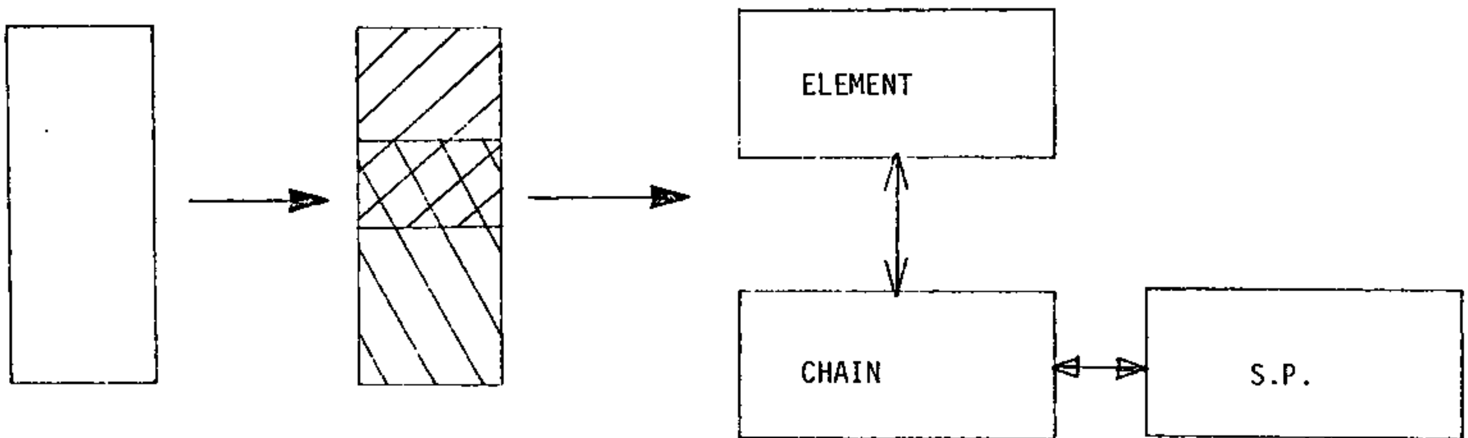
The SP contains the information internal to a device module - the meat of a device description. It is a general data structure that can reference a number of SIMULA processes. The SP is the structure that corresponds to a "component" in the simulator ; SP's can only communicate with other SPY* through the provided interface , that is, the Element and the Chain. Figure 3 illustrates the partitioning of a device along the above lines.

The Element is designed ar. a passive vehicle for the interconnection of device modules. Each Element contains n set of records representing the "ports" of a device. Ports can be connected to other ports, and the stale of the data represented by the ports of a device module indicates the externally visible state of the device.

The active part of tho interconnection mechanism resides in the Chain. One Chain is associated with ench clement; the Chain is responsible for the actions and reactions of a device with respect to ils ports. The Chain also acts as an intermediary between the SP and the Element, which do not interact directly. The Chain is implemented as a coroutine, whose actions are performed in zero time and are invisible to the SIMULA scheduler.

An example of a simple device module representing an inverter is given in Figure 4 . The SP is shown as SIMULA code, the Element ir, represented as an abstract data structure, and the Chain is illustrated in its role as interface between the two.

Figure 3: Partitioning of a Digital Device in the Multi-level Simulator



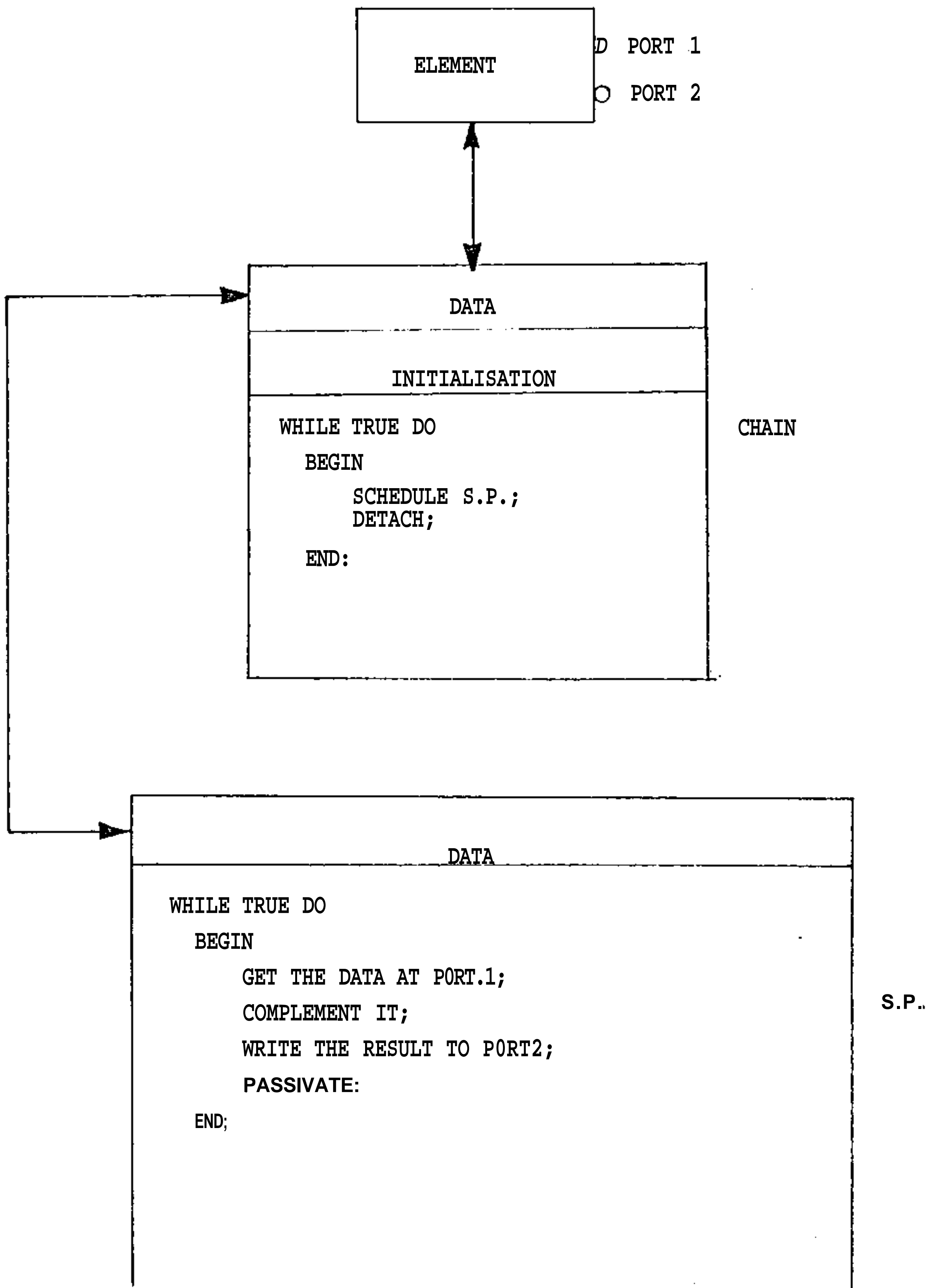


Figure 4: Representation of an Inverter Module

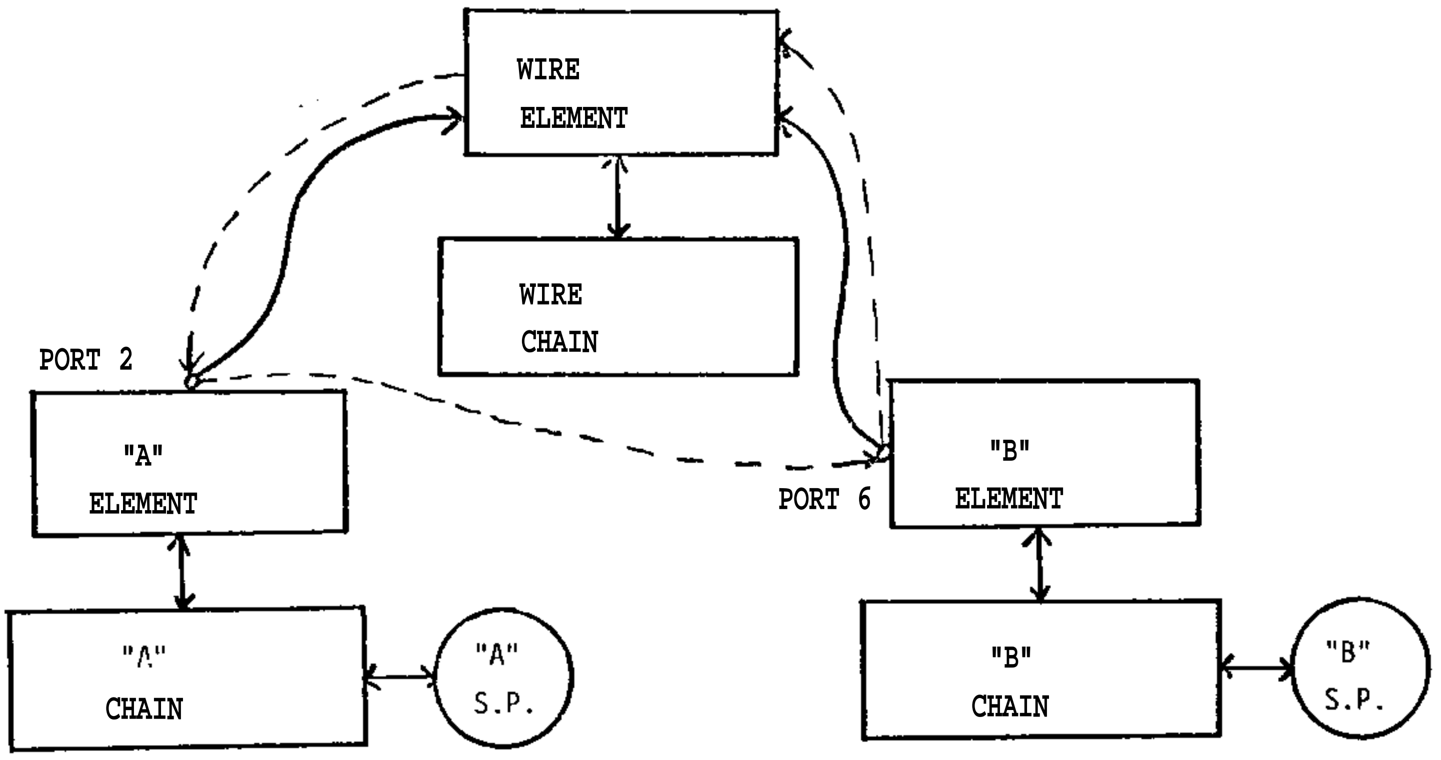
3.2.4 The Modulo Interconnections

Modules can be interconnected in two ways in this simulator - directly and via special *wire* modules. Direct connections allow the data records at one port to be directly accessible by all connected ports. This concept is useful, for example, when a CPU module has been described as a number of submodules, whose ports exist only as an abstraction. (A Z80 CPU has been described in this manner [8]). This method of interconnection allows structured design and pretesting of each submodule.

The more common connection method for large system simulations is the wire connection, meant to correspond to the usual physical interconnection of digital devices. A wire can be considered a degenerate case of a digital module, having no SP or explicit ports. Through its chain, the wire houses the actual wire data, and identification information and procedures which allow its logical behavior to be modelled. A wire connection is modelled as the direct access of ports to the same wire device, not to each other. A good example of this type of connection is an open-collector bus, since the data presented to the bus wires at a port might not reflect the actual logic values on the bus.

Wire typos arc dktinguished by logical behavior, data representation , and synchrony ; interconnection!"* of wircr. with differing properties is in general not allowed by the simulator. The user can interactively wire together the components of his system, producing a structure like that shown in Figure 5 . Changer, in the system being simulated can be made by adding components without halting program execution.

Figure 5: Representation of Device Interconnection by Wire



CONNECTION

- - - - - - LINKED LIST OF PORTS CONNECTED TO THIS WIRE

The multi-lovel simulator trentr. Ihe creation of complete devices as something to be delayed until just before actual simulation. When the user is interactively connecting ports, only the Elements are involved; the Chain and the SP are not yet created. When the simulator does create the rest of the device, it will only explicitly create the Chain. The Chain is expected to create Ihe SP as part of its initial actions, and the SP in turn will initiate the remaining internal data and control structure initialisation.

During execution, the SP's may write to ports by depositing data in the appropriate wire device, via a series of references. The v/ire Chain coroutine is called, and it proceeds through a linked li&t of the ports connected to the wire, activating the Chain associated with each port. This givos each relevant device the opportunity to schedule processes, and it proceeds in zero simulation time. Delays are introduced only by the scheduling of processes related to a device module and by explicit delays within each device module's SP.

3-3 SLIDE Simulation Overview

The previous sections have provided a glimpse of the way SLIDE simulations are produced along with the surrounding core environment, and a summary of the workings of the core itself. The discussion moves on now to consider more closely the particular nature of the SLIDE device / simulator core relationship, followed by an explication of the mechanics of SLIDE simulation.

3.3.1 Overall Relationship to the Core

Given thr Inck of simulating a rcjjisler-lransfcr level hardware descriptive language such as SLIDE , there is no inherent reason why that simulation cannot be accomplished in either table-driven, interpretive, or compiled mode. However, in addition to the normal requirement for speed of execution in an interactive program, in this project there was a requirement of compatibility with the multi-level simulator core. The nature of device simulation in the core forced the simulation of SLIDE descriptions to be done in compiled mode. This most immediate result of the embedding decision characterises Ihe nature of the relationship between SLIDE and the core.

As has boon alludoH !o olr.cwhore in this report, there is a one-to-one relationship between a SLIDE description and a resulting device in the simulator. That is, SLIDE descriptions will be mapped to exactly one library device, not three or ten. This may seem to be a trivial point, but the fact is that there were alternatives. It was suggested early in the project that SLIDE descriptions bo automatically analysed structurally as well as behaviourally to ultimately produce a group of directly connected subcJcvices. The aggregate behaviour of

the subdevicc; would constitute the simulation of the original SLIDE description. This alternative was rejcclrd because it v/ould havo been necessary to force an artificial software structure onto tho coro in order to accomoriale SLIDE interprocess relationships, and because it would have added an order of magnitude in complexity to the problem at hand without adding any fidelity or efficiency lo the simulation. The more intuitive notion of one library device per SLIDE description v/as chosen instead, the tradeoff being in the relative difficulty of deleting devices from interconnection structures; deletion in the multiple device approach should be easier, theoretically.

Since SLIDE descriptions map into one Element, one Chain, and one SP, their *superficial* characteristics do not immediately set them apart from other devices. What does set them apart is their *imdcrlyinu* characteristics - for example the behaviour and data structuring of the SP for a SLIDE device. Note that the degree of complexity of the SP for a given device, even a SLID!. dovicr, is not disccrnable by tho core. This reflects the software decision to bury the details of SLIDE behaviour so that they v/ould not affect the multi-level simulation as such. Even those software extensions that *v/crc* made to the core such as adding in a new wire type that could handle the high level of data detail required for SLIDE simulation were just that - extensions, not design modifications. To sum up, then, the relationship of SLIDE devices to the core is equivalent to that of non-SLIDE devices.

3.3.2 SLIDE Dovico Functioning During Simulation

The outlook of SLIDE device simulation is process oriented in that the focus of attention is the manipulation of SLIDE processes on the SIMULA scheduling list. But even beyond the narrow definition of "process oriented", SLIDE simulation necessarily reflects the point of view of SLIDE descriptions, and the SLIDE process is the fundamental descriptive tool of the language. So to describe the simulation of SLIDE processes is to describe the basic functioning of SLIDE device simulations.

At the beginning of the simulation (t«0) , after variable initialisation, all the SLIDE processes of a given device instance are given a chance to begin executing if they can. According to the semantics of the SLIDE language, a process may start executing whenever

1. Its declared initialisation conditions become true, AND

2. All the processes of which it is a subprocess are executing, AND

3. No process at the same "process level" is executing and has a higher priority.

Processes that could have started but for cither or both of the second two points above

are put into a special linked list. As each SLIDE process starts its execution, it evokes the "SLIDE Scheduler" which checks this list to see if any of the members can start up as a result of its initialisation. The SLIDE Scheduler will also assure that any executing process which is at the same "process level" ,but of a lower priority than the process which is about to begin executing, will be terminated. As a SLIDE process executes its actions, variables such as lines, buffers and registers will be accessed. Each time such a hardware variable is written to, it is responsible for checking if any of the relevant expressions of which it is a member are now true,[1] and if they are, evoking the SLIDE scheduler. As each SLIDE process terminates, whether by completing its actions , or by the mechanisms described above, it is entered into the special linked list. The list is once again checked to see if any of its members can begin execution.[2]

This special linked list is the key data structure in the operation of the SLIDE scheduler. It will contain those processes that are likely to attempt to begin executing due to implicit changes in system state , i.e., SLIDE process initiations or terminations, as opposed to explicit changes in system state , i.e., changes in SLIDE variables. This list fulfills a need for scheduling efficiency ; it would not otherwise be known, when an implicit state change occurred which processes would need to be checked for possible initiation and which could be safely ignored. Contrast this with explicit state changes , where the set of processes that can be safely ignored is fixed.

It should be noted that whenever a member of the list is being checked, it will be removed from the list if its Boolean initiation conditions are found to be false. This ensures that members of the list are only those that could be started due to implicit state changes.

---

[1] Relevant expressions are those which appear in the initialisation conditions of a process, or in a DELAY statement

[2] In terminating, the old instance of a process is garbage-collected, and a new instance of the process is created. It is this new instance that enters the list.

4. Mapping SLIDE to SIMULA

The programming tafɔk presented by this project was, like many non-trivial tasks, **best** attacked by breaking it down into a set of sub-tasks to be performed. The SLIDE **simulation task** was broken down into three sub-tasks , namely

- A SIMULA code mapping had to be devised that would not only simulate the intended behaviour of a SLIDE description as closely as possible, **but would also** fit into lhe multi-level simulator environment.

- With this mapping specified, a program had to be written that would do **the** actualtranslation from GDD file to SIMULA.

- Software had to be written to take a group of such translated SLIDE descriptions and incorporate them into lhe multi-level simulator.

The rest of this chapter will elaborate on the first of these tasks and on the nature of **the** software solution. The last two tasks will be dealt with in subsequent chapters.

4.1 A Review of SIMULA

The classical code generation tar»k í> that of mapping a complicated high level language into **a** simple low level language, CC- Tertian to PDP-11 assembly language. In the case of this project, it wns required to translate instead from a register-transfer level hardware descriptive language to a powerful special purpose programming language. While this meant that the solutions had to be to a certain extent *ad hoc,* this did present a certain advantage, namely, the ability to build a sophisticated body of support code that would enable the eventual code generator to produce shorter and simpler programs. To have reached as high **a** level of complexity in assembly language support code as was reached in SIMULA **would** have required infinitely more time and effort.

The semantics of SIMULA provide mechanisms for building new capabilities **on top** of existing oner, in a structured fashion. This feature of the language was to be at least **as** important as proce',» oriented simulation primitives were to the successful completion of **the** project. One can define in SIMULA a dynamic structure called a *class.* Like PASCAL records, SIMULA classes contain heterogeneous data, nnd numerous instances of a given class, called *objects,* can be cronlrd during program execution. In addition, though, SIMULA classes can have actions c'l'^ocuilrd with them , nnd each object will have its own copy of the actions of the class. Ther.e action', are executed no coroutines. Once a class has been specified, it may be used as a *prefix* to create *subclasses* which, in addition to any attributes they may have defined for themselves will automatically have access to the attributes of the prefix. For example, given class A , A class B, D class C, D-objects will have copies of A-type data and

actions that they c,nn accrv.r. in addition lo those in lhe actual definition of D. Similarly, C-objccts will havo hotii A and B capabilities available. The order of execution of actions is according to'prefix *order-,* when a C-objecl executes ils actions, it will run through A actions « and B actions before doing C-actions.

The simulation primitives provided in SIMULA are embodied in a special prefix that the language provides called class SIMULATION. SIMULA processes are objects of the class PROCESS, a prefix that SIMULATION makes available to the programmer. The task at hand in producing simulations of SLIDE processes was in part to build up the correct capabilities using PROCESS as a building block.

4.2 SLIDE Device Implementation

Every SP of a SLIDF device modulo in the simulator is a subclass of class BOX. Class BOX contains the data and routines that am constant for each SLIDE device; device specific data and procedures are specified in the subclass itself. The subclass therefore houses all the SLIDE hardware variable., the priority tree, rind pointers to other relevant data. Creating a new instantiation of the SLIDE device in the simulator means that a new BOX subclass object is created, ≤*m*<\ this in turn causer, the creation of every data and control structure needed for the simulation of the SLIDE device.

4.3 SLIDE Processes

4.3.1 Issues

In bringing together the divergent concepts of SLIDE process simulation and SIMULA process simulation, certain programming issues had to be addressed.

- Each SLIDE modulo, is a mapping of an independent SLIDE description, each having its own SI IDE process structure. The process structure and hence the particular scheduling constraints of a given SLIDE device are independent of all other SLIDE devices. The multi-level simulator has already been seen to run in a decentralized fashion; each device ic. a collection of autonomous SIMULA objects. It soemod consistent lo make each SLIDE device responsible for the scheduling of its associated processes, rather lh.m create a central core scheduler of some sort. So, lhe SP nf a SLIDE device was given lhe control structures to implement the scheduling la*.k for that device. Each SLIDE device module houses a related, but unique, version of the SLIDE scheduler.

- SIMULA processes as such are not endowed by class SIMULATION with sufficient scheduling dala structures to correctly model SLIDE processes. A SLIDE process within a SLIDE description is a member of a complex priority tree, specified in the description by the static nesting of the process ancJ by explicit priority

numbering. Initiation and termination of a GLIDE process must always be referenced to Iho priority tree. Such bookkeeping is clearly beyond the scope of a SIMULA process. Resides, the concept of termination itself , implying the automatic garbage-collection and recreation of a process and all its subprocesses, does not exist for SIMULA processes. Thus , most SLIDE process semantics had to be built up, using class PROCESS as a prefix, plus additional control structures.

- Each SLIDE scheduler within a SLIDE module needs to be invoked as the result of many different events. For example, a process can be started or terminated due to a change in a SLIDE variable (explicit stale change). Because of the priority structure, a process can also be initiated or terminated as a side-effect of some other process initiation/termination, even if these processes do not explicitly communicate at all (implicit state change). The SLIDE DELAY statement adds another complication by allowing a process to reschedule itself at any time, and to be woken up on some set of conditions* (possibly subject to a timeout). Hence extra control structures were designed to allow a simulating SLIDE device to evoke the scheduler at all such *significant* events.

4.3.2 Implementation

To more car.ily understand the nature of the implementation of SLIDE processes in the simulator, it is worthwhile to examine SIMULA processes. SIMULA processes are not only characterised by their explicitly defined actions and data, but also by the external control structures they are associated with. The SIMULA scheduler, consisting of the SIMULA event list and related scheduling routines, governs the "global context", if you will, of SIMULA process behaviour. SLIDE processes are also characterised by an external control structure, the SLIDE scheduler, *and* by their membership in a fixed data structure (the SLIDE process priority tree) as well ;is by their own explicitly defined actions and data. Together, these data and control structures govern the global context of SLIDE processes.

Figure 5 illustrates the static data structures of a SLIDE process. In the implementation of SLIDE processes, class PROCESS was used as a prefix to provide the rudimentary capabilities of interaction with the SIMULA scheduler. Two other properties that are common to every SLIDE process arc physical membership in the aforementioned priority tree and possession of an explicit priority number. These attributes were combined with class PROCESS to produce class SLIDEPR, the prefix for all SLIDE processes. Each SLIDE process is implemented as a SLIDEPR subclass with its own particular data and actions (body).

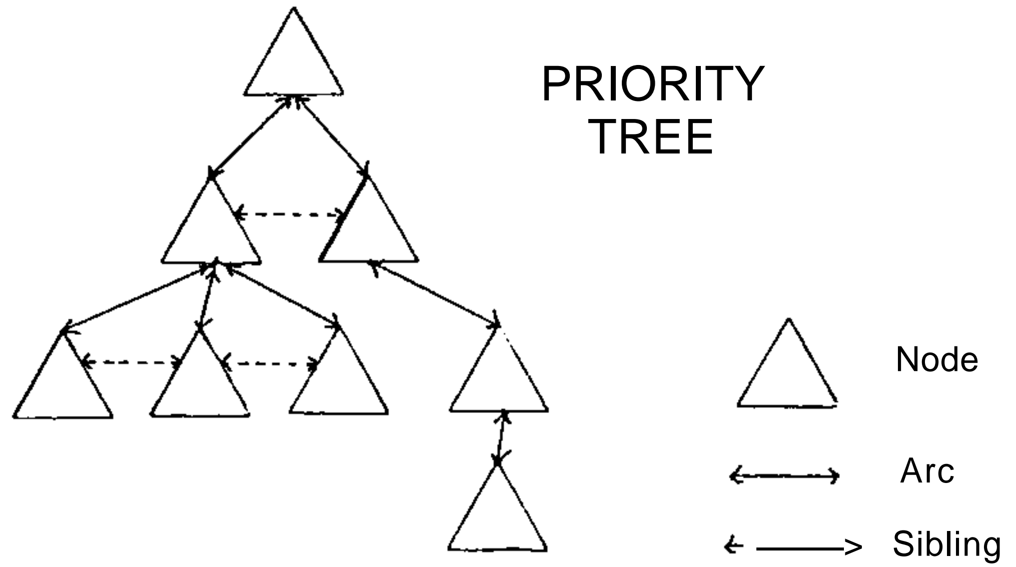
The next question in SLIDE process implementation concerns process termination, which physically calls for the particular process to be garbage collected and replaced. Now,

---

[1] Thoso conditions aro arithmetic and/or logic «?xprossion9 of SLIDE variabloii

regardless of initiation or termination, the membership of a SLIDE process in the priority tree is fixed. It is of no value to constantly create, destroy, and recreate parts of a priority tree along with the SLIDE processes themselves. So rather than support direct SLIDE process membership in the priority tree, it was decided lo separate the two by implementing the tree as an independent data structure which the 51 IDE process would reference through a pointer. Thus, as SLIDE process instances arc created and destroyed, only pointers to the priority tree come and go, not the components of the tree itself.

Further addressing the question of termination, it is important to try and implement this task as efficiently as possible, since this is potentially a major source of execution overhead. This task entails the removal of all references to the dynamic data structure modelling the SLIDE process at tho time, and redirecting them at a brand new instance of that data structure. If there was always only one pointer to a given SLIDE process, and that pointer was always easy to locale, this would reduce the workload of the task immensely. This is precisely the purpose of lhc *flak* object depicted in Figure 6. Flak objects "take the flak", in that all data structures wanting to directly reference a SLIDE process must settle for an indirect reference through the flak object instead. As is seen in Figure 6 , although the SLIDE process is abfe to reference its representative *node* in the priority tree **directly, that** node must go through the flak object to reference lhc SLIDE process.



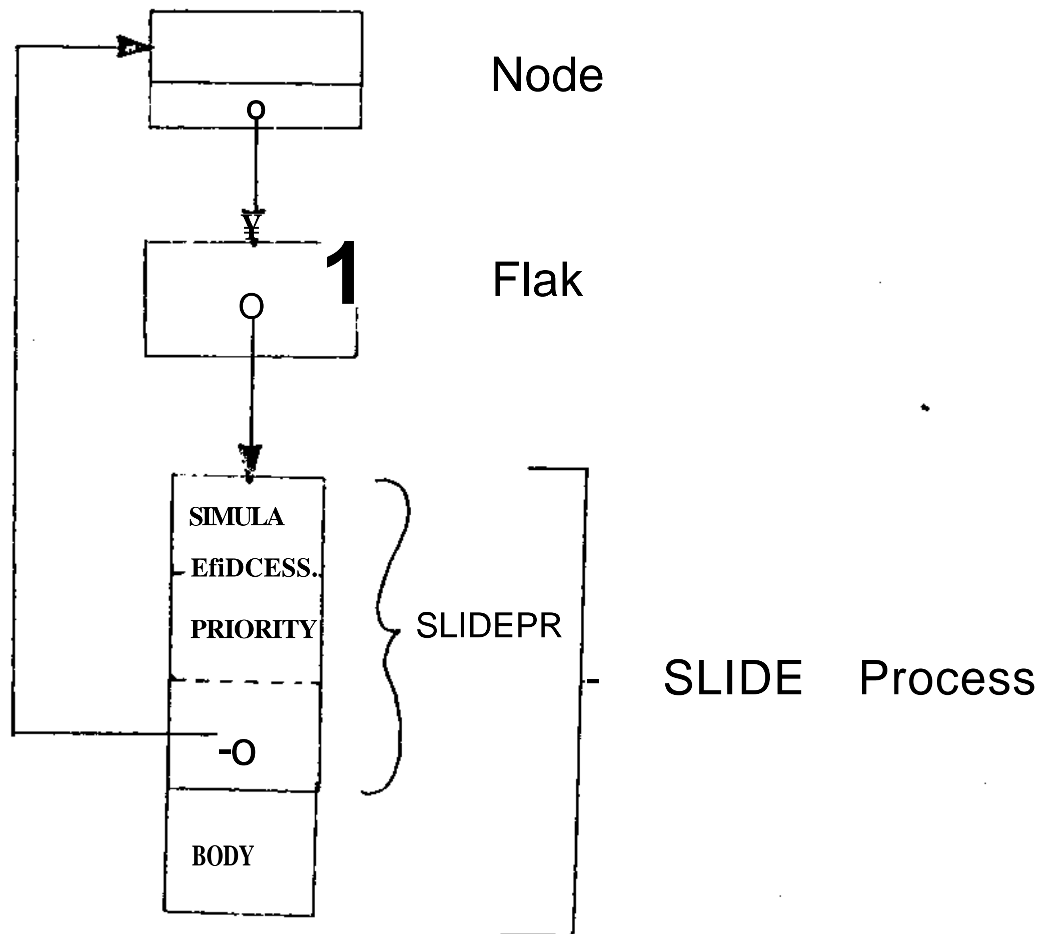
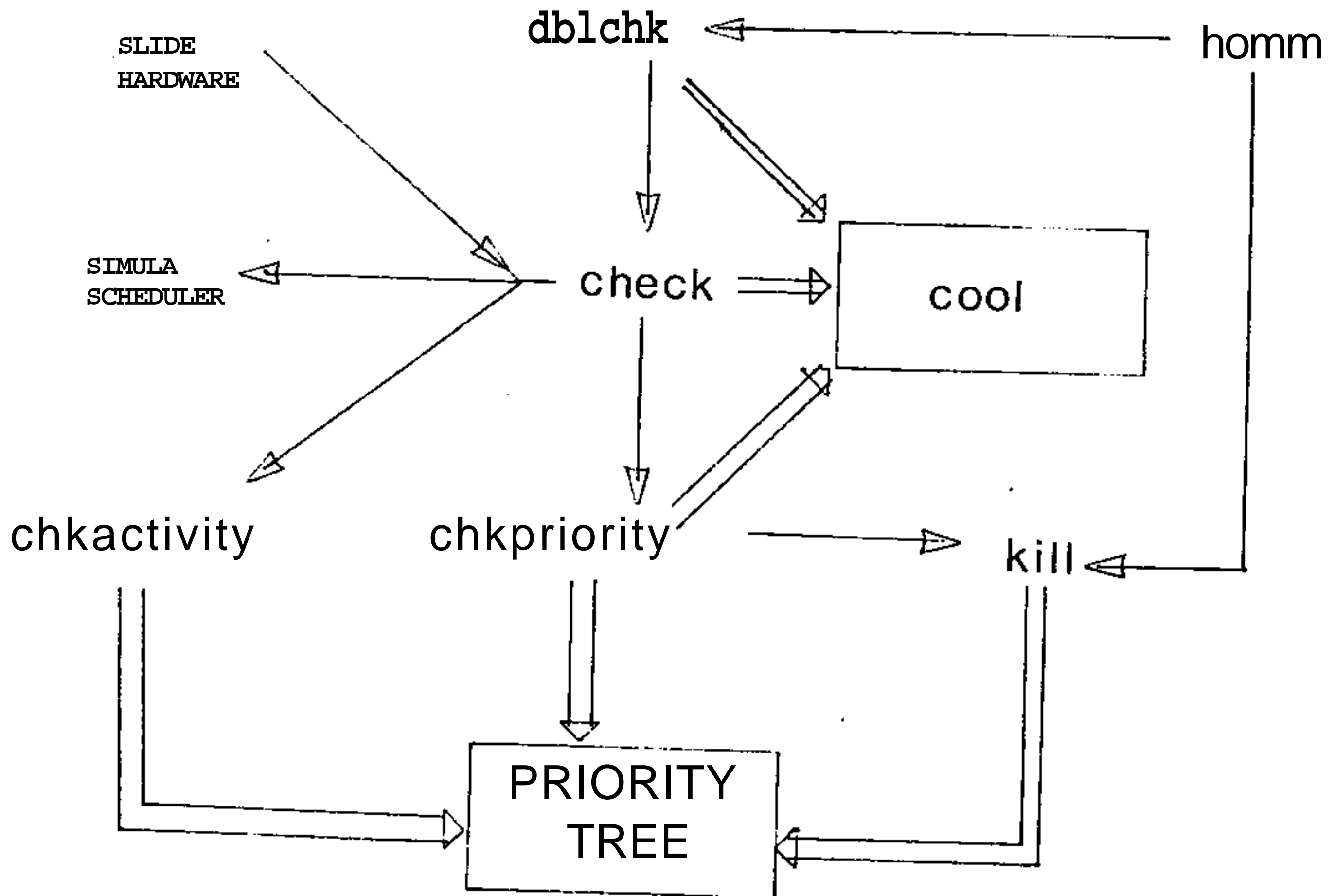


Figure 6: Static Data Structures of SLIDE Processes

Figure 7: Structures of the SLIDE Scheduler



X ——▷ Z X invokes Z

X ══▷ Z X accesses Z

The part of the 51.101' proce^ implementation described ahovo is a fixed data structure. The "SLIDE Scheduler" is embodied in an external control structure that operates on this fixed data structure as well as on itself, to provide the scheduling function for SLIDE processes. Three components arc involved in achieving this. First, there is the SIMULA scheduler, which comes "free" with class PROCESS. Second, there is a dynamically varying special linked list , named *cool,* of SLIDE processes (actually, flak objects) . Third, there is a set of six major scheduling routines that operate on the priority tree and the special linked list cool to perform the actual process manipulations. These routines are named *check, chkactivity, chkpriority,* $kill_9dblchk_f$ and *homm.* The structure of the SLIDE scheduler is illustrated in Figure 7.

Procedure chock is the central scheduling routine for SLIDE process simulations. Its operation can be charnrteriscd as follows. In any SLIDE description, there will be a finite number of DPI.AY 'tntements of the kind that contain a Boolean expression, and a finite number of initinlir.ation condition declarations (or SLIDE processes, each of which will also involve one SLIDE Boolean expression. All these expressions have the property that if they become true sometime during the simulation, a SLIDE process may need to be scheduled or rescheduled. Procedure check catalogues these expressions and the processes they affect. Whenever check(i) is invoked, the ith catalogue entry is evaluated, ancJ if the listed expression is found to be true then check will attempt to schedule the associated process. Whenever SLIDE hardware variables have data written lo them (explicit stale change), they will invoke check once for each catalogue expression in which they appear. The other source of check invocations is the mechanism that handles implicit state changes, which will be described in due course.

How far and in vyh.il manner the process scheduling attempted by check will proceed depends on the type of entry (initialisation or DELAY statement), and on the current state of the simulated device. For example, if the expression in a DELAY statement occurring in an inactive SLIDE process should happen to become true, it should not and will not have any scheduling effect on the process. In the absence of such obvious anomalies , scheduling will indeed be attempted, for a DELAY statement entry, all that is required is that the associated process be active for the SIMULA scheduler to be immediately invoked. The scheduling task for an initialisation entry is more complicated, as this calls for verifying that the various SLIDE procer.s initiation rules will not be violated if the given process is allowed to start. To handle this necessary bookkeeping, two Boolean procedures are provided : chkactivity and chkpriority. Check will invoke each of them in turn , and if they *both* return true this indicates that the process can indeed begin execution consistent with SLIDE process initiation rules, and so check can invoke the SIMULA scheduler.

Procedure chkaclivily is responsible for the enforcement of the SLIDE process rule that states that a process cannot be initialed unless all the processes of which it is a subprocess are already executing. II (Joes this by traversing the priority tree, making sure that every ancestor of the given SLIDE process is active, i.e., scheduled on the SIMULA event list. Procedure chkpriorily is charged with the enforcement of the rule that states that a SLIDE process may only bo initiated if no process at the same process level is executing and has a higher explicit priority numbering. It also accomplishes its task by traversing the priority tree, but this time it examines the *siblings* of the given SLIDE process. If any of them is active and has a higher explicit priority number than the given SLIDE process, then chkpriority returns false. While chKpriority is running, if any of the siblings is found to be active and possess a *Lower* priority than the given SLIDE process, then that sibling will be terminated , as is dictated by SLIDE semantics.

Now, if either of chkpriority or chkactivity returns false, the situation is a special one. The given SLID!' procrv.s has had its initialisation conditions come true, but could not start because of tho current Male of the device simulation. It is now possible that strictly due to the initiation or termination of some other profess (implicit stale change), this process could become eligible to r.lnrl. It is imperative to keep track of such a process so that it can be given a chance to start if it can. This is done by having check insert the process into the linked list cool for future reference.

Another set of processes that could he initialed due to an implicit stale change is the set of newly-terminated processes, and these too will be inserted into cool by the routine that handles SLIDE process termination, procedure kill. Procedure kill, besides performing the actual garbage collection and creation chores, also terminates all the descendants of the terminated process, as listed in the priority tree.

The mechanism for handling the scheduling task with regard to implicit state changes can now be described. I.inked list cool has been seen to contain all the processes at a given point in time that could be initiated by an implicit stale change. The scheduling of the processes in the linked list cool is done in response to *every* implicit state change that occurs. Whenever a 5UDE process is initialed , ils very first action is to attempt to activate every member of cool. This is done by invoking procedure clblchk, which removes each member from the list in turn and invokes check for its corresponding initialisation entry. Whenever a process is ahoiil to terminate, ils last action is to have itself "kilTcd and procedure check

properly invoked, all horn an inclopenden! third party SIMULA process, homm.[1] This completes the? description of the implementation of SLIDE processes as such. The next section discusser. Ihn implementation of Iho body of the SLIDE process itself.

4.4 Inside The SLIDE Process

4.4.1 Hardwaro

SLIDE variables can be lines, registers, and buffers, as well as combinational logic and associative memories. To achieve their simulation, SIMULA classes were written to emulate the behaviour of each hardware type.

The common characteristics of SLIDE line and register variables, such as read / write capability and the ability to detect bit transitions, were distilled into a SIMULA prefix, class hard. Subclasses for lines and registers were built up with class hard. In class hard the actions of reading nnrl writing data, and of cMocting positive and negative transitions on bits, are implemented as procedures. The actual actions of the class consist of more than the expected variable initialisation function, however. The actions of class hard relate to that part of SLIDE process bookkeeping that accomodates explicit state changes.

A hardware vai iable can affect a given DELAY statement or initialisation condition if it appears (either directly or Ihiough combinational logic to which it is an input) in the Boolean expression contained therein. From the last section, it is known that all such Boolean expressions arc catalogued in procedure check of the SLIDE scheduler. Therefore, included in the mapping of a SLIDE hardware variable is a list of the catalogue entries in procedure check that the particular variable can affect. The invoked actions of a hard object are to cycle through Iho entries so listed, calling check for each entry. Whenever a SLIDE line or register variable (^*nd* in a similar fashion, a SLIDE buffer variable) is written to by a SLIDE process, these actions of the variable will be scheduled. In this way, every explicit state change which could result in some other SLIDE process needing to be non-procedurally initiated is provided for. Providing the above list limits the scope of the search to only those entries that am relevant to a particular variable.

Now, class hard is a subclass of class PROCESS, so its actions are treated like those of any other SIMULA process. This means that the cycling action occurs autonomously from the

---

[1] Homm must ho a filMIH.A proc^^r, inn IP.-KI of n procoflnro, bocnuno ihn tnrminnting procoss shouldn't force its own garbogo collodion, which if would bo doing if lo invokp homm meant a procoduro call This point will come up **again** when SLIDE hnrdwnre variahlnn aro doacribrd

SLIDE process that set the variable, i.e., as a separately executing entity. But if as was noted above, these actions always occur during a write, it seems reasonable to inquire why they are separated from the write procedure in such a manner, rather than incorporated into it. The reason lies with the familiar case of SLIDE process termination. It is possible that during the course of its execution, a SLIDE process could in writing data to a hardware variable cause a higher priority brother process to begin executing. SLIDE semantics dictate that the first SLIDE process would have to terminate. This cannot be done directly from a procedure called by that process. As was the case with homm, providing a third party process from which needed terminations can be directed serves to maintain orderly execution. This is accomplished by making SLIDE hardware variables into SIMULA processes.

SLIDE global lines and registers represent a special case. They are the variables that are meant to correspond to "pins" in the SLIDE device description, so they have a responsibility to not only behave in a manner consistent with other line and register variables, but also to interact with the device interface to the rest of the simulator. This relationship with the standardised communication interface is crystallised along the following lines. For each global variable in the description, one port in the Element of the SLIDE device module will be assigned. The port and the global variable will cooperate to provide correct data to each other when necessary. That is, each time a global variable is written to, the resulting data is passed up to the corresponding port to be written to any wire that may be connected there. Conversely when a SLIDE device Chain is woken up, because the wire connected to one of its ports has been written to , the Chain will pass the data down to the associated global line or register.

A special case of the global line is the SLIDE synchronous line. The requirement for periodicity in behaviour called for a modification of the way the read / write functions were handled as compared with other lines and registers. A new SIMULA process called the history process was designed that focuses on the port in the Element that has been associated with a given synchronous line. Each sync line has one history process provided for it. The history process is first activated at time zero of a simulation and proceeds with its actions indefinitely. These actions consist of reading the data at the port once each period and pushing it into an internal FIFO buffer. Therefore, the history process provides an external periodic mechanism to which sync line actions can be referenced. Given this, the establishment of the correct read / write behaviour was straightforward. Whenever an executing SLIDE statement attempts to read a sync line, that action will be rescheduled on the SIMULA event list just after the history process for that line. Whenever an executing SLIDE statement attempts to write to a sync line, that action will be rescheduled to just before the history process.

As an illustrative example, consider SLIDE devices A and B in the simulator library, each possessing a synchronous line of the same periodicity. The user creates one copy each of A and D , and connects the respective ports together by wire in the simulator. Say that A is executing a section of SLIDE code of the form :

```
LOOP 1000 TIMES DO
BEGIN
         x - r   NEXT
         r ← r + 1
END
```

where x is the sync line and r is a local register, and say that B is concurrently executing this piece of SLIDE code :

```
LOOP 1000 TIHLS DO
BEGIN
         bh - y   NEXT
END
```

where bb is a SLIDE buffer and y is the sync line. If the declared periods of y and x are c time units, then a snapshot of the SIMULA event list would look like this :

Process	Ascociatod Timo	
A	nc	1
AY, hislory	nc	| Order Of
B'& history	nc	| Execution
B	nc	V

where n is an integer. The ordering of the history processes relative to one another is arbitrary. Ar; each procf?',s completes its loop actions it immediately attempts another access to its sync line. This will be rescheduled to time (rwl)c. Thus by automatically delaying accesses to synchronous variables until they line up with the associated history processes, the desired periodic behaviour of sync lines is achieved.

To complete this discussion of SLIDE hardware , the SLIDE buffer, the SLIDE table (associative memory) and SLIDE combinational logic will be touched upon. The SLIDE buffer implementation is similar in principle to that of class hard, except that bit transition checking does not exist, and a fifo queue data structure is implemented to accomodate buffer read / write semantics. SLIDE buffers will set a global Boolean flag whenever an attempt is made to read them while empty or to write to them while full.

SLIDE tables arc implemented as subclasses of a prefix class table. The principle of operation of the SLIDE table implementation is to have two integer vectors store the defining

mappings . Thus LNCOOing and DFCODing are equally easy. SLIDE tables will set a global Boolean (lag whenever an attempt is made to encode or decode a data value which did not exist in the defining mapping.

SLIDE combinational logic is also implemented similar to class hard, except that it is not a SIMULA process, and it is a read-only resource. Whenever a read is performed, the value of the combinational logic resultant is updated according to the current input data to the defining expression. It is not the case that the combinational logic output values change whenever an input changes. These updates only occur when the combinational logic itself is accessed . This of course saves considerable overhead over the more simple-minded method, and is just as accurate from the simulation point of view.

SLIDE combinational logic is not implemented as a process because it doesn't need to be ; the variables that it uses in the defining expression , which must eventually boil down to lines, registers, and buffers, are aware of their indirect memberships in expressions as inputs to combinational logic as well as their direct memberships. Hence, the variables will contain in their special vectors the necessary entries in procedure check. Since whenever check is actually invoked it v/ill attempt to access the given combinational logic in the expression, the required updates will always be performed in time.

4.4.2 Subroutines

The basic mapping for a SI.IDt:. subroutine is onto a SIMULA procedure. This is straightforward enough, but there is one interesting point concerning their implementation. A SLIDE subroutine can be called from anywhere within the SLIDE process in which it was declared. In addition, <\ny of the subprocesses of the defining SLIDE process may also call the subroutine. This posed a problem that relates to the cataloguing of processes and Boolean expressions in procedure check in the SLIDE scheduler. Beforehand, each expression in check could be identified a *priori* with a unique SLIDE process. This is clearly not the case with an entry in check that corresponds to a DELAY statement expression when the DELAY statement occurs in a SLIDE subroutine.

The solution to tho problem entailed extending the cataloguing in check and the mapping of DELAY statements to provide for *variability* in the identity of the affected SLIDE process during device simulation. Il ako entailed providing in the mapping of SLIDE subroutines some bookkeeping facilities to keep track of subroutine calls, so that virtually all cases could be correctly handled. (For example, concurrently executing SLIDE processes invoking the identical subroutine and executing the identical DELAY statement.)

4.4.3 Stalomonts

The SIMULA mapping of the CALL, NOP, BR, and IF statements in SLIDE are all straightforward and intuitive, and will not be elaborated upon here. The IFERROR statement is just an IF statement that is Keyed on a global Boolean flag. This flag is set whenever a SLIDE buffer is improperly accessed, and whenever an attempt is made to ENCODE or DECODE data that not in the specified SLIDE table. Execution of the IFERROR statement clears the flag. The more interesting mapping tasks v/ere those for LOOP and DELAY statements, and for parallel compound statements.

The LOOP statement mapping would have been trivial except that the SIMULA WHILE statement turned out to be unusable. This was due to the fact that SLIDE expression evaluation in SIMULA requires the execution of a number of SIMULA statements , in general. Thus SIMULA IF statement constructs had to be employed to achieve the desired functioning. For example, *LOOP 100 TIMES DO <stmt> UNTIL <cxprcssion>* maps into :

```
FOR i:=I STEP J UNTIL 100 DO
BEGIN
        <51mt>
        <Rxprcv;sion evaluation until Boolean resultant>
        IF Lloo lean resultant THEN GOTO exit;
END;
exi t : ....
```

The SLIDE DF.I.AY MMement is tin? pivotal statement of the language. It provides the fine in-line timing control that facilitates the accurate description of I/O hardware operation. While the mapping of the statement *DELAY 100* was trivial, because the SIMULA HOLD statement is semanlicnlly identical , the other two forms of the statement required some thought. For *DELAY UNTIL/WHILE <cxprcssion>* the following mapping was used :

```
label : <r»xpr cr.sion evaluation up to Boolean resultant>
        IF (NOT) Boolean resultant THEN
        BEGIN
                HOLD(very large number);
                GOTO label;
        END;
```

To complete the mapping, the catalogue of Boolean expressions in procedure check of the SLIDE scheduler is updated with *<crpression>* anci the identity of the process in which the DELAY statement occurred. From the above example it is seen that the DELAYed process is *never* actually removed from the SIMULA event list, just placed very far down it in simulated time. While this forcer, an otherwise unnecessary GOTO loop to be included in the mapping, it does serve to provide an easy distinction between processes that have not yet begun

execution (then nol on lho event list) and OfXAYed processes. The SLIDE scheduler uses this distinction to great advantage.

The final version of the DELAY statement, *DELAY 100 UNTIL/WHILE <cxpression> ELSE <stmt>* resulted in a more complex mapping :

```
xi=cuiTcnt r, iimilotcd time;
l abe 11  : <PX|>l .r"-°.i on evaluation up to Boolean resultant>
          IK (NOD Boolean resulUnt THEN
          BEGIN
            liriL.rU (100;.- SLIDE CLOCK) + x - current simulated time);
            IT x i (lOOvr SLIDE CLOCK) <> current simulated time
          • TURN
            BLG1N
                             <stmt>;
                             GOTO l oho 12;
            END
            ELSE  GOTO label 1;
          END;
labe12: .....
```

To understand lhir. mapping, it is important to keep in mind that this, code will be appearing inside what is actually a SIMULA process. As the code is entered, the variable x is set to the simulated time at that point in execution. If the expression evaluation results in the IF statement condition being true, then the execution of this SIMULA process must be suspended. This is dono by invoking lho SIMULA command HOLD, which freezes execution and schedules the process to be resumed *100* M- *SI IDE CLOCK* time units from now (since x = current simulated time) . When execution resumes , it will be at the statement following HOLD.

Upon resumption of execution, current simulated time v/ill be greater than or equal to x. If it is also greater than or equal to *x • 100 \: SI.IDC CLOCK* , then the DELAY statement has "timed out" and the statements *<stmt>* must be executed. If the DELAY statement has not timed out, resumption of execution could hnvo taken place one of two ways. Either the IF statement statement condition has indeed gone hue, or an uncalled-for reactivation has taken place. The latter possibility does exist because of the nature of the implementation of procedure check in the GLIDE scheduler,[1] which does not discriminate between DELAY statement expressions in an executing SLID!" process. In either case, looping back to label 1 produces the correct behaviour. In the fiisl case, the program will drop through to label2. In the second ense, the process will be suspended for the amount of time left until a time out

[1] A molhod for nlloring lhn implnmonlnlinn of procedure chock to plnminnto this popmbilily is montionod in (ho conclusions.

would occur, given the original limo al which lho DELAY began.

The mapping of parallel compound statements is intended to provide "FORK and JOIN" behaviour. Given the following example :

```
BEGIN
          ......
          ......
          ......
          ......
END       ;
BEGIN
          ......
          ......
          ......
          ......
END       NEXT
<31mt>
```

SLIDE semantics dictate that the actions in these two statement streams be started at the same lime (I ORK) , *nnd* that <slmt> not be executed until all the actions in the two streams have completed (JOIN).

To accomplish thi'., each statement stream is mapped into a separate "degenerate" SLIDE process , drgrnrjrntr? in that it will have no entry in lho priority tree. As an example, given an executing process A in which three parallel compound statements are encountered, name the compound statements 1,2,3 and their respective statement streams A, M, and T. The following mappings would result :

```
PROCESS A :      Antivote 1
                 DELAY UNTIL 1 is finished
                 .....

PROCESS 1 :      Activate 2
                 A
                 DELAY UNTIL 2 is finished

PROCESS ? :      Activate 3
                 M
                 DELAY UNTIL 3 is finished

PROCESS 3 :      T
```

After the activation statements take place , processes A, 1, and 2 are suspended at the same simulation time, and process 3 begins execution at that same simulation time. By virtue of the DELAY statements, process A cannot resume execution until all of 1, 2, and 3 have

completed their action'.. Thus the FORK and JOIN behaviour is achieved. Note that as separate processes, the throe statement streams do indeed execute autonomously and in parallel. The so-called DELAY statements shown in the example are meant to signify the form of the resulting SIMULA mappings . For instance, each of A, 1 , and 2 will have an entry in check, and their associated expressions will be " 1 (or 2 or 3 respectively) is finished". The same mechanisms will be used to reschedule the processes as would be used for a real SLIDE DELAY statement.

As a closing nolr, this mapping for parallel compound statements also includes an accounting mechanism for the degenerate processes. This is necessary to insure correct SLIDE process termination. In the above example, if A was to be terminated, the accounting mechanism would insure that processes 1, 2, and 3 would be terminated as well.

4.4.4 Timing

The liming task in the simulation of a 51 IDE description was considered, and maintaining time fidelity bolwrmi communicating 51 IDF processes was seen to be crucial for interconnection simulations. Fortunately however, it was found that in general it is not necessary for *every* action in a SLIDE process to occur at the absolutely correct simulation time, because not every action is necessarily significant . A significant action is said to occur whenever any SLIDE statement that could possibly cause the SLIDE scheduler to be invoked, or could produce activity at the ports of the SLIDE device, is executed. It was decided to include in the code generator an algorithm that would insert correct delays at proper points with respect to significant actions, to reflect the true passage of time. By not having to insert delays after non significant actions, the simulation was expected to correspondingly speed up at no cost to modelling accuracy.

The sequential passage of time in the body of a SLIDE process is embodied in the two statement separators MFXT and ; (semicolon). The semicolon indicates that no time is to pass between the completion of the previous statement and the start of the subsequent statement. NEXT calls for the insertion of a delay lasting one SLIDE CLOCK period between those two points. If all statements were treated equally, the mapping of time passage would simply involve the insertion of a SIMULA HOLD statement wherever a NEXT appears in the SLIDE description, and inserting nothing wherever the semicolon appears.

The desire to reduce the number of MOLD statements that occur in the generated code makes the translation more complex. Each SLIDE process will be assigned a Boolean flag that is to be true only if the statement just executed was significant. There is also an integer counter that is used to keep track of the number of SLIDE clock periods that the simulated

time of the cxreuting process if. off hy, by virtue of having ignored r.omc NdXT's. Now , the code generator dotorminr^ the significance of each statement in a process ,[1] as defined above. If the statement *to be executed* is significant, it needs to be executed at the correct simulated time, and so a corrective delay is inserted as follows :

```
IF counter \ 0 THEN
BEGIN
        HOLD(counter * SLIDE CLOCK);
        counter := 0;
END;
```

After *any* statement completes its execution , the code generator inserts code to set or clear the Boolean flag as appropriate to the statement just executed. Then the following code will be inserted if a NEXT was encountered after the statement in the GDB :

```
IF fitKi is iran iiii.N
BEGIN
        HDI D( (counter +1J * SLI OF CLOCK);
        count or :~ 0;
END
ELSE
        counter :~ countnr + 1;
```

Nothing is inserted if a semicolon was encountered.

With the insertion of the above code as necessary before and after SLIDE statements , the overhead incurred duo to SIMULA HOLD statements is minimized consistent with time fidelity requirements.

4.4.5 Expressions

The implementation of SLIDE expression evaluation centred on just a few concepts. By and largo, this implementation was able to make use of the power of the SIMULA language to produce a flexible mapping that simplified the job of the code generator in producing executing expressions.

The fundamental operand in SI IDL K the bil-r.rgmcnl, i.e., a string of bits. There is no inherent limit in SI Il)f" to the width of bit r.rgmrnlo or to their accessibility. The mapping of bit-segments into SIMUI A had to retain these properties, and this was achieved quite simply in class *Inter.* Class inter provides a standardised representation medium for SLIDE data and forms the basis for evaluation of SLIDE expressions. An inter object is a pointer to a one

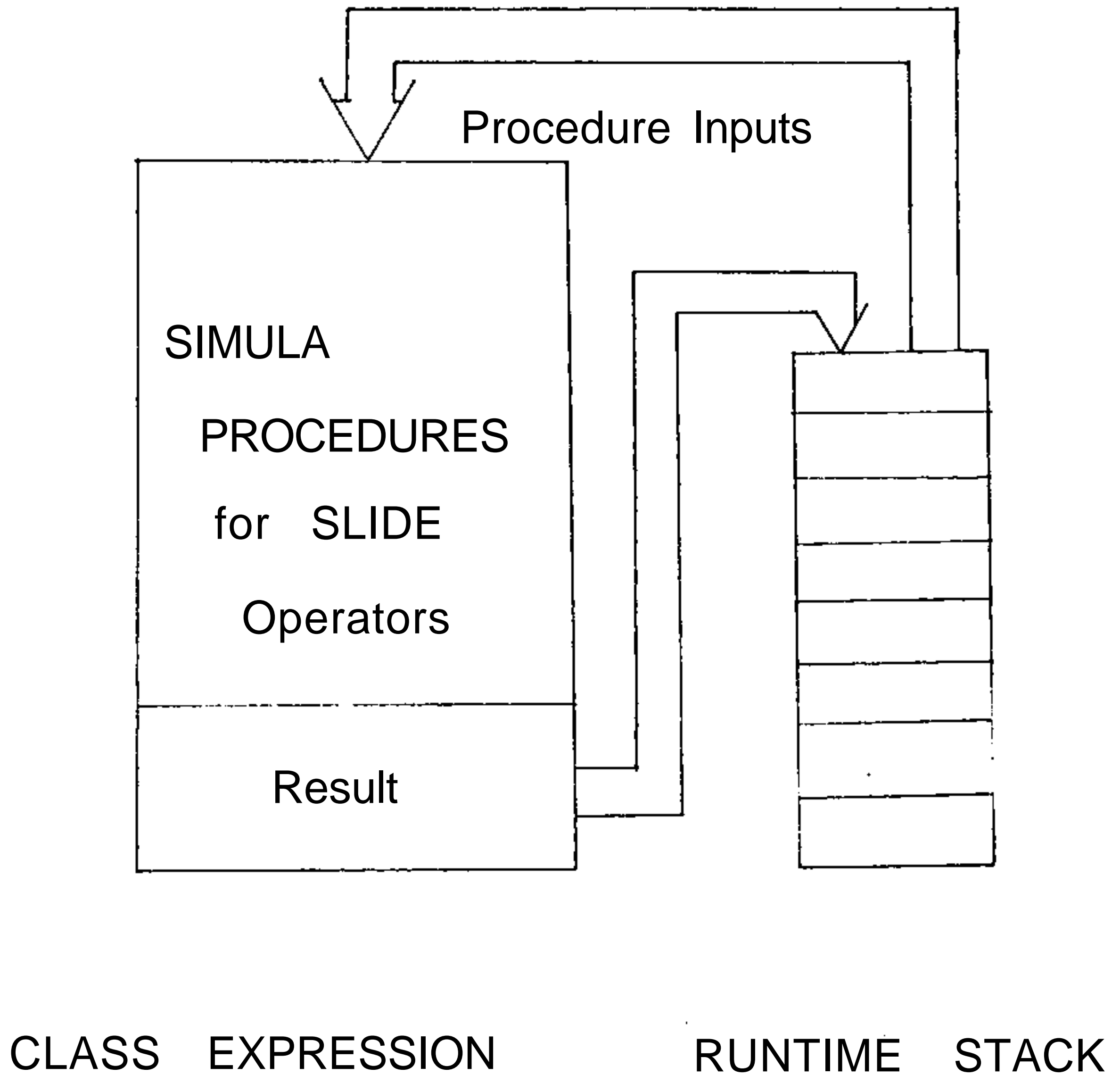
[1] Section 5.3 doRcribos tho problems of dolormining slalamonf significance

dimensional array of integers. Storing the data in an array removes most practical limits to its scope[1]. Referring to the array through a pointer gives flexibility in operand manipulation at the cost of extra storage. Most arithmetic and bit-logical operations are implemented to accept inter objects as inputs and to produce inter objects as outputs.

The mechanisms for expression evaluation are twofold, and are illustrated in Figure 8. First, there are the various SIMULA procedures that perform the actual SLIDE operations. Second, there is a runtime stack for storage of temporary results. The SIMULA procedures co-exist inside an environment called class expression. Class expression, in addition to housing the procedures, also houses a single inter object, into which operation results are placed. Class expression will interact with the runtime stack to produce efficient and correct expression evaluation. The runtime stack will push and pop inter objects. It can also interchange the two objects at the top of the stack, which is useful for correctly ordering operands.

[1] Currently, one SLIDE bit is mapped into one integer ; bit-stuffing macros will be employed in the future to produce a bit-to-bit mapping

Figure 8: SLIDE Expression Implementation



One expression objocl n»>d ono runtime st»vk will exif.t per SLIDE device instance (they are located in the SP). More than one set is not necessary for a given SLIDE description. Limiting the number to only one set in the whole simulator would open up the possibility of inadvertent data contamination, because of the autonomous and independent nature of the ongoing device simulations. The decentralised nature of the simulator core precludes the kind of centralised control of expression evaluation that would be required otherwise .

To close out this overview of SLIDE expression evaluation, the implementation of a unique expression will be clir.cusr.rd. This expression had to bo realised completely outside the class expression environment. This was the *sequential behaviour expression,* which takes the form *X EQL :V1:V?:V3: ... :Vn:.* This expression will bo true if, at the simulated time of its evaluation, the last n valuos on Iho synchronous line were* first VI followed by V2 and so on to Vn. The expression is falso otherwise. X must be a synchronous line, and the values Vi are constants.

The evaluation of Ihis expression entails Knowing the exact behaviour over time of any sync line, to within some maximum time window '.i/e. Recalling the implementation of the sync lines, it is soon that tlu* hi-.lory ptocrs'. of a sync line, through its internal buffer, provides exactly this rosourm. Tho history procoss maintains from time zero a record of occurring values on tho lino. Thus, one component in sequential behaviour evaluation is provided by the relevant history process.

For each occurronco of a sequmlül hrhaviour expression in a SLIDE description, there will be mapped ono ohjoc.t of class sequonco. Whenever the expression is evaluated, the actions of the sequence object will be invoked.

The actions of the sequence object are split into a transient phase and a steady state phase. In the transiont phase, the sequence object simply waits for enough values to have occurred on tho lino to bogin actual testing for Iho given sequence. Until simulated time exceeds n timer; tho sync line period, there is nothing to test for, and the result of the expression evaluation is always false.

After enough timo has passed, tho sequonco object will begin its steady-state phase. Upon invocation, its actions aro to do a word by word comparison between the sequence to be tested for and Iho List n values on tho line ns recorded by tho history process in its buffer. The elapsed timo between successive expression evaluations is taken into account, and the the location of tho window through which the history process buffer is examined is shifted accordingly.

Thus, the evaluation of this typo of expression is entirely independent of class expression. Each occurrence of a sequential behaviour expression results in a new sequence object being

created in the SP of the SLIDE device module. This differs from the conventional operator implementation in that , for example, no matter how many AND's occur in a SLIDE description, there will be only one ANDing procedure (inside one expression object) in the SP. However, conventional operators have no need to consider the time between successive invocations ; dependence on the passage of time necessitates a more complex implementation for sequential behaviour expressions.

5. SLIDE Code Generator

5.1 Context Of Operation

To fully understand the operating principles of the code generator program, it is necessary to understand the overall context of its operation within the software system described in Chapter 3.

5.1.1 Pre-Compiled Code

The mappings from SLIDE to SIMULA discussed in Chapter 4 were expressions of the nature of the final SIMULA code. Exactly how they are supposed to come about, or where they fit within a framework of automatic program generation was not discussed. To a first approximation, desired SIMULA mappings can be broken down into two categories with respect to the code generation problem : code that is generated anew for each SLIDE description, and code that is always present in the multi-level simulator. The relationship between the two is symbiotic ; the always-present code is a continually existing resource that the code generator program will assume to be available to the code that it generates.

The domain of the always-present part of the SLIDE to SIMULA mapping consists of the generally used SIMULA prefixes upon which specific instances of SLIDE entities are built, such as class hard, and globally ur.ed procedures and variables. Conveniently, SIMULA provides a facility for allowing classes, procedures, and data structures to be independently pre-compilcd into one new *external* prefix. Any program that is prefixed by this external class will have access to the constituent properties of the class. This methodology was used to implement the always-present part. In the actual implementation, general support code for the multi-level simulator core is ak>o included in the makeup of the external class. The advantage of all this is lies in that, for example, class hard has only to be written once and compiled once to bo accessible to all subsequent SLIDE simulations. The precompilation facility removes the need for the code generator to be constantly reproducing such code each time it is executed.

5.1.2 Preprocessor

Another clement in the operational context of the SLIDE code generator is the preprocessor. As was previously stated in Section 3.1, the preprocessor pieces together fragments of SIMULA code from any number of code generator output files into a single SIMULA program. In the original implementation of the simulator core [8], a simple program existed to piece together the various device modules into a runnable simulator. It was

designed to allow a user lo write hio own device modules in SIMULA without having to worry about gross syntactical drUiils. The current preprocessor takes this concept and expands on it to provide a fAenei^T| tool for the formatting of fragments of text.

A notion of the world-view of the preprocessor can be obtained from the following analogy. This Report can be considered to be the root of a tree, whose sons could be "Chapter 1", "Chapter 2", and so on, and each of these could have sons "Section 1", "Section 2", etc. The relationship of the "real" report to the tree is that the tree can completely specify the ordering of the different parts, subparts, and so on of the report, such that the report could just ns well be read by traversing the lrce (in the correct fashion) as by any other means. Now, if the tree were stripped to the bare bones and retained, such that each node contained- only a label such as "Chapter 3" or "Section 16", and the various textual pieces of thn rrpoi t wore mixed up into a random pile of fragments, with each fragment tagged "Sulv.ff lion / of Section q of Chapter d", it would still be possible to read the report in order. The tiro could be used ns a guide by which subsections could be synthesised into sections, sections into chapters, and chapters into a report, without paying any attention to the semantics of the lag names, only lo their location in the tree.

The preprocessor is r« program that tries to apply the principles (hopefully) conveyed by the above analogy. If a general body of text is able to have a relational structure specified for its constituent sub-texts , then, given only the sub-texts - suitably labelled - and a specification of lhe structure, the preprocessor program will recreate the original text body out of the input sub-texts. In particular, lhe preprocessor expects the structure to be specifiable as a tree, and it will piece together the text by following a post-order traversal of the tree. Thus, it combines the text specified by the sons of a node before moving on to the right-hand brother of the node.

The requirement for tree-specification of tevl body structure is not a problem for this application of lhe pmprocessor, namely the creation of one SIMULA program from a multiplicity of input text files. As a high-level programming language, SIMULA has a specific syntactic structure lo adhere to which is quite amenable lo a tree-like representation.

Incorporated in lhe text input to the preprocessor will be commands that will guide its Operation . Them nro four such commands : a tree structure specification command , a text labelling command , n premature tree lravrrs.il command, and a file inclusion command . The purpose of the former two commands should he clear from the above discussion. The latter two commands will be described next.

The premature tree traversal command forces the text associated with the subtree of a node in the structure tree to be combined into a single fragment, and tagged with the node

label. To sor I ho ulility of such a comm.md, consider a simple example. Assume that computer programs in some language have boon structurally described as a single level tree, i.e., as consisting of *Declarations* and *Actions,* Given n fragmented programs, where each program is presented a% a piece of code labelled *Declarations* and a piece of code labelled *Actions,* to bo input together to the preprocessor, the desired output would of course be n single programs. However, if the tree traversal process were to run continuously from beginning until end, the resulting body of text would be an attempt at one large program, with the n declaration sections of the n programs grouped together followed by the n action sections. This is because each occurrence of a given label is the same from the point of view of the tree ; any two text fragments that are labelled identically will have, by definition, no predetermined ordor with respect to *each other,* only with respect to differently labelled fragments. Thus, all the declarations and then all the actions would be lumped into an amorphous whole, because? no line was drawn between them.

The premature I roc traversal command, properly invoked , would allow such a line to be drawn botwen programs, so that thn respective fragments could be properly congealed into individual programs.[1] To do this requires a certain degree of physical organisation among the input fragmrnk; thi'. could not be achieved in a totally randomised setting. Thus, the premature tree traversal comancl is not a general replacement for a more detailed structure specification. However, under the right conditions it ir, useful in eliminating the need to overspecify structure in order to achieve the* dosirod results.

"Iho file inclusion command i» simply a drvico for specifying input files to the preprocessor. They can be nrslrd so that groups of input files may be treated as a single entity by the preprocessor. Thus the code generator may safely output multiple fragment files for a single SLIDE device mapping, ao long as it provides the necessary file inclusion commands.

5.1.3 Final Simulator Program

Another element of the operating context of the code generator is the final SIMULA program to which its output must conform. The form of the final program is constant, and can be described as follows :

---

[1] Tho rotative) orclnrmg of program*; in Urn mnuHíng output would ho rnnriom

```
Special 1'lMr.laratinns
Class definitions - Simulator Cor?
Class nnrl Pi nr«Mlui o definitions - SLIDE
Procedure dr»f ini t ionr. - .nimulotnr core
Global varial» lo dofini1ions
Glohal vnri.'iblc initialisations
Main program
```

The simulator core parts of the above program correspond to the non-precompiled part of the core as soon in Figure *2.* This part is entered in fragmented form to the preprocessor , and is fixed. Note that Ihn simulator main program is the front-end monitor with which the user will interact.

Of course, the slot marked off for SLIDE clashes and definitions is reserved for the output of the code gonernior, after rearrangement by the preprocessor. Moving in for a closer look, it is soon that the ultimate structure of the code that is produced by the code generator will be as follows :

```
SP ( Simulation Process )
Chain
Spniir-rj Pi orp'/.n-, (pi nrp'.',r'^ derived from parallel compound stmts)
Procedures -itir1 Prnconr.eo (SLIDE subroutines and processes)
```

Each of the above sections corresponds to a node in the structure tree that will be passed to tho preprocessor./[1] I he particular partitioning is more a reflection of certain code generator output logistic requirements than of anything else. For instance, the relative ordering of class *wul* procedure definitions is immaterial in SIMULA ; many other arrangements would have been equally suitable in this sense.

To achieve this structuring in tho final preprocessor output, the code generator uses the full arsenal of preprocessor commands. For example, the premature tree traversal commands are utilised to insure that the above sections are not inadvertently grouped with the corresponding sections of other SLIDE device mappings.

---

[1] Thin r.lruelurn in fixorl for nil SI 101. rlnvicon, HO Il»O currr.porwlinp prrproc*»nn^r commando »r© pariaod in **Iho** non-prncompilorl *par* input filn *\n* r.p.iro Ihi» corln gonor.ilor having to do it oner for nach SHOT ditvico, which is clonrly unnncoTi.iry.

5.2 Discussion of Operation

The code generator program is designed to operate within the context described in the previous section to produce SIMULA code that will eventually form SLIDE device modules within the simulator. Its input is a single file containing the GDB tree of one SLIDE description. The code generator traverses the tree exactly one time, producing SIMULA code on the fly as required. Suitable labelling of this code and proper insertion of commands to the preprocessor assure that the output will be correctly rearranged into the intended format.

5.2.1 Uniqueness of Variable Names

Upon invocation by the user, the code gcnrralor will ask the user for the name of the GDB file, and the name by which he or she wants the resulting SLIDE device type to be known in the simulator. Among other things, those names arc used to insure uniqueness of the resulting SIMULA class and procedure names in the final simulator program.

The effort to minimise the chances of vaiinhlr name clashes in the simulator is based first on the fact that inside a SLIDE description, identifiers musl Lie unique across a given process level only. Thus, by appending process Irvrls to identifier names, the code generator prevents n.ime clashes within a given SI IDF. drvicc in the simulator. To deal with the problem across devices w/r, not as straightforward. It was decided to base SIMULA class and procedure nnmrs o\) the user inputs described above. If the user names different SLIDE descriptions uniquely, i.e., does not use the same file name or module name for different SLIDE devices, then lliesc names can be used to insure complete uniqueness of all identifiers in the simulator. As long as the u'.or is made aware of this minor constraint , there should be little ensuing difficulty.

5.2.2 Procedure Interaction and Related Comments

The operation of tho code generator is summarised in figure 9, which diagrams the major procedures and their interactions with each other and other facilities. By way of explanation, each of the pictured procedures is briefly described herein.

Initialisation	1 hi', procedure initialises glnh.il variables, dumps beginning code for the $P and I lip Chain, and initialises the low-levrl utilities (creates the symbol table, opens filr.% etc.) .
Finalisation	lhis procedure dumps lhr» code to generate SLIDE hardware variables plus the final code for lhr SP and the Chain, and wraps up the low-level utility chores (closes files, outputs iilNCLUDE commands, etc.).

PROGRAMgen	This procedure does the bulk of the SIMULA mapping for the SLIDE description MAIN process.
PROCESSgen	Generates the SIMULA mapping for the body of any SLIDE process. Since SLIDE processes are nested inside each other, PROCESSgen can invoke itself.
Mainset	Handles the mapping required for the case when the current process is the MAIN process. It essentially does what INITgen would do if there were an Init statement for MAIN processes in SLIDE.
DECLARgen	Handles any declarations that may exist in the current SLIDE process by calling the four procedures TABLEgen, CLOCKgen, COMBgen, and HARDgen, as indicated by the current node in the GDB tree. TABLEgen, CLOCKgen, and COMBgen create the code needed for SLIDE table, clock, and combinatorial logic declarations, respectively. HARDgen generates the SIMULA declarations for SLIDE hardware variables (cf. Finalisation).
INITgen	Encountering the Init statement for a SLIDE process in the GDB tree causes INITgen to be invoked. This procedure will dump the SIMULA code that will cause a new SLIDE process to officially exist as a data structure although its body has not yet been specified, viz. enter a new node in the priority tree , a new entry in procedure check, and so on.
SUBRgen	Dumps the beginning and end code for any SLIDE subroutine that is encountered in the GDB ; STMTSgen is invoked to handle the body of the subroutine.
STMTSgen	This procedure generates the SIMULA code for a sequence of SLIDE statements.
STMTgen	This procedure generates the SIMULA code for a single SLIDE statement, by invoking one of the ten procedures diagrammed, as appropriate. For example, LOOPgen generates the code for LOOP statements , and so on. A sequence of SLIDE statements bracketed by BEGIN, END is treated as either a parallel compound statement (Pprocessgen), or as an ordinary sequence of statements (STMTSgen), depending on the context.
Expgen	This procedure generates the appropriate SIMULA mappings for all SLIDE expressions. Since expressions are often found embedded in other expressions, expgen is recursive.

Figure 9: Procedure Interaction In The Code Generator

Code Generator

Initialisation PROGRAMgen Finalisation

PROCESSgen

Kaituu ·t Dr-CLAii^en F^OCLio.S.^en 1K11gen SUL.Rp;on STMTSgen

HARDgen TABLEgen CLOCKgen COMEgen STMTgen

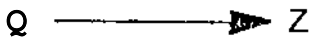
EXPgen

Pprocessgen

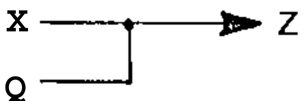
NOPgen CALLgen DELAYgen ASSIGNgen IFgen LOOPgen IFERRORgen BRgen

STMTgen

EXPgen



Q calls z



X & Q each call Z

All tho cJi.ii',r;immr(J procedure", interact with the low-level resources of the program, namely the symbol table, and the input and output utilities. The symbol table is the repository of basic information about the various entities of a SLIDE description. DECLARgen, INITgcn, and SUBRgen will enter the variable names and relevant parameters of SLIDE hardware, etc. into the symbol table table as they encounter them. The information in the table is both updated and used by other procedures during their code generation tasks. The input utilities contain the data structures and inputting primitives necessary to support GDB tree file access and traversal. The output utilities provide corresponding primitives to handle the output requirements of the code generator, such as simultaneous outputs to multiple files and delaying output of generated code by use of text stacks.

Not surprisingly, the '.truelure of the procrdurcs as shown in Figure 9 corresponds to that of the GDB of a general SLIDE description ; it reflects the tree traversal operation that is the driving component of the code generator program. The GDB provides another benefit aside from a structured representation of SLIDE descriptions , which is that the code generator is free to base its actions on the assumption that the original SLIDE description was syntactically toirrcl. This is a simplifying factor in many instances. For example, in the case of an unconditional branch (BR) to a labelled statement, where the program is unable to find the label in the symbol table, it goes ahead and generates tho SIMULA mapping anyway ; the label must exist, otherwise the original description would never have gotten past the SLIDE compiler.

The syntactic correctness assumption is one factor that allows the code generator to be implemented ar. a one-pass program. In the above example, a second pass would have been necessary to resolve the question of the label's existence. The preprocessor is another factor in preventing multiple passes through the GDB. It gives the code generator the freedom to output any code it deems necessary, at any stage its execution. Thus there is no need to ever "wait until the next time around" to perform a code generation task because "it's too late to do it now" .

5.3 Points Of Interest

In this section, some of tho difficulties **lli,«t** arose during the solution of the code generation problem will be described, and example occurrences of each will be briefly discussed. They all arise out of I he following observation : although the software tools exist to *physically* provide the code generator the freedom to generate any category of SIMULA code at any point in its execution, there are still limits to what can be accomplished at any One time. These limitations result because of the nature of some of the SLIDE to SIMULA

mappings, and because of insufficient information during the tree traversal.

The above points can be itemized along the following lines . Having arrived at a given point in the GDB tree, it is desired to perform a certain code generation task. However, it is possible that

- Some of the code that is to be generated must be saved up and output at a later time because of the nature of the SLIDE to SIMULA mapping, or

- There will not be enough informalion to do the task until the entire GDB tree has been processed, or

- There will not be enough information to do the completed task during any part of the codn generation phase , only at runtime.

The first stumbling block is really a consequence of the fact that the preprocessor is only useful in structural specification of lext. While SLIDE statements are being processed, *behaviour* is being described ; there can be no structural ordering that says that all DELAY statements must come before all CALL statements, for instance. Thus , when a non-sequential mapping arises for a SUOf statement, there is no recourse to structural semantics to make the output sequential ap.jin. An example where this comes up is in the parallel compound statement mapping (Section 1A3) where the body of a spewed process is to be bracketed by extra SIMULA code (Activate and "Delay"). This mapping is achieved through use of a text stack facility that allows tho "Delay" section to be saved and automatically dumped later on at the appropriate point. Only the Activate part is immediately written by the program ; the rest is put on the stack and the program moves on. The preprocessor could not be used to get around this case because no adequate structuring is definable.

The remaining two stumbling blocks deal with the more fundamental question of desired task vs. information vs. time. That is, given the desire to perform a certain code generation task, there will exist cases where that task cannot be fully performed by the code generator because of a lack of information at the time, which will necessitate waiting either 1) until the whole input GDB has been processed, or 2) until runtime. Although the particular cases cited next are peculiar to this project, this should not lake away from the fact that these are basic issues of the problem of code generation that managed to surface in spite of the power of the operating environment.

An example of case 1) that arises is SLIDE hardware variable generation. One of the parameters of class hard is an integer that specifics tho length of the list of procedure check entries for that variable (Sec Section 1A1). This parameter is named ma*. When the declaration is arrived at for a SLIDE hardware variable in the GDB tree, max is not known. In

fact, the value of mav is not finally cletcrminnd until the entire SLIDE description has been processed. The operational consequence of this is that as described previously, procedure HARDgcn only produces the declarations for SLIDE hardware ; the actual creation of SLIDE hardware variables is delayed until the end, when procedure Finalisation does it.

The following example of case 2) involves the handling of timing between SLIDE statements. As was mentioned in Section 4A4, the code generator is to determine the significance of any given SLIDE statement. (Any SLIDE statement that could cause the SLIDE scheduler to be invoked or could cause activity at the ports of the SLIDE device is said to be significant.) The exact time at which this determination can be made varies with the statement, however.

If each SLIDE statement is surveyed, it is seen that the significance of DELAY statements, NOP statements, and [3R statements is unchanging (the first two are always significant* , and the last one is never significant), and so the code generator will output the code that sets or clears the Boolean flap, as required. The significance of LOOP, IF, IFERROR, and CALL statements is determined solely by the SLIDE statements that they cause to be executed; they have no "infringe" significance. In these cases, the code generator will not insert any code, but just lot thn current flag value pass through as is. The significance of the SLIDE assignment statement is a function of the particular SLIDE hardware variable that is being written to. If the may parameter mentioned earlier is zero for a local SLIDE hardware variable that is being written to , the code generator cannot make a decision one way or the other. It m.iy be that later on in the SLIDE description the variable will be involved in a DELAY or INIT statomcnl, and so max will become non-zero, but that can't be told as yet. Since it is impractical tr. hold *up* the generation of assignment statements until the finalisation part, it is left to the executing SIMULA code to determine significance in this case. Therefore, this is an example where a certain code decision is actually deferred until runtime.

If the code generator has to defer the decision to runtime, the code that is normally inserted before a significant statement will still be inserted, but bracketed by the following :

```
IF the variables max parameter \ 8 then
BEGIN
            .....
            .....
            .....
END;
```

Since the significance of a statement is corv.tant in a SLIDE cicscription, having to defer the

* NOP io considored nignifiennf bocaupo lhc only Fomanlic ©ffeel of NOP NEXT is lo advance limo.

decision until runlimo gnve slightly inefficient results. In retrospect, this is one case where having a two-pass code generator implementation could have been advantageous.

6. The SLIDE Simulator Test Case

This chapter describes a test run in which SLIDE descriptions were written, compiled, interconnected and simulated in order to demonstrate some of the capabilities of the SLIDE simulator. This rxamplo represents a single point in the space of simulations that can be executed with this facility.

The example involves the following configuration: (see Figure 10)

- The PDP-11 UNIBUS
- A UNIDUS CPU.
- A Peripheral Device attached to the bur..
- A small CMOS memory attached to the bus.
- A synchronous data link connected lo the peripheral device and to a black box - the source of the synchronous data.

The data link uses an SDLC-liko protocol. The peripheral device converts received synchronous data bits into 16 bit words, and writes them to the memory over the bus.

To expedite the lest, the following simplifications were made:

- The CPU consists only of the processor status words and the bus arbitrator.
- The data link protocol is always in information-transfer format.
- No error checking is done on the data link.

The omitted details could have been included and simulated ; SLIDE could even be used for CPU description although it is not intended for that purpose.

6.1 Peripheral Device Description

The SLIDE description of the peripheral device module, which is the most active module in the simulation, will now be focusscd upon. As shown in Figure 10, the peripheral device has two independently executing functional sides. It has an input side, which is responsible for pulling data off the synchronous line according to the protocol *flag, address, data, flag.* It also has a UNIDUS interface side, which receives a 16 bit data word from the input side. The bus side is responsible for transferring data words lo memory according to the UNIBUS protocol.

Figure 10: SLIDE Simulator Tost Case

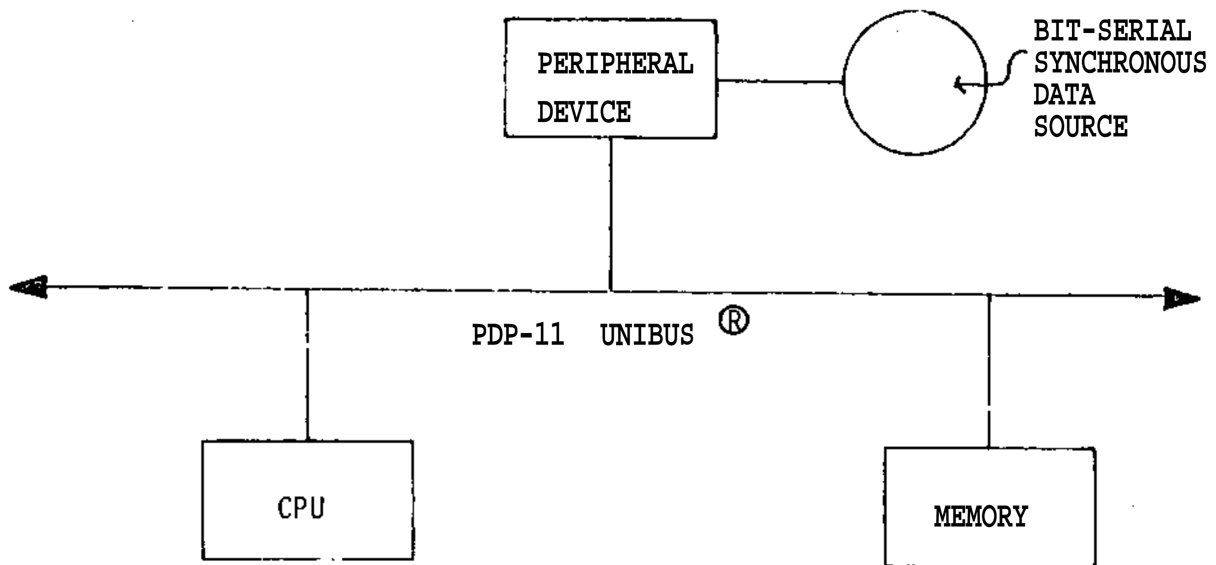
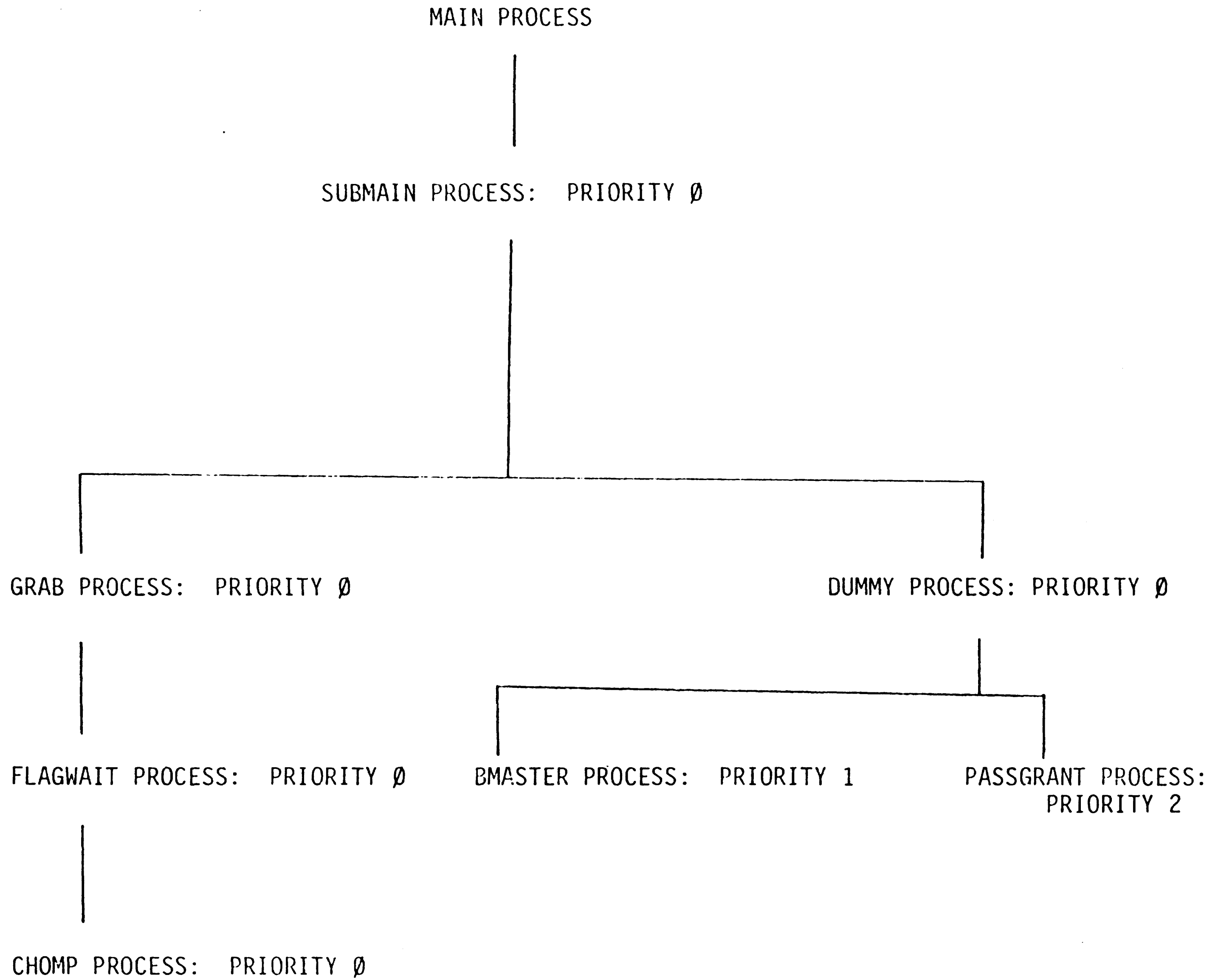


Figure 11: SLIDE Process Structure of the Peripheral Device Model



The operation of this peripheral device war, modelled with the SLIDE process structure shown in Figure 1 1.

On the bus interface side, there exist the two processes BMASTER and PASSGRANT, statically nested inside a dummy process. DMASTER is one level of priority higher than PASSGRANT, which merely passes grants along the bus when the peripheral device has not made a bus request. Thus, PASSGRANT is always active unless BMASTER is active, since BMASTER has higher priority[1] .

When CHOMP has assembled a word of data, BMASTER is initiated and PASSGRANT is terminated. When the transfer is completed, BMASTER terminates itself and PASSGRANT starts again.

The SUDMAIN and DUMMY processes were used to encapsulate local SLIDE variables; DUMMY isolated the bus interface side of the peripheral device (BMASTER, PASSGRANT) from the data link side (GRAB, FLAGWAIT, CMOMP). MAIN, SUBMAIN and DUMMY contain no actions except those embedded in subprocesses; thus they are "active" but asleep (not in a busy-wait). This is achieved by using the "DELAY WHILE 1" instruction, which puts an active process to sleep indefinitely.

GRAB, FLAGWAIT *and* CHOMP input data off the synchronous line, delete inserted zeros and detect flags and addresses. When the peripheral device address is recognized, the bus interface process is modified and 16-bit data words are assembled. BMASTER requests the UNIBUS when data begins to be assembled.

Other devices connected are BLBOX, the black box that generates data for the synchronous line, and DELAY, a generalized delay gale with a variable delay parameter, logic type and output bit width. In tho example, a 75 nanosecond delay of open-collector logic type was created to simulate the UNIBUS skew on the MSYN and SSYN lines.

6.2 Summary Of Simulation Test Results

The simulation run began by instantiating and interconnecting SLIDE modules. Then, the simulation was started and values were traced. Tho results of the simulation are now summarised ; a fuller discussion of some of the results may be found in Appendix II.

The actual test was broken down into three cases. The first case ran the configuration as

---

The initiation condition for PASSGRANT aro "IN1T PASSGRANT WMCN 1"

it was, with the synchronous line data rale just below that of the UNIDUS (as it was modelled). The desired behaviour was indeed observed as data was written into the memory. This is documented in the traces reproduced in Appendix II.

For the second test case, in order to demonstrate some of the utility of the simulator, the synchronous line data rate was adjusted to be higher than that of the UNIBUS. Since the peripheral device had no buffering capability , and data was arriving faster than it could be formatted and shipped to the memory over the bus, the result was a loss of data. In fact, since the new data rate was less than twice that of the UNIDUS, every other data word was lost, as can be seen in Figure II-9 of the appendix. The above two cases ran on the identical SLIDE device descriptions. The synchronous line data rate, being a simulation time parameter in the original SLIDE descriptions of the black box data source and the peripheral device, was adjustable without need of rccompilation.

The third caso involved inserting a logical error in the SLIDE description of one of the devices, and so did involve rocompilalion of lh.»t device. Tho third test was run to illustrate the effect of a stuck lino on the UNIHUS. MSYN was stuck high and then the memory device attempted a read, and raised SSYN. The peripheral device operated normally until it delayed waiting for SSYN to be lowered. The memory held SSYN and waited for MSYN to go down; MSYN was stuck, and the bus hung up. The results of this are seen in Figures 11-10 and 11-11 in the appendix.

7. Conclusions

This report has described the design and operation of a new simulation facility for evaluating I/O and interfacing strategics . The rhetorical task of this report is to be completed by a brief comparison.of the SLIDE simulator with SARA and with SABLE, and by some recommendations and speculations concerning the simulator as it now stands.

In Section 2.4, a characterisation of SARA and SADLE was was presented in terms of the hardware simulator taxonomy of Section 2.3. While SABLE and SARA were found to comprise distinct leaves on the taxonomic tree, the SLIDE simulator is perhaps best described as a hybrid. This results from the way the SLIDE simulator came about, i.e., as a simulator embedded in another simulator. However, if one takes the view that taxonomies should be free of hybrid leaves always, then one might claim instead that the real problem lies with this "inadequate" taxonomy. Nevertheless, much of the nature of the SLIDE simulator is reflected by its location in the taxonomic tree :

- The SLIDE simulator ur.es, for the most part, the SLIDE HDL for behavioural specification. But the user can also hand code digital device descriptions in SIMULA and enter thorn into the simulator core. Thus, there are really two behavioural description languages existing in parallel in the simulator.

- The multi-level simulator core can support simulation of the gate level through the system level of abstraction, all coded in SIMULA. SLIDE- described devices, by definition, are restricted to the register transfer level . So there exists a dual simulation environment supporting on the one hand a number of levels of abstraction using a single descriptive language (SIMULA) and on the other hand a single level of abstraction using a single language (SLIDE).

- Characterisation of behaviour is deterministic in the SLIDE simulator, and structure is informally, interactively specified.

- The degree of separation of structure from behaviour is rather weak in the SLIDE simulator, since structure can be described to a significant degree in SLIDE.

- The SLIDE simulator interactively permits the system under test to be modified during simulation.

- The operation of the SLIDE simulator is process oriented, and its execution can be categorised as compiled.

It should be evident that there is a significant disparity amongst these three hardware simulators. Unfortunately, the determination of which of the simulators is "better" does not really lie in a metric that can he directly applied to the taxonomy. Rather, it has to do with application, and in terms of I/O hardware simulation, which is the focal point of this project, the SLIDE simulator can be seen to be superior to either SARA or SABLE.

SARA is a powerful tool designed to support a specific design methodology [11]. Its generality leads one to conclude that anything that could be described and simulated in the SLIDE simulator could be described and simulated with SARA. However, the task of describing I/O hardware in SARA is surely as complicated as writing a description in some general purpose programming language from scratch. The sum of the GMB and PLIP descriptive vehicles is an abstract, non-hardware relative behavioural description system , with no semantics that relate to I/O and interfacing hardware as such. Such a Petri Net-like description of an I/O bus protocol, for example, would be more a monument to the diligence and patience of its author than an example of a description that both accurately and clearly reflects the operation of the bus and the hardware that interfaces to it. In contrast, the SLIDE language frees the user from having to build up a set of I/O semantics from scratch, and is designed to reflect the operation of I/O hardware.

To a lesser degree, SAF3LE suffers from the same problem as SARA with respect to I/O hardware description and simulation. For instance, in ADLIB, nets are abstract data structures, far removed from the "open collector bus" level of description. The assumption is made that the value that a component believes it has written to a net and the value subsequently seen by the components attached to that net should be identical always. This is fine for abstract ckita structures, however it does preclude the wired-or and wired-and functions common to digital device interconnections; Hill and vanCleempul [14] note that wired-or's on nets mir.t be fudged by inserting dummy components. Another hindrance is that in ADLIB event'. such as updates to nets can be scheduled but not cancelled, so that prospects for succinctly describing such features as system resets are reduced. Finally, the process structuring and semantics available in SLIDE would, in ADLIB, have to be added on artificially with extra data and control structures. This would only serve to obscure the true nature of the hardware's behaviour. Thus, although ADLIB seems to provide the requisite generality to describe I/O and interfacing hardware, its semantic direction makes it less suited for the job than SLIDE.

The SLIDE simulator as it now stands is not a production-quality software package. To make it into one a number of improvements will be needed on the user interface. In particular,

- The style of the user interface provided by the multi-level simulator is towards the software "hacker" , not towards the average engineering user. If a user community is ever to be developed for the SLIDE simulator, the user interface will have to be reevalunlcd and redirected away from the SIMULA programmer and back towards the user.

- As a first level expansion of experimental capabilily, test bed modules should be added to the simulator core library. These modules, connected to the

configuration undor lest, would generate dnla to drive the simulation. This data could ho produced by the modules themselves via interactively specified probability dMrilmtion% or fetched from user-specifiable disk files that contain the data to bo used (trace - driven simulations).

- More interactive tools need to be provided to the user. For example, insertion / deletion of breakpoints and activity counters in SLIDE descriptions, SLIDE process activity tracing, and manual setting / resetting of SLIDE variables.

- More information and simulation results should be available to the user on demand, to bo output to di!.k files if dosired. For example, records of variable activity over limn could be provided, ns could SLIDE process activity logs (including snapshots of the SIMULA event list plus the linked list cool), and subroutine invocation logs. All provided data should be in a formal suitable for statistical nnalysis.

On another piano, the? SLIDE simulator is not an optimised facility. For instance, no attempt was made to provide an optimizer for the code produced by the code generator. More fundamentally, the St.IDf. to SIMUI A mappings themselves have not yet undergone any sort of performance evaluation and upgrading. In retrospect, not every mapping managed to provide the most efficient operation j the highest concern at the time was to "get it working". An example of this is the mapping of SLIDE DELAY statements described in Section 4.4.3, where it is possible to have processes unnecessarily reactivated only to be immediately put back to sleep. This could have boon prevented by assigning a boolean flag to each DELAY statement entry in procedure chock that would be set if the related DELAY statement was currently in force. Inspecting this flag would prevent the unnecded activations, thereby saving SIMULA scheduler overhead which is not negligible.

When considering the optimisation problem , one cannot avoid the attendant problems of the SIMULA language itself. SIMULA is an extraordinarily slow and inefficient language, and the version that exists on the CMU PDP-10*s is rife with runtime bugs. SIMULA is not in wide use in North America, no its portability is limited as well. One could speculate that a faster , more reliable, and more portable SLIDE simulator would result if the entire package were translated into, say, PASCAL.[1] However, the translation would be far from trivial, since the SLIDE simulator software makes extensive use of the power of SIMULA, from the class concept on up. The answer to the question of whether to do the translation by just building SIMULA primitives out of PASCAL or instead to start over from scratch and tailor everything to SLIDE simulation is not obvious to this author. Realistically though, if such a transition from SIMULA to another programming language is to be made, it should be done so as early

[1] This is prociooly what occurrod in (ho SAOLE projocl

as possible in the evolution of the simulator, and , in the interest of programming tractability, into a language such as PASCAL, ALGOL, or BLISS (as opposed to a language such as FORTRAN).

Aside from the above language considerations, some speculations are in order on the future of the SLIDE simulator. One envisioned addition to the simulator has been a facility for ISPS simulation that would parallel that for SLIDE within the multi-level simulator core. Thus, ISPS-describecl hardware could be included to make possible some ambitious multi-computer type simulations. Physical feasibility questions aside, there is a basic problem here, which is One of building skyscrapers on bungnlow-sizc foundations. The multi-level simulator core and SLIDE have fortunately made a good marriage. However, the core itself is a primitive implementation of ideas that arc five years old. For all their I/O descriptive drawbacks, both SARA and SABLE are much more sophisticated and powerful simulators , and they are indicative of tho stalr-of-the-art. The simulator core by comparison constitutes the bare minimum, no more. In this light, it would be ill-advised to base a major multi-level, multi-language simulation facility for the CMU DA community on the simulator core. Instead, it would be prudent to take the lessons learned in producing the SLIDE simulator and combine those with the lessons learned by others in producing their multi-level simulators to come up with a solid , usable state-of-the-art system. The SLIDE simulator as it currently stands should be developed as an interconnection strategy evaluation tool, not as a general multi-level simulator.

References

[1] Abramovici, M., Drcucr, M. A.f Kumar, K.
Concurrent Fault Simulation and Functional Modeling.
In *Nth Design Automation Conference Proceedings,* pages 128-137. IEEE and ACM, 1977.

[2] Barbacci,M., Barner,,G., Catlcll,R., Sicwiorck,D.
The Symbolic Manipulation of Computer Descriptions ; The ISPS Computer Description Language.
Technical Report, Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., March, 1978.

[3] Barbacci.M., Nnnlc,A.
The Symbolic Manipulation of Computer Descriptions ; ISPS Application Note: An ISPS Simulator.
Technical Report, Dcpt. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa., March, 1978.

[4] BirtwKtlo, G. ot al.
SIMULA BEGIN.
Petrocclli/Charter, 641 Lexington Ave., N.Y.C.,N.Y. 10022, 1973.

[5] Breuer, M. A. , Editor.
Digital System Design. : *Digital System Design Automation : Languages, Simulation, and Data Base.*
Computer Science Press, Inc., 1975.

[6] Caplcnor, H. D. and Janku, J. A.
Top-Down Approach to LSI Design.
Computer Design 13(8): 143-148, Augur.t, 1974.

[7] Chen, Robert C. and Coffman, James E.
Multi-Sim, A Dynamic Multi-Level Simulator.
In *15th Design Automation Conference Proceedings,* pages 386-391. IEEE and ACM, 1978.

[8] DcBencclictis, Erik P.
Multilevel Simulator.
Master's thesis, Dcpt. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa. 15213, 1979.

[9] Estrin, G.
Modeling for Synthesis - The Gap Between Intent and Behaviour.
In *Proceedings of the Symposium on Design Automation and Microprocessors,* pages 54-b9. 1ETE and ACM, 1977.

[10] Gardner, R., Er.trin, G., Potash, H.
A Structural Modelling Language For Architecture of Computer Systems.
In *Proceedings of 1975 International Symposium on Computer Hardware Description Languages,* pages 161-171. IEEE and ACM, 1975.

[11] Gardner, R. I.
A Methodology For Digital System Design Based On Structural and Functional Modeling.
Technical Report UCLA-ENG-7488, Computer Sciemce Department, UCLA, January, 1975.

[12] Hellestrand, G. R.
MODAL : A System for Digital Hardware Description and Simulation.
In *14th Design Automation Conference Proceedings.* IEEE and ACM, 1977.

[13] Hill, D.
ADLIB : A Modular, Strongly Typed Language .
In *Procceedings of 1979 International Symposium on Computer Hardware Description Languages,* pages 75-81. IEEE and ACM, 1979.

[14] Hill,D., and vanCleemput,W.
SABLE: A Tool for Generating Structured, Multi-level Simulations.
In *Proceedings of the 1979 Design Automation Conference.* IEEE and ACM, 1979.

[15] Hill, F. J. and Peterson, G. R.
Digital Systems : Hardware Organization and Design.
Wiley, 1978.
Second Edition.

[16] MacDougall, M. H.
System Level Simulation.
In *Digital System Design Automation : Languages, Simulation, and Data Base,* chapter 1. Computer Science Press, Inc., 1975.

[17] Parke, Frederic I.
An Introduction to the N.mPc Design Environment.
In *16th Design Automation Conference Proceedings,* pages 513-519. IEEE and ACM, 1979.

[18] Rammig, F. J.
Hardware Description Language CAP and its Applications.
In *Proceedings of 1979 International Symposium On Computer Hardware Description Languages,* pages 138-144. IEEE and ACM, 1979.

[19] Razouk, R.
The Graph Model of Behaviour Simulator.
In *Proceedings of the Symposium on Design Automation and Microprocessors,* pages 67-76. ILEE and ACM, 1977.

[20] Szygenda, S. A., and Lekkos, A. A.
Integrated Techniques for Functional and Gate-Level Digital Logic Simulation.
In *10th Design Automation Conference Proceedings,* pages 159-172. IEEE and ACM, 1973.

[21] VanCleemput, W. M.
A Hierarchical Language for the Structural Description of Digital Systems.
In *14th Design Automation Conference Proceedings,* pages 377-385. IEEE and ACM, 1977.

[22] Wallace, J. and Parker, A.
SLIDE: An I/O Hardware Descriptive Language.
In *Proceedings of the. 1979 International Symposium on Hardware Descriptive Languages, Palo Alto, CA.* IEEE and ACM, October, 1979.

[23] Wallace, J.
The GLIDE Compiler.
Research note, Electrical Engineering Department, Carnegie-Mellon **University, April,** 1979.

[24] Wallace, J.
SIGNALS: A Proposed Extension to GLIDE.
Research note, Electrical Engineering Department, Carnegie-Mellon University, Feb., 1979.

I. Simulator Commands

The capabilities of the SLIDE simulator are only partly reflected in the commands available to the user at present. This is because the SLIDE simulator is still under development, especially at the user interface. The following list of currently implemented commands therefore docs not represent the maximum performance level of the program.

ADD <labol>: <dovico> <paramotors> <namo> <namo> , . . <name>;

e- cj. ADD BILL: INVERTER UI U2;
ADD SUE : DELAY U2 U3;

The ADD command creates the data structures for the ports of <device>. The resulting module is labelled to distinguish it from other instances of <device> in the simulator. <parameters> is optionally used by non- SLIDE functional modules for passing of device-related parameters. The remainder of the command field does the wiring for the ports of <device>. A one-to-one correspondence exists between each <name> and a port of the device, by the left to rip.ht position of <name>. For each name, a wire model is created and labelled with <namo>. Then it is connected to the corresponding port. So for BILL in the above example, WI is the name of a wire that is connected to port 1. W2 is the name of a wire that is connected to port 2, and to port 1 of SUE. The conceptual results of these two ADD commands are shown in Figure 1-1.

ALL

This command prints out the accumulated connection information from all the ADD's that have been done so far.

DUMP <filonamo>

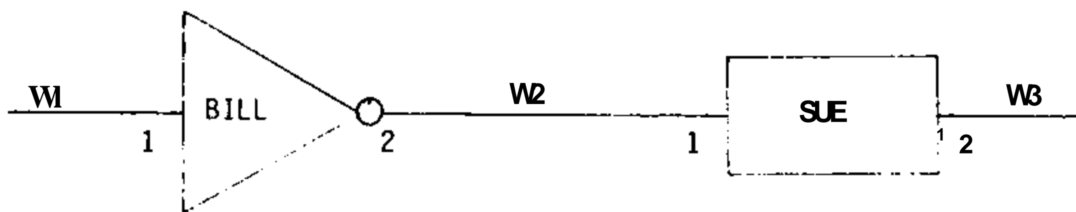
This command dumps out the accumulated connection information to a file called <filename>.

GET <filonamo>

This command retriever* lhe connection information in <filcnamc> and implements it.

So, the user could build up a test interconnection usinff ADD's, then save it using DUMP. At any time, even on a different simulation run, he or she could recreate the interconnections by GETting the appropriate file.

Figure 1-1: Results of example ADD commands



PROBE <wiro>

```
e. g.   PROBE ADC
       -PRUDE XYZ
```

This command turns on a trace for the named wire. Whenever this wire is written to, the state of the wire and the current simulated time will be output to the terminal. -PROBE <wire> turns off the trace on <wire>.

WHAT <label>

e. g. WHAT SUE

This command caur.es the entire state of the device <label> to be output.

SIMULATE

This causes the actual functional model of each device specified in the ADD commands to be created. It is at this point that any simulation time parameters specified in the original SLIDE description arc bound by the user.

GO <number>

Run the simulator for <numbcr> microseconds.

UNTIL <numbor>

Run the simulator until simulated time equals <number>.

FREEZE

This command caur.oo the core imago of the simulation program to be saved. This allows easy restarts for simulation tests that start at a certain point, but then are personalized via various parameter combinations, for example.

II. Test Traces And Commentary

This appendix contains the simulator traces for the test runs described in Chapter 6, and some commentary on the first test run.

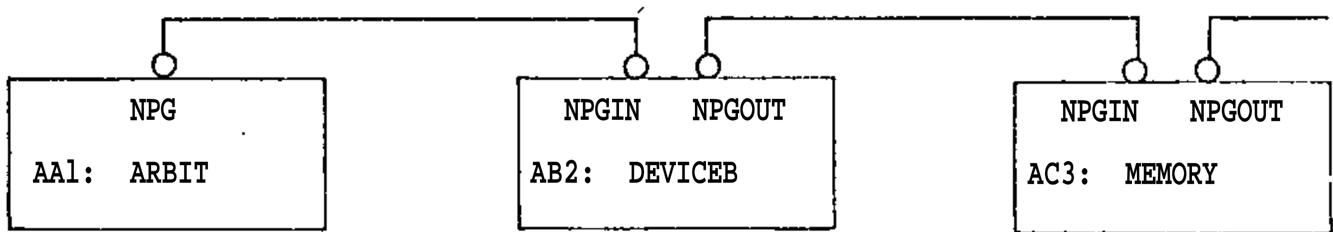
Figure II-2 shows the interconnection commands used to create the example configuration. The interconnections were fetched from a prepared file using the GET command (the file is TEST2). The resulting interconnections are displayed further down with the ALL command. The GET command result is equivalent to ADDing each of the devices and interconnections shown individually.

First, an ARDIT device is created from the SLIDE module of the same name, and it is called "AAI", distinguishing it from other ARDIT devices we may want to use. At this point, the user has already been given a list of the ports available for interconnection and their names, e.g., "BBSY corresponds to port 6". At interconnection time, the user may rename these ports to avoid confusion between multiple copies of devices, and list the names of ports, in order to the interconnector. Ports are then connected by name correspondance. So, ARBIT device AA1 might have a BDSY port (port 6) which we will name BBSY1. ARBIT device AA2 might have a BBSY port which we will name DDSY2. (Thus the instantiated devices and their interconnections may not have names corresponding to labels in the SLIDE modules themselves.) The ports which are named at interconnection time are given in terms of increasing port numbers from left to right.

In our example (Figure II-2), "A" caur.es a wire labelled "A" to be created and port 1 of ARBIT device AA1 to be connected to it. "D" causes port 2 to be connected to wire "D". On line two of this example, we acid a DEVICEB Module named AB2 to the system model. (This is the SLIDE module for the peripheral device.) By typing "A" at the port 1 position of AB2, we connect it to wire A.

The most interesting connection of wires is the daisy-chain of bus grant wires on the UNIBUS. DEVICEB and MEM0RY(AC3) each have a grant-in and grant-out line, at ports 9 and 10 and 10 and 11 respectively. The bus arbiter has the grant line emanating from port 19. By connections shown in the example (Figure II-2), we achieve the setup shown in Figure II-I.

Figure II—1 r Daisychaining of UNIBUS NPG Line



The GET command (equivalent to the ADD command) causes devices to be connected at the *element* level. The SIMU command then causes the *chain* and *simulation process* for each device to become present. The connections that have been made are checked for compatibility at this point and any parameters given in the original SLIDE module are given values at this point. In our example in Figure II-2, we see that .PER was the period of the sync line, .ADDR is the SDLC address, and JOPME is the address to put the first word in memory.

The trace facility command, PROBE (PR) caused each wire being probed to output its state whenever it is written to, whether the value on the wire had changed or not.

Once these preliminary commando wore executed, we ran the simulation for .239 usec with the UNTIL 0.239 command.

Figure II-3 shows the tracing on the sync line, called "INTO". It had a period of 30 nanoseconds for this example, so every period the wire got written to, and a trace output resulted. This trace output printed tho current simulation time, along with the wire name. On the next line, the logic type, bit width and current wire value were displayed. This figure shows data on the wire coming after the flag and address. Note the zero insertion after 5 one's, which adheres to the SDLC protocol.

Bus connections were displayed by showing all device ports that are connected to each wire, along with the values each of them arc putting on the bus.

Figures II-4 to II-8 shows the peripheral device actions over the UNIBUS. The device first gets control of the bus from the bus arbitrator, then acts as a bus master to write data to the bus memory (bus slave).

```
.EXEC SDTC
[19:77!56J
LINK: TOadinq
tLNKXCT sDLC EXerution1
SLTDF/^uj ti-T eve| Simulator version 1*0
Welcome and Good Luck!!

•GFT TFST2
•SIMU ^
%Slmijlat1on time parameters for AA1 : ApBjT rn^y be bound now
%flnlsbed
%Simulation time parameters for *B? : DEV1CKR irdy be bound no*
#AD1j=1on
.PE :=30
%finishe
%Slmulation time parameters for AC3 : MEMORY may be bound now
.TOPME:=10
%f
%Sln»ulation time parameters for AD4 : RLPUX may be bound now
•PER:s30
•Apl:=too
%finished
JALL
AA1:     ARBIT   A n MSYN SSYN TNTR C PA PB SACK BBSY INIT ACLO DCLO

         f'PP BR4 BR5 BR6 BH7 NL'G BG4 BG5 BG6 BG7 PSW READY;
AB2:     DEVICER  A 0 MSYNH SSYNP <* SACK BR5Y NPR Np# NPG2 1N1'T;
AC3:     MEMORY   A n MSYN SSYM TNTR C SAC^ BBSY *PR NPG2 NPG3j
AD4;     RLPUX   TNTO;
AE5;     DELAY/DELAY(75)/LOGIC(4)         MSYN MSYNB;
AF6:     DELAY/DELAY(75)/LOGIC(4)         SSYN SSYNB;
•PR A
•PR n
•PR MSYN
•PR MSYhP
•PR SSYN
»PP SSYHP
•PR C
•PR SACK
•PR BBSY
•PR NPP
•PR NPC
•PR NPG2
•PR NPG3
•PR INTO
•UNTIL 0#.239
-->         0.030us TfJTO
 LOGIC=    6 SIZE=    o PERlnn=        30.000US
VALUES UN WIPK-
0
VALUES At.ONG »!IS-
->          0.060US TNTO

 LOGIC=    6 SIZE=    0 PERIOD=        30.000US
VALUES ON WIPE-
1
VALUES AT.ONG |UJS-
```

Figure II-2: First Test Run : Trace 1

Figure II-3: First Test Run : Trace 2

```
••>          0.510us  INTO
 LOGIC=      6 s1Zts    0 pERIOD=        30.000US
VALUES  ON  WIPE"
X
VALUES  AT,UWG  »US-
-->          0.S40•js  TNTO
 LOGIC=      6 SIZE=    0 PERIOD=        30.000US
VALUES  ON  W1RE-
1
VALUES  AT,1)NG  HUS-
•->          0.S70US  TNTO
 LOGIC=      6 fiFZEs   0 PERIOD=        30.0U0US
VALUES  ON  W1RE-
1
VALUES  AT.UNG; BUS-
-->          0.600us  TNTU
 LOGIC=      f>.s1ZtJs  0 PERIOD=        3f).000Us
VALUES  UN  WJPE-
1
VALUES  AT LJNG1 iUJS-
-->          0.630u5  TNTO
 LOGTC=      6 siZKs    0 PERIOD=        30.000US
VALUES  OV  WIRh-
1
VALUES  AT I.'NI; MUS-
-->          0,660us  TfJTO
 LOGIC=      6 5?I7K =  0 PERIOD=        30.000US
VALUES  ON  WIRF>
0
VALUES  AT UNG .nus-
-->          0.690Us  Tf4TU
 LOGTC=      6 S1ZKs    0 PERIOD=        30.000US
VALUES  ON  WIPE-
1
VALUES  AT.LING BUS.
-•>          0.720US  TNTU
 LOGICr      6 51ZEs    0 PERIOD=        30.000US
VALUES  ON  W1PF>
1
VALUES  ATIUNG BUS-
-->          0,7!>Ous  INTU
 LOGIC=      6 .SIZL=   0 PERIOD=        30.000US
VALUES iON  W1RE-
1
VALUES  AM JNG BUS.
```

```
«»>           1•020u5; TNTO
 LOGTC=       6  S1ZEs    0  PERIODS        3o .000Us
VALUES  ON  WIRE-
1
VALUFS  AT.ONG  nl?S«
-->           1•02fiU«; NpR
 LOGIC=       4  S1ZEs    0  PERlODs        0 .000Us
VALUFS  ON  W1PE-
1
VALUFS  AT.ONG  HHS-
AC3    9
0
AB2    R
1
AAt    14
0
-->           1•O'^Sus  NpG
 LOGIC*       6  S1ZEs    0  PERIOD^        n.noous
VALUFS  OM  WIPE-
1
VALUES  AT.ONG  1UIS-r
-->           1"#O29us  SACK
 LOGIC=       4  sxzfc:=   0  PERIPD=       0•000us
VALUES  OH  WIRf;•
1
VALUFS  AT.OWG  HUS-
AC3    7
0
AB2    6
1
AA1    9
0
-->           t .02^Us  npp
 LOGTC=       4  c!ZE=     H  pEPlOD=       n•000Us
VALUFS  Of  WRU-
0
VALUFS AT 1LING  BUS-
AC3    9
0
AB2    ft
0
AA1    14
0
-->           1.030us  wpc
 LOGTC=       6  S1ZEs    0  PEPlODs        0 •000us
VALUES  ON  W1RE-
0
VALUFS  AT,(3NG  8"S-
»•>           1#03tus  BHSY
 LOGIC=       4  S1ZEs    0  PEPIOD=        0 .000Us
VALUFS  ON  WIRE-
1
VALUFS  AT.dNG  BUS-
AC 3   f1
0
AB2    7
1
AA1    10
0
```

Figure II-4: First Test Run : Trace 3a

Figure II-5: First Test Run : Trace 3b

```
-->          1«032us  A
 LOGTC=    4  SlZfc;=   17  pERIOD=          0.OOOus
VALUES  ON  WIPE-
oooooooooooooooooi
VALUES  AT.ONG  QUsl
AC3    1
ooooooooooooooooooo
AB2    1
oooooooooooooooooi
AA1    1
ooooooooooooooooooo
-•>          1.032US  n      pfRinn=          0.000us
 LOGTC=    4  Sl?.fc:=   IS
VALUES ON WIPF-
11ti1111711o11oi
AC2    2
VALUES  AT.ONG  iWS-
OOOoooooooooonooo
AB2    2
1111111 111101 101
AA1    2
ooooooooonooooooo
->           1 .032us C
 LOGTC=    4  SlZEs    1  pER1PDr          0.000us
VALUFS  ON  WIRg-
10
VALUFS  ATONG  BUS-
AC3    6
00
AB2    5
10
AA1    6
00
-->          1.033us  SACK
 LOGTC=    4  ^IZE=    0  FERlf-I) =         0 .000us
VALUFS  ON  WlRE-
0
VALUFS  ATUNG  BUS-
AC3    7
0
AB2    6I
0
AA1    9I
0
-->          1.1fi5us  MsYNB
 LOGIC*    4  SlZEs    0  PERlPHs          0.000Us
VALUFS  nN  WlpE-
1
VALUFS  AT.ONG  BOS-
AE5    7
0
AB2    3
1
```

```
••>          1•?bOUs  »ISYN
 LnGTC=    4  SIZEs    0  PERIOD*          0.000US
VALUES  ON  WIPE-
1
VALUFS  ATONG  BUS-
AES    1
1
AC!    3
0
AA1    3
0
```

Figure II-6: First Test Run : Trace 3c

```
-->          1.760us SSYN
 T.OGIC=     4 SlZEs   0 PERIOD=          0.000Us
VALUFS ON WiPF-
i
VALUFS ATONG B"S-
AF6    1
0
AC!    4
1
AA1    4
0
«>           1.335us SSYNIJ
 LOGTC=      4 SIZE=   0 PERIOD=          0.000US
VALUFS OW WIRE-
1
VALUES AT.UNG «!IS-
AF6    ?
1
AB2    4
0
-->          1.336us MSYNP
 LnGTC=      4 SIZE=   0 PERIOD=          0.000lls
VALUFS nw WIRE-
0
VALUFS AT#|iMG BUS-
AE5    2
0
AB2    3
0
-->          1.136us D
 LOGIC=      4 SlZEs  is PERIOD=          0•000VIs
VALUES aw WIRE-
00000000p0000000
VALUFS ALONG BUS-
ACS    7
0000000000000000
AB2    2
0000000000000000
AA1    2
0000000000000000
```

```
—>          1.4 llllS  MSYN
 LOGICs    4 siZE=    0  PEl'IODr          0.000us
VALUES  ON  WlpE-
0
VALUES  AT.ONG  RUS-
AE5    1
0
AC3    3
0
AA1    3
9
—>          1.4 12'Js  SSYN
 LOGTC^s   4 SlZEs    0  pERIflfJs         n.ooous
VALUES  ON  WIRE-
O
VALUES  ATONG  BIIS-
AF6    1
0
AC3 .  4
0
AAt    4
0
—>          1.43&us  A
 LOGIC=    4 SlZEs   17  PERIODS          0.000us
VALUES  ON  WIPR-
oooooooonooonooooo
VALUES  AT.UNG  BUSl
AC 3   1
oooooooonooonooooo
AB2    1
oooooooooooooooooo
AAl    1  -
QQ9OOOOO 000UQ9QQQO
-->         1.439us  r
 LOGIC^2   4 SlZEs    t  pERlODs          n.ooous
VALUES  ON  WIPE-
CO
VALUES  AT.UNG  H«.»S-
^c^    6
00
AB2    5
00
AAt    *
00 .
-->         1.439us  BBSY
 LOGIC=    4 SIZE=    0  pERIP|)=         o.oonus
VALUES  ON  HIRE-
O
VALUES  ALONG  BUS-
AC3    8

0
AB2    7
0
AAt    10
0
```

Figure II-7: First Test Run : Trace 3d

```
—>          1 .4B7us  S.SYNP
 LOGICs    4  SIZEs    o  PERIOD=         o.ocous
VALUES  ON  WIRE-
0
VALUES  AT.UNG  BUS-
AF6    2
0
AB2    4
0
```

```
-->        . 1.504Us
•WHAT Aci
*State Of AC}
NPG3
 LOGICs    6 SlZfcs    0 PERI(iI)s       n.ooous
VALUES ON WIPF-
0
VALUFS AT.ONG n"s-
NPG2
 LOGICs    6 SIZE=    0 PER1OHs       n.ooous
VALUES ON WlpE-
0
VALUES AT,UNG »US-
NPP
 LOGICs    4 SlZEs    0 PER1PDr       0.000Us
VALUES ON W1RE-
0
VALUFS AT CINU BUS-
AC3   9
0
AB2   8
0
AA1   H
0
STATF OF.M :
ooooooooooonooonooc
oooooooooooooonooo
oooooooooooooooooo
oooooooonoooooooo
oooooooonuouoooo
oooooooonoooooooo
oooooooonooonooo
oooooooonoounooo
oooooooooooooooooo
U i l l l l l i l i o i i r>i
oooooooonooonooo
-->        6.979us
#WHAT AC?
*State Of AC3
NPG3
 LOGTC=    6 SIZE=    0 PER1OD=       o.ooous
VALUFS ON WIPE-
0
VALUFS ATONG R!IS-
STATE Ob H :
oooooooooooooooooo
oooooooooooooooooo
0011101100100001
1111111111101101
0011101 1 11101101
1111111 1 oojonont
0011101 1 11101101
1111111 1 nuionooi
001Hoi 1 11101101
oooooooooooonooo
*EX1T
```

Figure 11-8: First Test Run : Trace 3e

```
40 Garbace collections executed durinq 8322 ms

End of SJHUL^ oroarm* execution.
CPU tjirp: u - }? tri.p.n^ ti~.i
```

```
• EXEC SDT.C
tl8:io:ooj
LINK: T.oaciing
[LNKXCT 5DLC £Xeciitjonl
SLIDE/Multi -level Simulator version i.o
Welcome and food Lucfc!j

•GET TFST2
#SIMU
%Simulation time parameters for AA1 • flpP^T mav be bound no*
*finished
%Simulation time parameters for AU2 • DEVICER way be bound now
•AD1:=100
•PER:=20
%finished
%Simulation time paran»eters for AC^ : MEMORY may be bound no*
.TOPME:=10
tflnlsherj
%SImulation ti^e parameters for Au4 ': BLBOX may be bound now
•PER:=20
•AD1:=100
%finished
IPP A
#PR D
•PP MSYN
•PP WSYNP
•PR SSYN
• PR SSY?>ip
• PR C
•PP SACK
•PP PBSY
•PR Npp
•PP NPC
•PR NPC2
•PR NPG3
•PP INTO
• UNTIL O#.239
-->          0«020Us  TNTO
 LnGIC=      6  STZLs    0  PERlnnr         20.000us
VALUFS DN WIRE-
0
VALUFS AT.ONG BUS-
-->          0.040us  INTU
 LOGIC=      6  SIZE=    0  PERinn=         20.000us
VALUES OV WIRP-
1
VALUES ATUNG RUS-
mm>          0.060us  TNTU
 LOGTC=      6  51Zfc;=  0  PfcJPlfiDir     20.000Us
VALUFS ON WIRE-
```

Figure II-9: Second Test Run : Trace 1

```
STATE OF M :
ooooooooooooonoou
liuniif nut ioi
ooooooooooooooooo
11111111111101101
ooooooooooooooooo
11f11111I1011 101
ooooooooooooooooo
liuiiii i uoi ioi
ooooooooooouoooo
11111111111101101
ooooooooooooooooo
```

--> 0.000us MSYN
LOGIC= 1 SIZE= O PERIOD= 0.00 0 us
VALUES ON WIRE-
1
VALUES AT.M7'i. BUS-
AE5 1
0
AC 3 3
0
AA1 3
1
--> n.o0 01js n
LOGIC= 4 SIZE= 15 PERIOD= 0.000u<;

oooooooooCooooono
VALUES ALUMC BUS-
AC 3 7
nU0uo(inuouoùo-jno
AH? 7
000 00 0.00000 Ii 000 0
AMI ?
0000000000000000
-->
LOGIC= -1 .^]r'H:= 0 PERIOD= 0.000us
VALUES ON WIRE-
1
VALUES ALUMC BUS-
AP6 1
0
AC3 4
1
AA1 4
0
--> 0.030us T rr0
LOGIC= 6 SIZE= 0 PERIOD= JO.000US
VALI'Fti ON WIRE-
0
VALUES ALUMC BUS-
--> 0.060us PSTO
Lib C;T(!= h .SIZE= 0 PERIOD= JO.0C0us
yALUFS UM IJ1PE-
1
VAI WS AT.i:'N; BUS-
--> 0.075us MSYNP
LOGIC= 1 SIZE= 0 PERIOD= n.nu0us
VALUES ON WIRE-
1
VAL..UFS />T,ONC BUS-

1
AB2 3
0
0.075us SSYNP
LOGIC= 4 SIZE= 0 PERIOD= 0.000us
VALUES ON WIRE-
1
VAT.UF5 iiT ÀJMC t311 S-
AK6 ?
1
AB2 4
0

Figure 11-10: Third Test Run : Trace 1

Figure II-11: Third Test Run : Trace 2

```
-->            1 .03'ius  SACK
  bfH;TC=     -1  *J7L=     o  pK'-If-t;r              O.OOOus

0
VAT.UFS  AT.IjNc:  bIIò-
AC 3    7
0
AB2     6
0
AA1     0
0
-->            1.0500US  TI•TU
  LOGIC=     6> ?;17F=     0  PERIOD=            JO.oonus
VALUES ON WIRE-
0
VALUES ALONG BUS-
-->            1.0800US  TITCI
  LOGIC=     6> S17L*F     0  PERIOD=            io.ooo«s
VALUES ON WIRE-
0
VALUES ALONG BUS-

                                   OOOOOOOUOOOOOOOOO
                                   OOOOOOOOOOOOOOOOU
                                   0U0U0U0U0U0U0U0U0
                                   000000000()000000
                                   00000000000000000
                                   0V000 000000()000 0
                                   000000°0000()0000
                                   00000()0OnonoOOO0
                                   00000000n(j0o000u
                                   00000000000000000
                                   OUOUOOOUOOOONOOO
                                   #EXIT
```