Noodles: A Design Tool for High Level Design and Verification using SDL

Thesis submitted in partial fulfillment of the requirements for the degree of

Bachelor of Technology (Honours)

in

Computer Science and Engineering

by

Atri Rudra

Under the Guidance of

Dr. P.P.Chakraborti

Department of Computer Science and Engineering
Indian Institute of Technology
Kharagpur

May 2000

CERTIFICATE

This is to certify that the thesis titled **Noodles : A Design Tool for High Level Design and Verification using SDL** submitted by Atri Rudra, to the Department of Computer Science and Engineering, in partial fulfillment for the award of the degree of *Bachelor of Technology* is a bonafide record of work carried out by him under my supervision and guidance. The thesis has fulfilled all the requirements as per the regulations of this institute and, in my opinion, has reached the standard needed for submission.

Dr. P.P.Chakrabarti
Dept. of Computer Science and Engineering,
Indian Institute of Technology,
Kharagpur - 721 302, INDIA.
Dated : May, 2000.

ACKNOWLEDGEMENT

I am thankful to my supervisor **Dr. P.P.Chakraborti** for his guidance and encouragement throughout the course of this work.

This work would not have been possible without the invaluable help of **I.Bhattacharya, A.Dasgupta** and **A.C.Pathak**.

The name of the design tool is due to **Nitin Garg**.

ATRI RUDRA
Dept. of Computer Science and Engineering.
Indian Institute of Technology
Kharagpur - 721 302 , India.

Contents

List of Figures

Chapter 1

Introduction

1.1 Introduction to SDL

Specification and Description Language(SDL) is a standardized programming language for high level specification and description of systems, embedded or otherwise, that comprise concurrent, communicating, reactive processes. It is largely used for circuit level specification and for simulation of network protocols.

Design style is an important aspect of using any design tool. While SDL has capabilities of code execution, branching, looping and interprocess communication, it encourages a design style which is natural to such communicating processes. It includes the features of simple block hierarchy, processes (state machines), states and transitions, sending and receiving of signals, branching and the use of synchronizing timers to enable a natural design style for hardware systems.

SDL gives us a model of processes as finite state machines equipped with signal queues. Signals are a mechanism for interprocess communication. Signals may be thrown and received by processes. A signal thrown by a process is broadcast to all the process in its domain. Each process has a set of states from which it can make a transition on consuming a signal from the queue. The set of signals by which the process can make a transition (i.e., can handle) when in a particular state is defined in the **signal-in boxes** drawn after the state. If the queue is empty, or if the desired signal is not present in the queue, the process blocks in that state. While making a transition, a process may execute C-code (written in the **task-blocks**) or throw signals(**signal-throw constructs**) to other processes. Signals received by a process accumulate in its queue and are consumed in order while making a transition. A signal at the head of the queue is either handled while making a transition, or it is rejected unless explicitly saved by using the **save-construct**. Signals may also have payloads — a feature that helps in generic design. SDL is equipped with C-style variables that may be shared between multiple process and may thus be used for communicating values.

1.1.1 The Arbiter Example

Example 1.1.1 We now illustrate with an example how SDL may be elegantly used for system specification. The example concerns the design of an *arbiter system*. The problem is stated as follows:

There is a central resource held in non-sharable mode, which is required by various devices from time to time. There is an arbiter to handle requests from devices and grant the resource as and when possible. The resource cannot be granted to more than one device at a time.

The protocol that we use for the design is as follows. Every device, sends a *Req* signal to the arbiter whenever it requires the use of the resource. If the resource is free, the arbiter immediately sends a *Grant* signal and waits for the device to complete its job. The device goes through a blocking wait for signal *Grant*. When it receives that signal, it executes whatever task it had to perform with the resource and then indicates that it has completed its job by issuing an *Over* signal. The arbiter keeps waiting for an *Over* signal from the device that was granted the resource since it cannot handle further requests from other devices while the resource is in use. However, it keeps saving the requests from other devices in a queue. Through the use of payloads as identifiers for the devices, the user is relieved of the burden of enumerating signals over the sender domain, and the system lends itself to generic design. The general technique used [Fig. 1.1] is that every device has its own *id* or signature, which it communicates with every signal that it sends. Thus even if multiple senders throw the same signal, the receiver will be able to identify a particular sender for every signal that it catches. Also, a sender may use this concept of an identifier to indicate a particular receiver for a thrown signal that has multiple receivers defined in its domain.

The example clearly illustrates the elegance and simplicity provided by SDL for system design, and parallely highlights the need for a verification platform for the language that would check various specifications on the system, like whether the arbiter eventually acknowledges all requests that it receives. □

1.2 Motivation and Objectives

The motivation behind the development of **Noodles** can be divided into three broad categories-

- Design tool: SDL is widely used for high level synthesis - however all the design tools for SDL which are currently available are propriety in nature. Also no tool commercially available is written in **Java**. Thus the idea was to develop an enviornment for

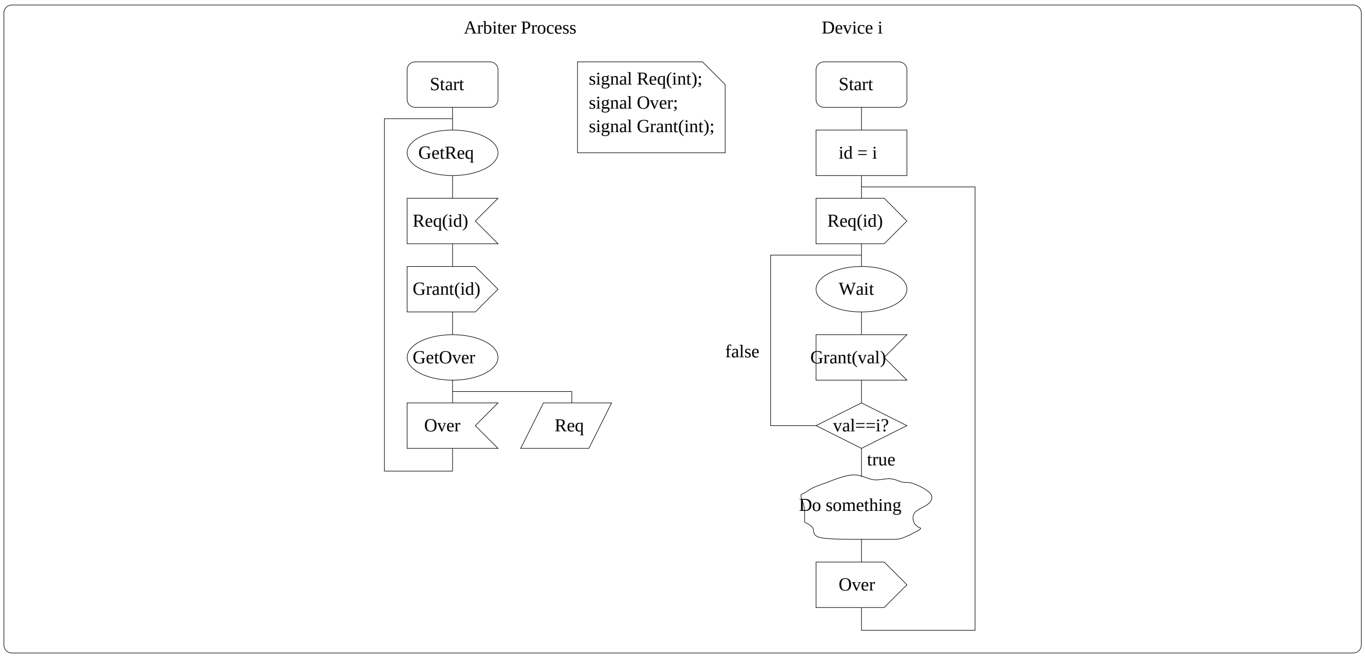

Figure 1.1: Generic Arbiter With Payload

design in SDL using **Java** . The main advantages in developing the tool in **Java** is the portability of the tool along with the fact that it can be put up on the internet in the form of an applet.

- Augmented Constructs: Though SDL semantics provide a rich and varied set of constructs for design purposes, a need was felt for some constructs which make the design process much simpler. Though these constructs would not add any power to the SDL semantics, they would eliminate the need for elaborate design procedures §3.2.

- Automated Static and Dynamic Analysis: Though simulation is a means of verification of the system designed, a complete design tool must have some static and dynamic analysis so that the user may verify some aspects of his design. Dynamic validation (see Chapter.7) is done along with the simulation while Static Analysis (see Chapter.6) is done independently of the simulation .

1.3 Work Done

This section gives a brief overview of extent of realisation of the objectives of the project.**Noodles** has been fully developed in **Java**.

- Design Tool: The design enviornment is complete. The major functional parts of the design enviornment are:
  - *GUI :* As SDL is a visual description language, it is necessary to provide the user with a GUI for input purposes. The GUI has been implemented using the *awt* package of Java.
  - *Simulator and Debugger:* The simulator and the debugger have been implmented. The code generated for simulation and debugging (see Chapter5 for more details) is in Java.
- Augmented Constructs: The following two constructs have been added in **Noodles**:
  - *Peek:* The SDL semantics specifies that the process queues can be accessed only while inserting or deleting signals from it. It was felt that if the user was allowed to *peek* into the queue without changing the queue configuration, it would ease his design. See §3.2.1 for more details.
  - *Library:* Any high level design will be easier if pre-defined libraries were present. Also the SDL semantics does not have any specific construct for for a function call- this feature is advatageous if the design has many processes (or blocks) which differ only by a parameter. In **Noodles**, libraries which are similar to macros have been included. See §3.2.2 for more deatils.
- Automated Static and Dynamic Analysis: One of the fundamental problem in SDL verification is that of state reachability. However, this problem has been shown to be undecidable [4]. Static Analysis tries to solve this problem by restricting the domain of the problem. This static analysis which we call *Weak Reachability* is included in **Noodles**.

  Another direction in which the problem was tackled was to force the simulation in a direction which would lead to some states being unreachable(if there are any).This method of *adverserial scheduling* has been integrated into **Noodles**

1.4 Layout of the thesis

The rest of the thesis is organised as follows. Chapter2 gives an analysis of SDL. Chapter3 gives a brief overview of **Noodles**. Chapter4 goes into the details of the GUI of **Noodles**.

Chapter5 goes into the details of the simulator and debugger in **Noodles**. Chapter6 looks into the static analysis part. Chapter 7 describes the dynamic validation techniques used in **Noodles**.Finally conclusions are drawn in Chapter8. AppendixA decsribes the format in which the system description is stored in files.

Chapter 2

An Analysis of SDL-C

2.1 Introduction

Specification and Description Language(SDL) is a standardized programming language for high level specification and description of systems, embedded or otherwise, that comprise concurrent, communicating, reactive processes. It is largely used for circuit level specification and for simulation of network protocols.

This chapter presents a brief study of SDL-C and its capability to design hardware systems and the power of its features.

2.2 Features of SDL-C

Since systems that are subject to specifications are usually quite complex, any specification language should have some structuring concepts. In basic SDL a system description is structured into block descriptions and process descriptions. Figure 2.1 shows an exmaple.

The *block* can contain *block(s)* or *process(es)*. However the bulk of the system description is in the process description.

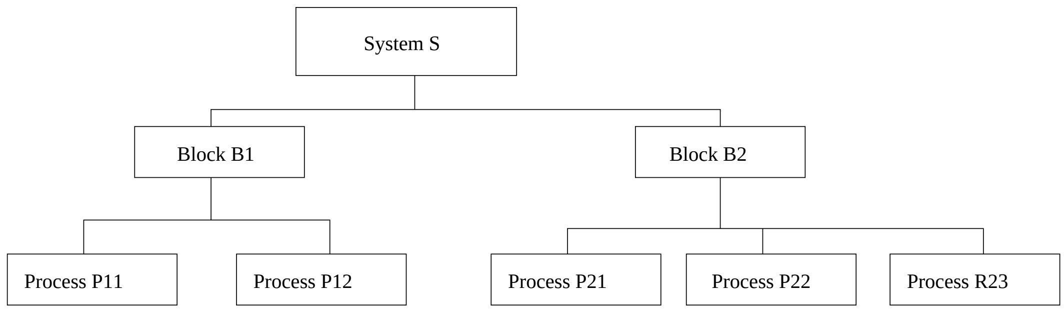

Figure 2.1: Structuring of a system into block descriptions and process descriptions

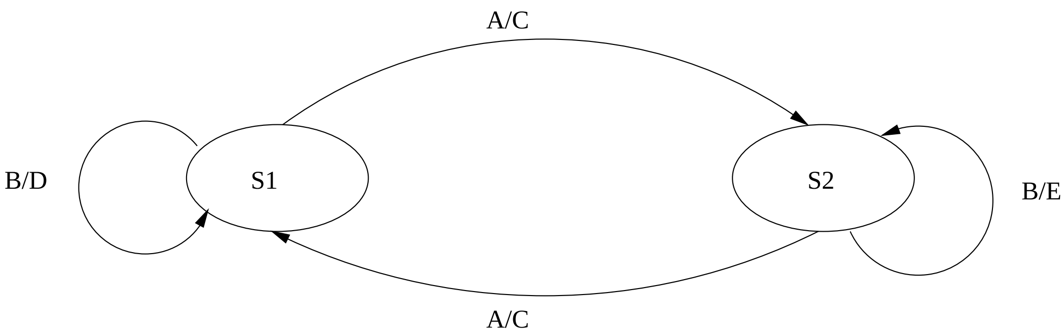

Figure 2.2: Behaviour of a finite state machine

2.2.1 Process Description

States and transitions

A process in SDL is an extended finite state machine, i.e. a finite state machine that can use (manipulate) data stored in variables local to the machine. The behaviour of a finite state machine is described by states and transitions. Figure2.2 shows the behaviour of a finite state machine defined by a directed graph.

Two states S1 and S2 are involved. If the machine is in State S1, it can perform a transition to state S2, initiated by an input A (i.e. by the consumption of signal A). During the transition it will produce an output C (i.e. sending of signal C). The input B in state S1 will result in a transition to the same state, and so on.

In SDL there are three constructs for the description of a process *input* or Signal In, *output* or Signal Out and *state*. A special state state called the *start* state is necessary because a behaviour must begin somewhere.

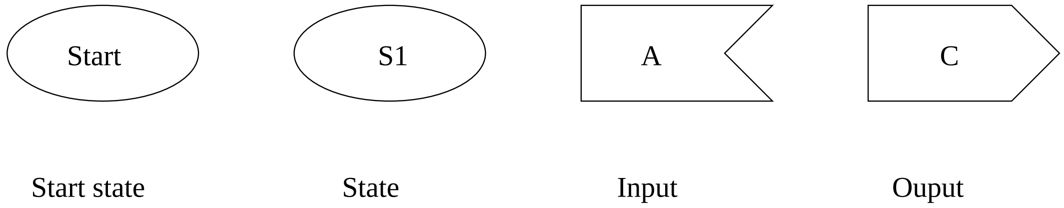

Figure 2.3: Basic Constructs for the decription of a process

Figure 2.4 shows the description of the finite state machine of figure 2.2 by a process diagram. The figure shows syntactically correct SDL.

Description and use of data

As mentioned earlier, a process may locally use and manipulate data stored in *variables*. These are described by a decleration construct as shown in figure 2.5.

During a transition, the process can use and manipulate its own local variables, using the *task* construct. A task construct can contain a set of valid C (Java in **Noodles**) statements. In figure 2.6 the variable *counter* is incremented by one.

Time in SDL

In a system description certain time constraints must usually be defined. SDL has the *timer* construct for this. The timer is an object, owned by a process, that is able to generate a timer signal and put the signals into the input queues of the process.

A timer must be described in the process description. Figure 2.7 shows an example.

During a transition a timer can be activated with the *set* construct. The set construct has two arguments. First one is the absolute time for the expiration of the timer, and the other one is the name of the timer. For the specification of the expiration time, the expression *now* can be used. *now* always gives the current time during the interpretation of the system decription.

In the example of figure 2.7, the timer T is set to the value *now+13*, i.e. the timer should expire after 13 time units. After time has advanced and reached the previously specified value, a timer signal with name T is put into the input queue of the process.

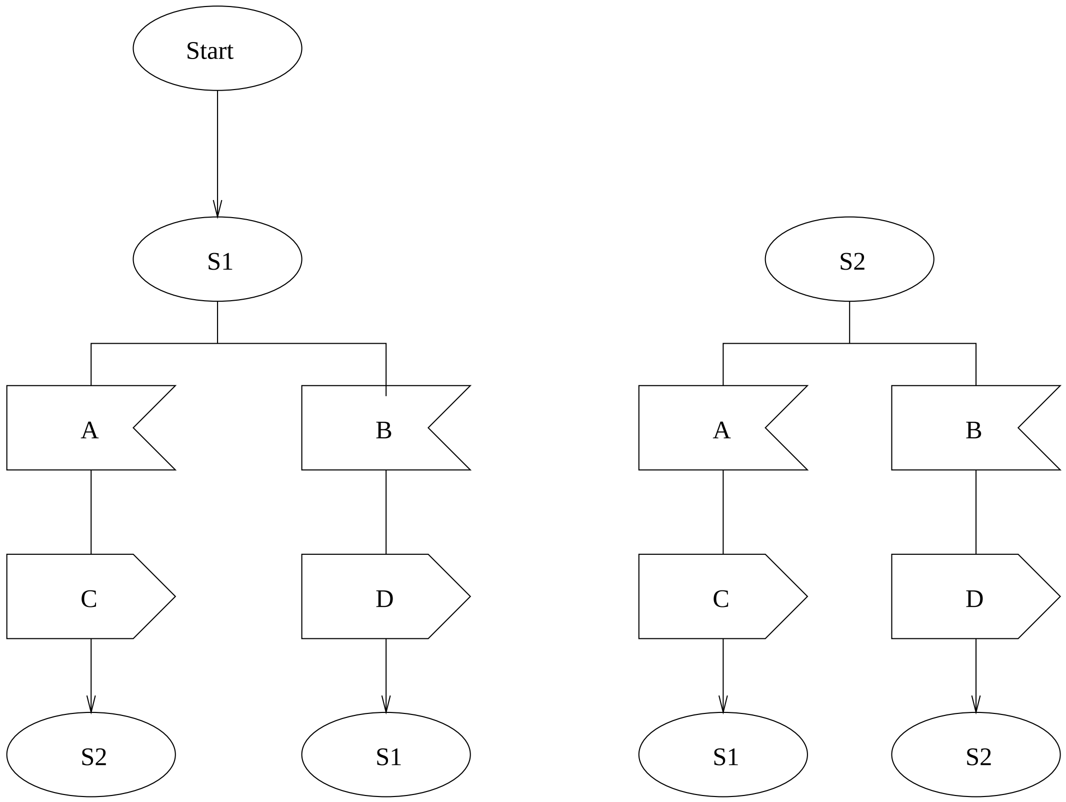

Figure 2.4: Description of the finite state machine of Fig.2.2 by a process description

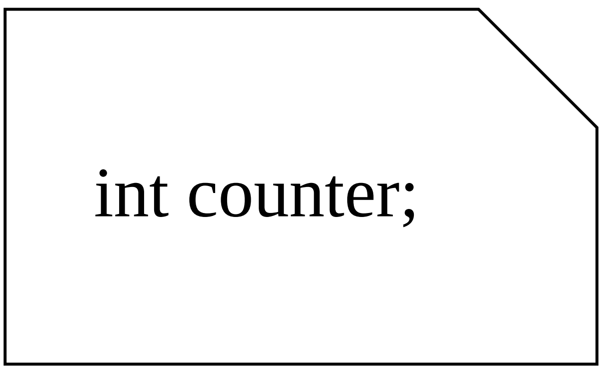

Figure 2.5: Description of a variable using the *decleration* construct

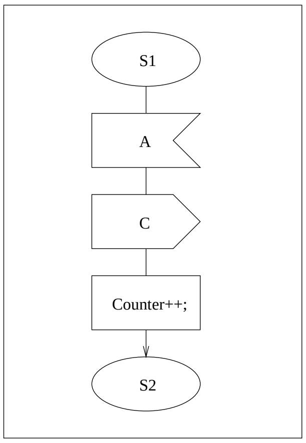

Figure 2.6: Manipulation of the variable

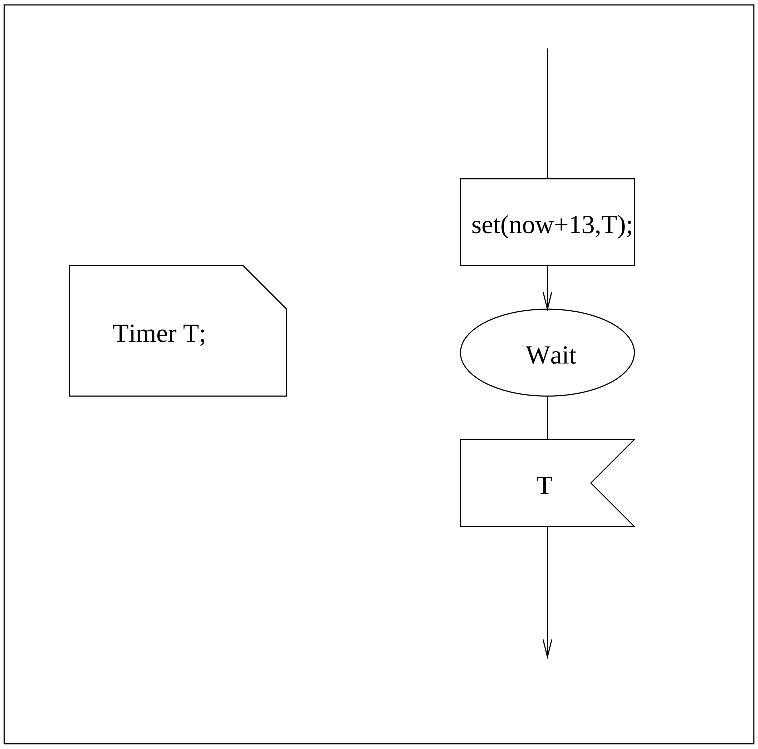

Figure 2.7: Description and use of timer

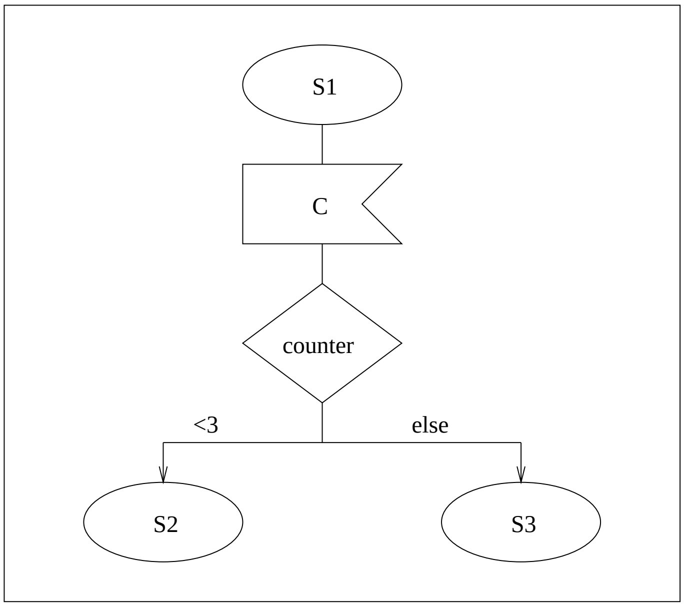

Figure 2.8: Use of the decision

An activated timer can be deactivated with the reset construct. After resetting the timer, the process will behave in a way as if the timer has never been activated.

Decisions

The values of variables can be used to control a transition. Sometimes it is necessary to specify different actions depending on the value of some variable. This is done in SDL by using the *decision* construct. Figure 2.8 shows an example in SDL of the decision construct. Depending on the value of the variable *counter*, the process will perform a transition from S1 to S2 (value is less than 3), or to S3 (otherwise).

Signals and Data

The concepts of *input*(signal consumption) and *output* (signal sending) were introduced in connection with the transition. Previously signals were shown to be used only for the initiation of a transition but they can also be used to carry data values (called *payloads*) from one process to another. In this case the *signal out* construct includes one or more data values. In the corresponding *signal in* construct, variables are specified which store the received values. The types of the values sent in the *signal out* construct must be compatible, of course, with the types of the variables in the corresponding *signal in* construct. The ordering of the values sent together with a signal depends on the ordering of the sorts in the signal definition.

With the signal A in figure 2.9 the values 5 and *true* are sent from *process P1* to *process P2*. The *Process P2* owns variables v1 and v2 of type integer and boolean, respectively.

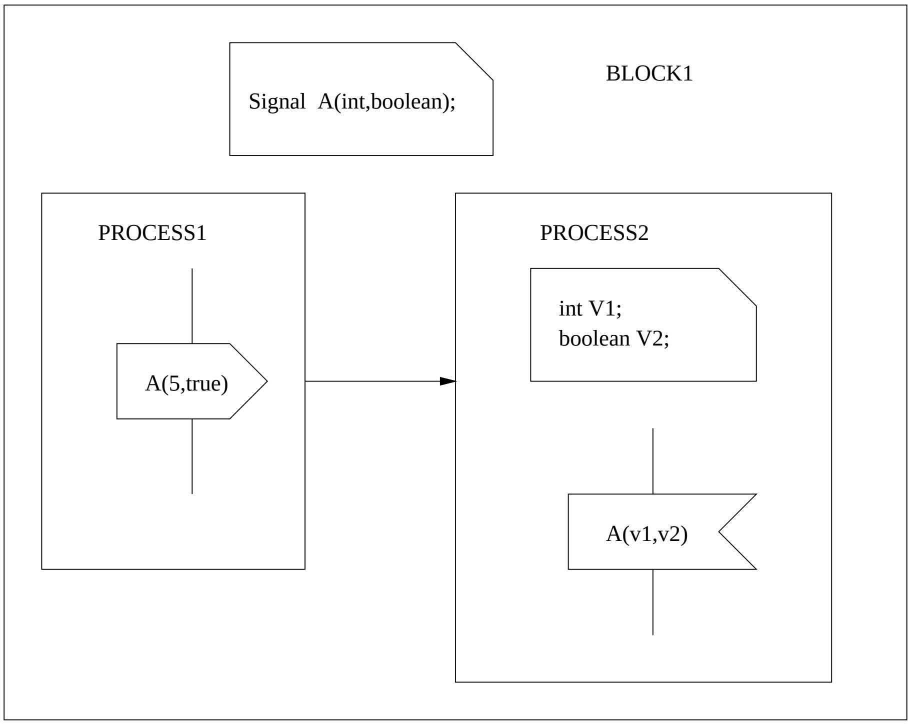

Figure 2.9: Signal with data

When signal A is consumed by *process P2*, the value 5 is assigned to v1 and *true* is assigned to v2.

2.2.2 Semantics of Communication

In the previous discussion only a vague, intuitive explanation was given of how interprocess communication is established. In this section the mechanism is explained in more detail.

With every process an input queue is associated, which acts like a FIFO queue. Any signal arriving at the process and belonging to its so called *complete valid input signal set* is put into the queue, see Fig.2.10. The complete valid input signal set defines those signals that the process is prepared to accept.

If in a given state the input queue is not empty, the first signal (in FIFO order) is removed from the queue (it is consumed). It is checked whether this signal can initiate a transition. If yes, the transition is performed; if no, the signal is discarded. Figure 2.11 shows an example.

Let us consider the process of Figure 2.11 in state S1. The input queue contains signals A, D, A, abd B, in the given order. the signals A and D can initiate a transition. The signal A is first in the input queue. It is removed from the queue and the process preforms

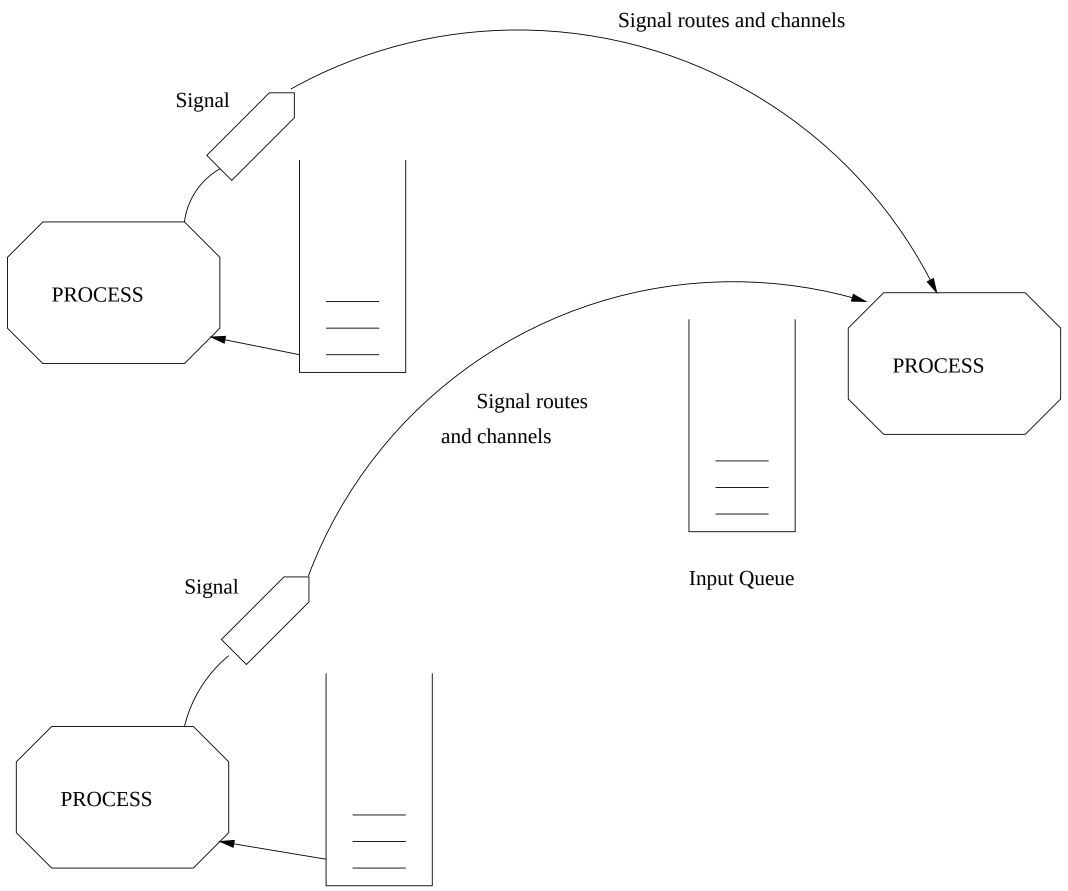

Figure 2.10: Process Communication

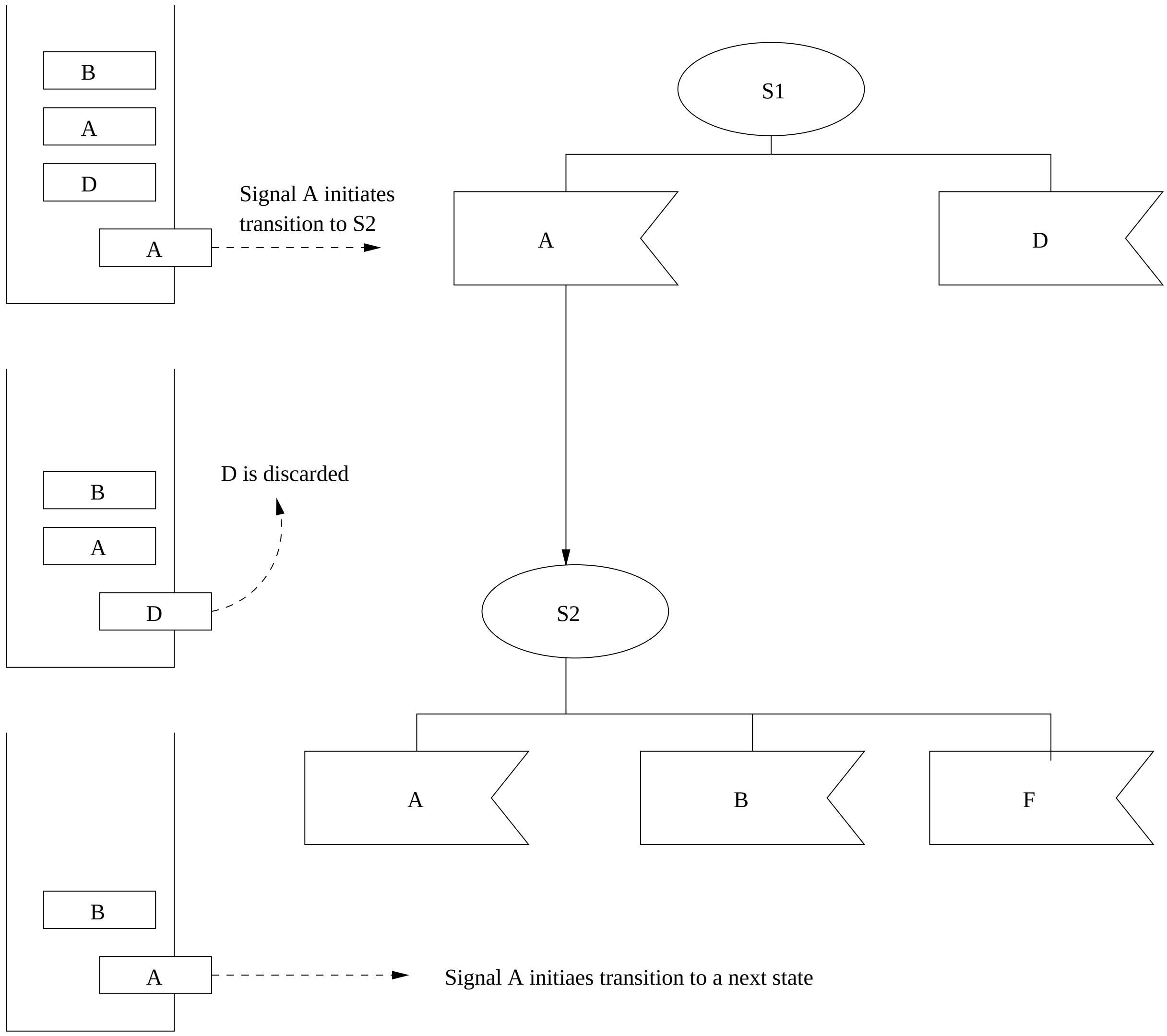

Figure 2.11: Signal Queue Mechanism

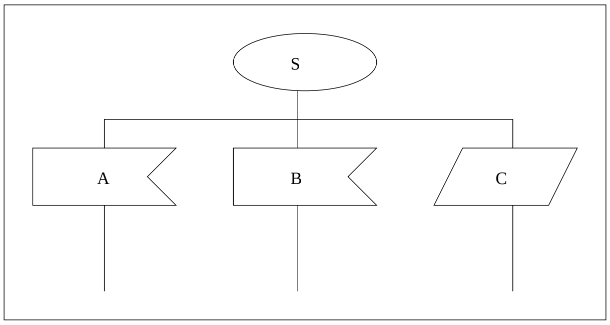

Figure 2.12: Example of the use of the *save* construct

a transition to state S2. Now the signal D is first in the queue. Since D cannot stimulate the process in state S2, it is discarded. The next signal is signal A, which can initiate a transition from state S2 to some other state (not specified in this example).

Sometimes discarding of a signal is *not wanted*, but the signal should be saved for future use. SDL has the *save* construct for this. Figure 2.12 shows an example. In state S the signals A and B are expected. If in state S a signal C is first in the input queue, it will not be discarded, but saved for future use.

The timer mechanism is related to the input queue mechanism. Figure 2.13 shows an example.

Suppose that during a transition the timer is activated by *set(x,T)* to the expiration time x. At the time x a signal with name T is put into the input queue of the process. The timer signal T is consumed by the process after all other signals which have been in the queue already at time x are consumed. That implies that it cannot be guarenteed that the process will actually be stimulated exaclty at time x, but some time after x. If, after the signal T has been put into the queue, a *reset* construct is interpreted, T is removed from the queue at once.

2.3 Modelling Systems using SDL-C

Design style is an important aspect of using any design tool. While SDL-C has capabilities of code execution, branching, looping and interprocess communication, it encourages a design style which is natural to such communicating processes. It borrows the features of simple block hierarchy, processes (state machines), states and transitions, sending and receiving of signals, branching and the use of synchronizing timers from SDL effectively to enable a natural design style for hardware systems.

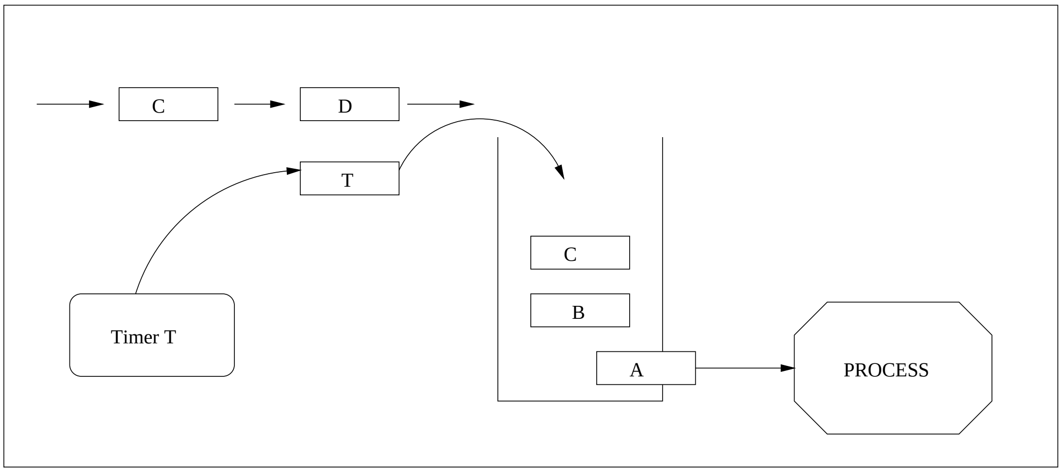

Figure 2.13: The timer mechanism

In SDL-C, processes communicate through signals. A signal, when fired by any process, is broadcast to all processes in its domain. Every process has FIFO signal queue, where signals accumulate. Signals are are either handled, rejected or saved. Signals may also have payloads — a feature that helps in generic design. SDL-C is equipped with C-style variables that may be shared between multiple process and may thus be used for communicating values.

Though not explicitly stated, SDL-C appears to encourage a design style where the systems may communicate with each other in ways which are quite common in actual embedded systems. These include:

- Shared Memory: All data which is global to a set of blocks or processes is automatically shared between them. It is, however, the user's responsibility to ensure mutual exclusion if required.
- Buffers: The signalling between processes in SDL-C is in the form of a signal queue. The queue of signals at a process can be looked upon as buffers where signals wait before being consumed.
- Timers and Clocks: In addition to interprocess communication using signals, special signals called timers and clocks can be used for synchronizing and providing an underlying time handle to the user.

- Emdebbed Code: Use of embedded code is allowed in the task blocks. Here the designer may use a subset of C code. Normally SDL alone is useful to implement controllers. Implementing datapath operations or software code (for embedded systems) in SDL itself is very tedious. This involves writing out an equivalent state machine for the embedded code which would have either an equivalent datapath in the ASIC (like a code for multiplication, encryption, compression, etc) or a software code in an embedded system. By the use of simple code blocks, which is natural to such equivalents, SDL-C makes SDL extremely suitable for designing at a high level. Other than the choice of SDL itself, this addition is the most important construct, which if used judiciously will enable very quick designs. However, the designer must have a clear idea of when to use C constructs and when to implement these in the original SDL state-transition structure itself. The manual also needs to provide more information to the designer so that maximal advantage of SDL-C may be taken.

2.3.1 The Arbiter Example

We now illustrate how SDL-C may be used for specification. The first example concerns the design of an *arbiter*. The problem is stated as follows:

There is a central resource held in non-sharable mode, which is required by various devices from time to time. There is an arbiter to handle requests from devices and grant the resource as and when possible. The resource cannot be granted to more than one device at a time.

There are some properties which must be adhered to in the design. The design should be free from starvation. In other words, all devices that request for the resource have to be served ultimately. Also, there should preferably be fairness in the deisgn. This means that no device should have to wait arbitrarily long before it gets its turn. Ideally two rules can be employed. One is first come first served and the second is user defined priority. The first two examples try to ensure the first come first served policy.

The protocol that we use for the design is as follows. Every device, sends a *Req* signal to the arbiter whenever it requires the use of the resource. Since the design is expected to be fair, it will be granted the resource ultimately. This is acknowledged by the *Grant* signal from the arbiter. The device goes through a blocking wait for signal *Grant*. When it receives that signal, it executes whatever task it had to perform with the resource and then indicates that it has completed its job by issuing an *Over* signal. The arbiter keeps waiting for an *Over* from the device that was granted the resource since it cannot handle further requests from other devices while the resource is in use.

Without payloads The arbiter can be designed in various ways to illustrate the different communication techniques that are featured in SDL-C. It is clear that SDL-C is capable

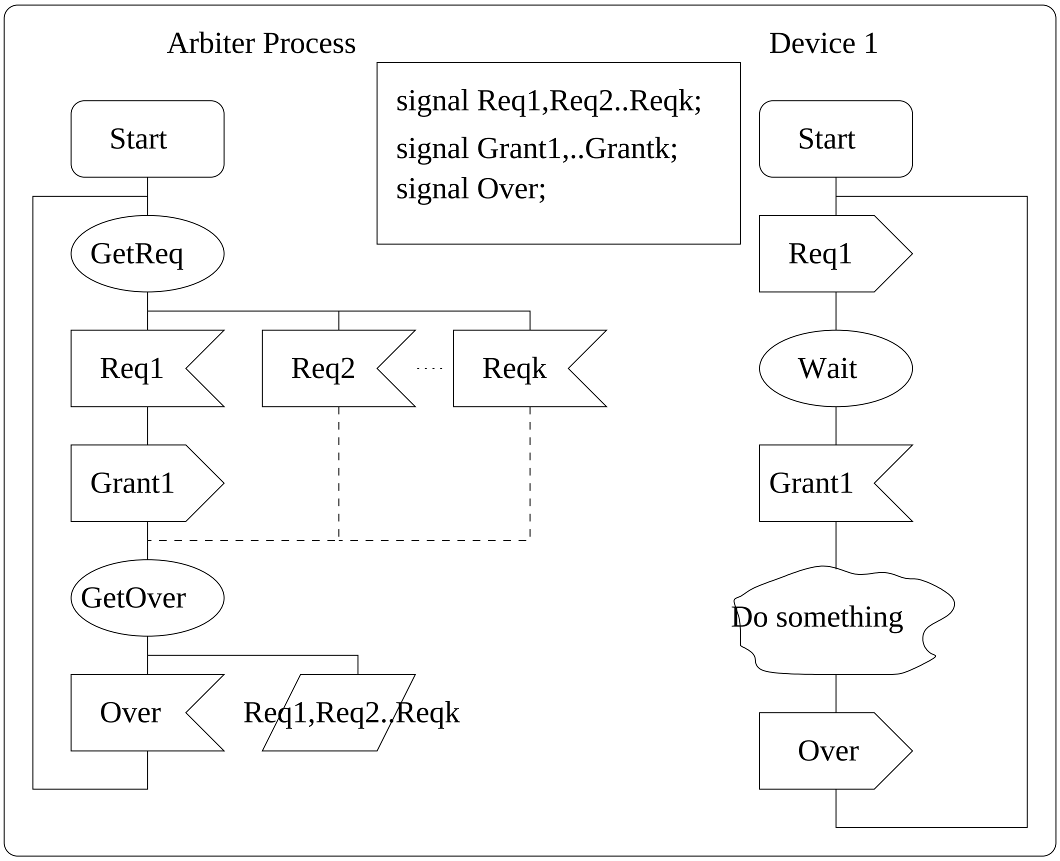

Figure 2.14: Arbiter Without Payload

of supporting different design styles corresponding to different hardware configurations. These include use of separate sensors to detect and store signals, etc. We discuss this issue later in the section. In the first example [Fig. 2.14], we do not use payloads for signals. Thus all devices cannot use the same request signal, since the arbiter needs to be able to identify the sender. Hence, the *Req* signal is enumerated over the domain of its senders, as Req_1 for device 1, Req_2 for device 2, and so on. Similarly, the *Grant* signal too is enumerated, to indicate which device is being served.

Thus, a particular device, say device i, will throw signal Req_i and wait in a state for the $Grant_i$ signal. Once it has received that signal, it holds the device for a particular number of time steps and then throws $Over$, indicating that it is through with the resource. The arbiter waits for requests from all possible devices. Whichever device requests first is served first by dispatching the special grant signal for that device. The arbiter then waits for $Over$ from the corresponding device. Note here that this signal need not be enumerated, since the arbiter may safely assume that the same device, which was granted the resource, is now releasing it.

To ensure that all device requests are served, the arbiter saves the request signal from every device *specifically*, at every state where such signals are not handled. Requests will always be served in the order in which they arrive. Thus, a critical design problem is handled very elegantly by this semantics. We observe the following.

Observation 2.3.1 *If the user is careful to save every unhandled signal at every step, then the FIFO semantics of the signal queue definitely guarantees fairness in serving of signals.*

The unbounded nature of the queue brings in some theoritical issues discussed later in this and subsequent chapters.

Observation 2.3.2 *The fairness of the design is taken care of internally by the FIFO semantics of the signal queue of the arbiter. This is an important feature of SDL-C and prevents possible state explosion if such a fairness were to be directly implemented by the user. For example, if the queue was a set and any available signal could possibly fire, then the design given above would not work always.*

The above example is simple, but it has been presented to highlight the importance of the basic inherent fairness that is captured in SDL-C. We now extend the design to include other features and make it more general. As is clear, the above design suffers from the obvious drawback that it is not generic. The arbiter has to be rewritten every time a new device is added and new signals are defined. We now show how that may be avoided with payloads, and how use of this novel feature makes designs simpler and more succint.

Payloads It is through the use of payloads that the user is relieved of the burden of enumerating signals of the sender domain, and the system lends itself to generic design. The general technique that is used [Fig. 2.15] is that every device has its own *id* or signature, which it communicates with every signal that it sends. Thus even if multiple senders throw the same signal, the receiver will be able to identify a particular sender for every signal that it catches. Also, a sender may use this concept of an *id* to indicate a partcular receiver for a thrown signal that has multple receivers defined in its domain.

The advantages are directly illustrated through the arbiter example. Here, it is now no longer necessary to enumerate the *Req* and *Grant* signals, which are instead defined with an integer value as a payload, and has all devices along with the arbiter in its domain. All devices now throw the same *Req* signal supplemented by *id*'s as signatures. The arbiter throws the *Grant* signal with the same *id* as the last *Req* it received. Since all devices are listening onto the same *Grant* signal, they have to decipher from the value of the payload which one among them is being served. Now, when a new device is added to the system, all that needs to be done is ensuring that it has an unique signature.

Observation 2.3.3 *Payloads play a major role in preventing state explosion and writing generic code in SDL-C. But use of unbounded integers as payloads creates problems during validation.*

Shared Variables Another paradigm that may be used for communication is the use of shared variables. We now demonstrate how one may use this concept in system design.

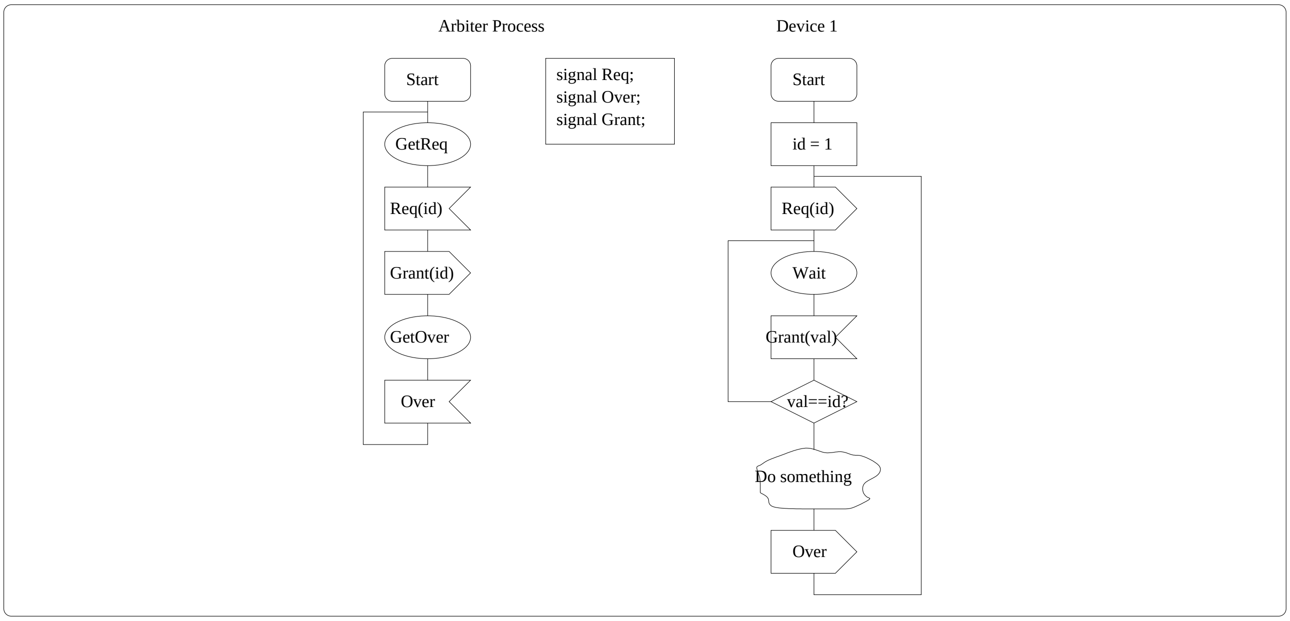

Figure 2.15: Generic Arbiter With Payload

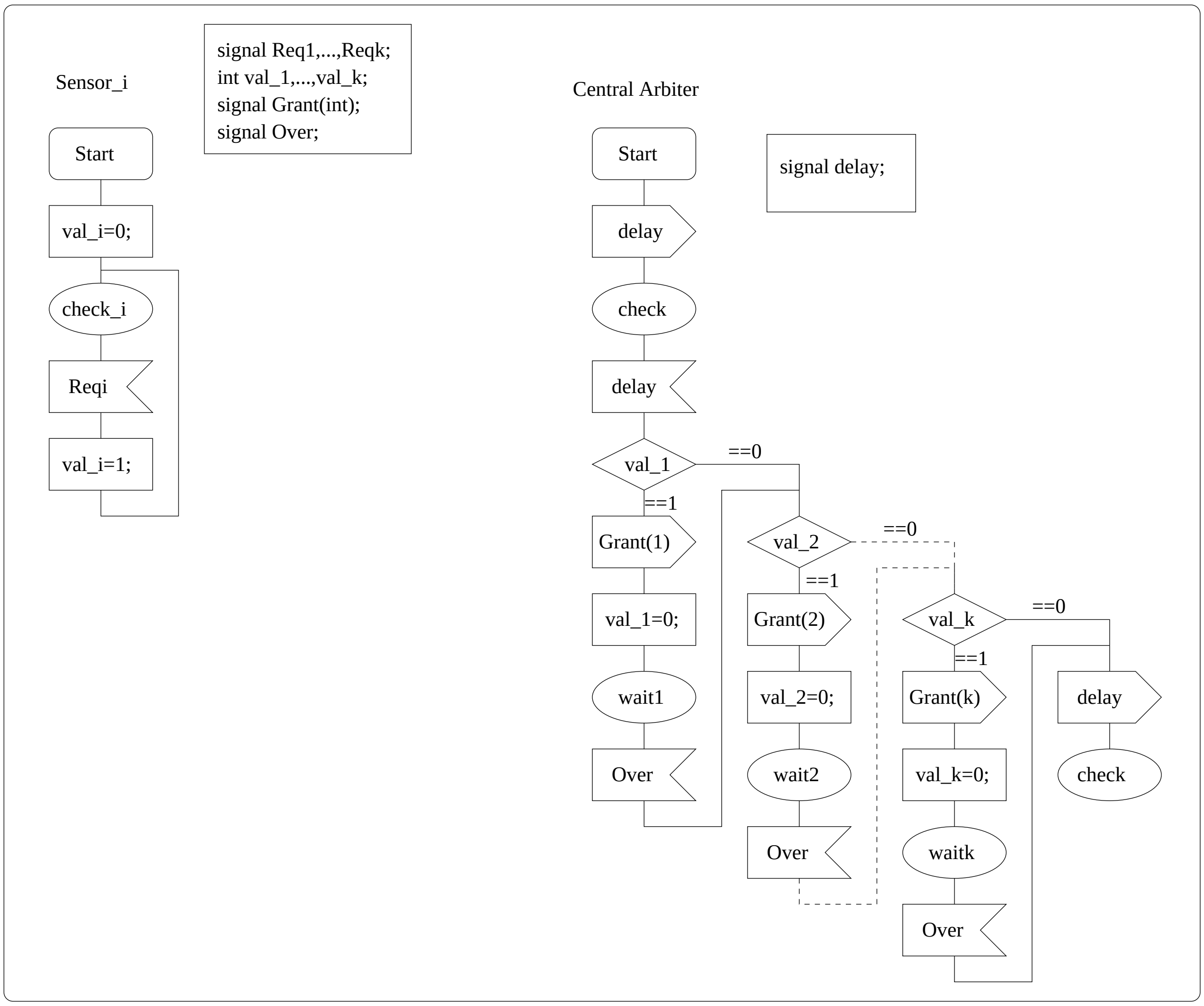

Figure 2.16: Arbiter With Parallel Sensors

Shared variables may actually be used to replace signals in design of synchronous systems, where state transitions occur not because of signals but in step with clock ticks. In fact, one can actually gain in variable complexity at the lowest level by using shared variables for signals that have one sender and multiple receivers. If signals are used, then every receiver will end up with a copy of the message. But this is really not necessary and we could do with just one variable which is read by all receivers.

Alternative design of an arbiter, which is now modelled as having parallel sensors for each of the devices is shown in Fig. 2.16. Of course, mutual exclusion has to be guaranteed while accessing the shared variables. The additional advantage that the designer gets with communication through shared variables is that he can always check which all signals are available — similar to peeking into the queue — and decide communication accordingly. This is in a way similar to DMA in that the arbiter may be compared with a processor which does not need to worry about some input device which has been granted the right to write directly onto some memory location. The arbiter can now concentrate only on deciding which device to serve. It does not need to wait for signals(interrupts). This example can also be made generic but would require use of C-code. The above example can be presented to show that SDL-C can accommodate different design styles.

Other Design Considerations Designers can also incorporate their own fairness conditions on top of the default fairness that SDL provides. What we had ensured [see Fig. 2.17] was that all devices that have issued requests are served in a round-robin fashion. In other words, no device will be served more than once consecutively, if there are requests from other devices pending, even if its requests have been received before that of the others. This we had implemented taking advantage of the blocks of 'C' code that SDL-C allows. Requests from devices are not served immediately on detection but are stored by the arbiter temporarily in an array. This array has an entry for every device keeping track of the count of pending requests from that device. The arbiter remembers which device has been served last, and traverses the array in a circular fashion, starting from that point, serving requests on its way. Obvioulsy, there is nothing sacrosanct about this fairness principle. We have implemented this because this principle is much used, but mainly to demonstrate what could be elegantly achieved through 'C' code. It is therefore clear that various types of control can be incorporated using code blocks in SDL-C. However, care should be taken to follow some design style.

The arbiter could be generalized to handle multiple resources in addition to multiple devices. The use of C-code for this would be more apparent. It is apparent that the use of C in SDL-C would prevent state explosion, which arises when we use only SDL or other methods like state charts.

Observation 2.3.4 *C code prevents state explosion. As an illustration, we can try to imagine how we would have had to incorporate round-robin fairness using just signals. Of course, it is possible — SDL is powerful enough to simulate a Turing Machine. But the designer would need to n states to remember the last n requests served, where n is the number of devices. However, C code must be discrimately used following a design style. It is necessary to identify the major blocks in a system and the flow of data and control between them. It is also necessary to identify the important states within a process and events, external or internal, that lead to them. However, states and transitions should be inserted only where absolutely necessary.*

Observation 2.3.5 *Signals local to a process do not have much use apart from emptying the signal queue and should be used sparingly. They do not have any intuitive significance either.*

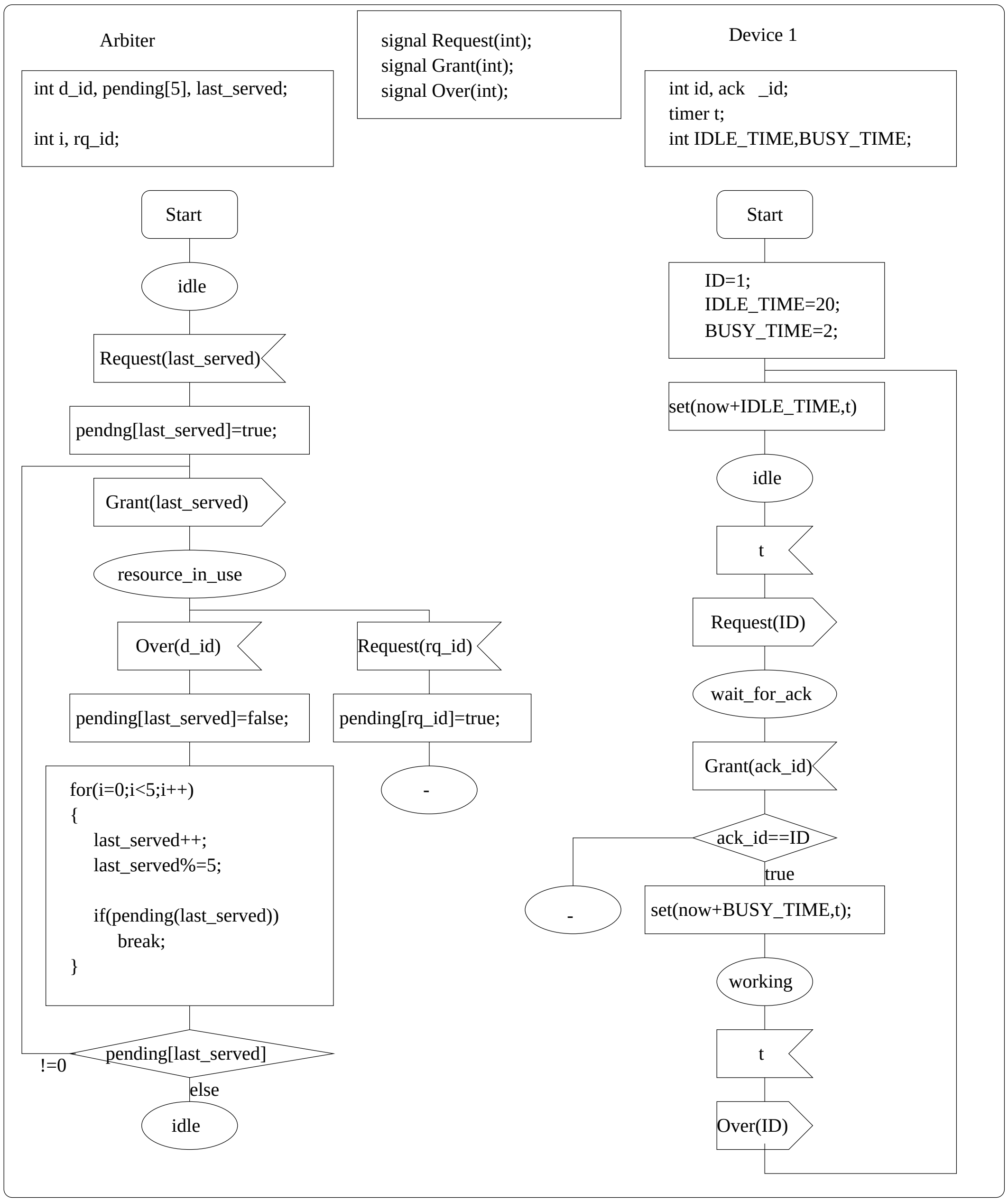

Figure 2.17: Fair Arbiter

Chapter 3

Overview of the Design Tool

3.1 System Overview

This section gives the reader an idea of the architecture of **Noodles**. The schematic diagram of **Noodles** is given in Fig. 3.1. Each component is brielfy dealt with in this section . The following chapters take up these in greater detail.

3.1.1 GUI

The syntax of SDL necessitates the inclusion of a GUI in **Noodles** for the purpose of taking input from the user.The GUI contains the following components (see Fig.3.2 for a snapshot of the GUI of **Noodles**):

- Canvas : This essentailly is the input "pad" for the user .Using the mouse and choosing the construct the user draws the specifications for the blocks and the processes in the system one is designing.
- Button Panels : Buttons (grouped together in panels) provide different functionalities to the user -
  - Construct Panel: This panel consists of buttons clicking on which the user chooses which contruct is to be drawn .
  - Manipulation Panel: This panel consists of buttons using which the user can change whatever he has inputted till now - i.e. buttons for edit,cut,paste etc.
  - Simulation Panel: This panel consists of buttons which start the simulator, debugger and the verification tools.

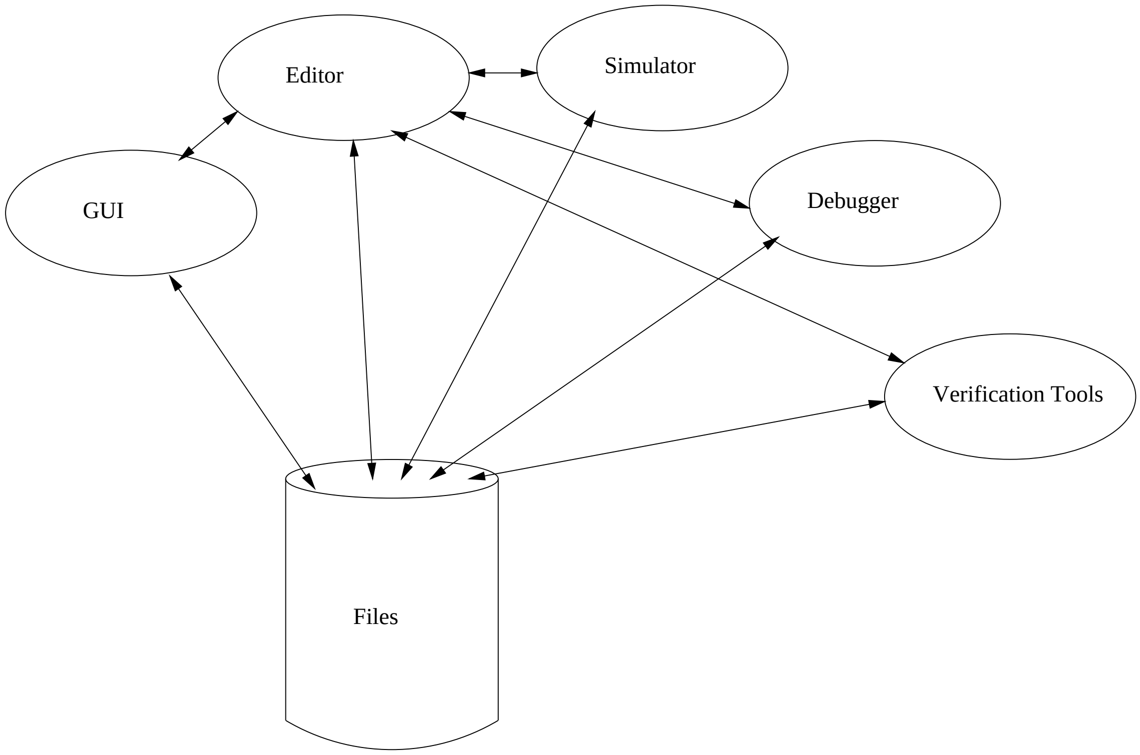

Figure 3.1: Schematic Diagram of **Noodles**

- File Panel: This panels consists of buttons which help in using the file system , i.e. save file , load file etc.

Some checking is also done while the user is using the GUI :

- Successor Checking: Only a subset of constructs are allowed by SDL semantics to follow any construct - this check is performed whenever a new construct is added or a new connection is made between two existing constructs.
- State-less Loop Checking: A state-less loop (i.e. a loop in the description of a process which does not have a *state* construct) is not allowed by SDL semantics - this check is performed when a new connection is made between two existing constructs.

3.1.2 Editor

As can be seen from Fig. 3.1 any component can communicate with the GUI only through the Editor componenet. It basically has two important parts :

- Action Listener: when the user clicks on any button in the GUI , some action has to be preformed - Editor defines the functions performing these actions.

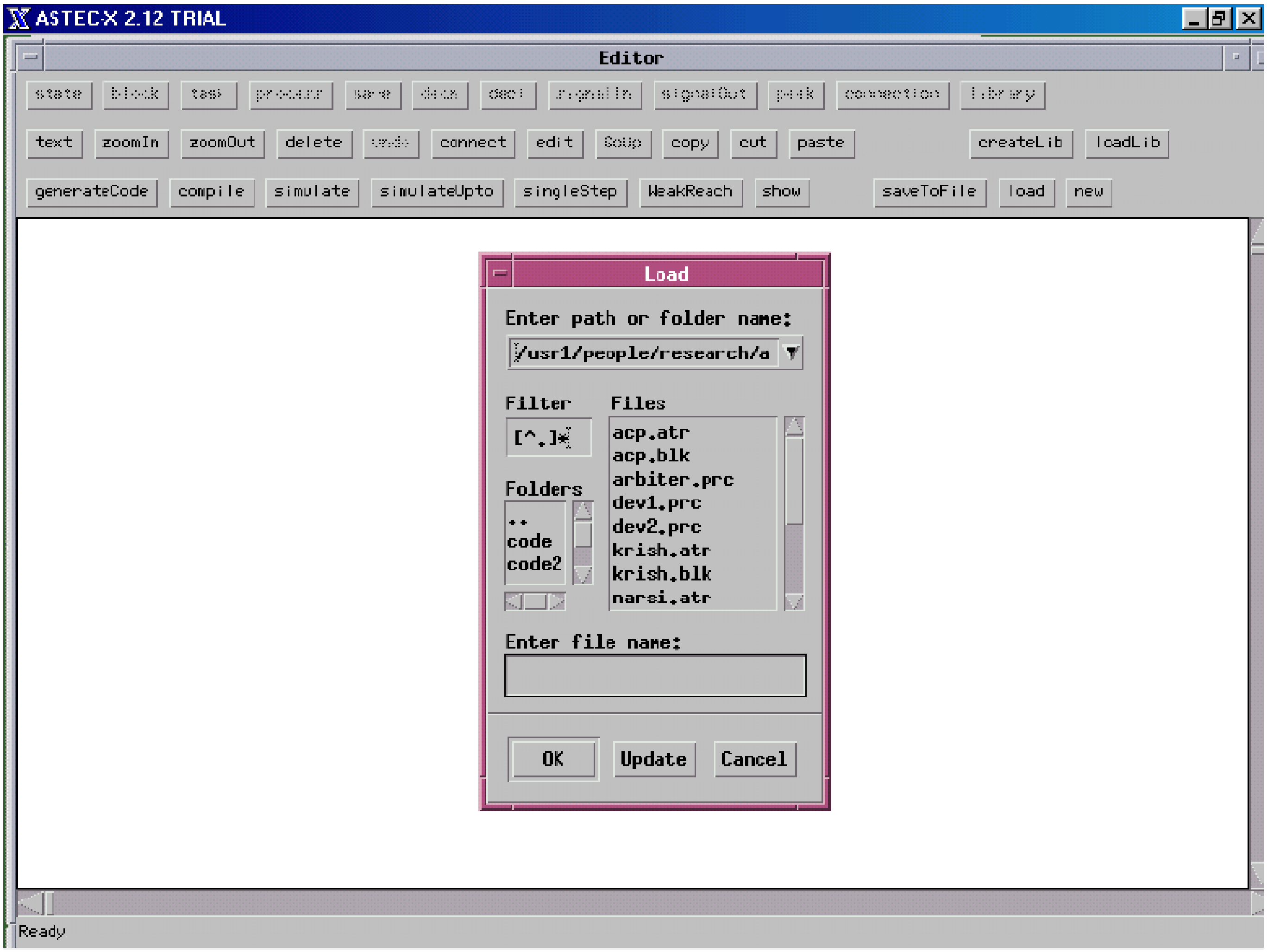

Figure 3.2: Snapshot of the GUI of **Noodles**

- Interface: As mentioned earlier the Editor provides an interface through which other components communicate with the GUI. Note that the Editor calls functions in the simulator, debugger and verification tools when the user clicks on the appropriate button in the Simulation Panel §3.1.1. Also any output of these called functions are re-directed to the GUI by the Editor.

The Editor also maintains the data structures for the GUI - two lists are used to capture the information about the input from the user :

- Constructs List: This list contains the information about the various constructs in the description of each process i.e. type of the construct, information about it's geometry (so that it can be drawn in the GUI) , links to parent (s) and children (or child).
- Connections List: This list does not contribute any extra information about the relationships between constructs in the process description (other than the text that can be present on the links) . This list is,however, necessary to store information about the geometry of the link (as in the general case a link in the GUI can be polyline).
- Hierarchies List: This list contains the information about the hierarchy in the system being designed by the user.

3.1.3 Simulator

The following steps are executed in **Noodles** for simulating the system description given by the user :

- Extraction: The system information which is available from the GUI has unnecesary (from the point of view of simulation) information regarding the geometry of the constructs while in the Connections list , almost whole of the information is useless§3.1.2. Thus the relevant information has to be extracted from the system description which is available from the GUI . This part can also be done in the Editor.
- Simulation: After the previous step is done the actual simulation can be done on the resulting description.The following points are to be noted in this respect -
  - The SDL sematics does not say anything about the scheduling of processes during simulation . We have used a round robin cycle based simulation using the concept of a scheduling cycle . Every process is given a chance for a single transition[1] based on it's current queue configuration . However, the signals

[1] This single step will henceforth will be called *single step execution*

thrown in a scheduling cycle do not affect the queue configurations of the other processes in the same cycle . The queue configurations of the processes are updated only at the end of the cycle. The variables are however updated in the same cycle[2].

– There are two ways in which the single step execution can be done :
  * Do the simulation based on the description graph got from the extraction step.
  * Generate code for each block/process in the system description . Note that converting the process description to code is very easy as the state machine corresponding to the process description can be represnted by a switch statement in a programming languauge.
– redirect the output to the GUI via the Editor.

In **Noodles**, simulation is done by generating code . This method was preferred to the one using the system description graph (got from the extraction phase) due to the following reasons among others:

- Simulation by executing code will be faster than using the system description graph. The problem will be more marked as the size of the design increases . This was an important factor as speed is of utmost importance in simulation.
- Simulation using the system decription will require parsing the statements in the task block . Also a symbol table for each process/block will have to be maintained for the variables . Using the second option eliminates this need by using the in-built facilities of the programming language (in the code will be generated) compiler.

Observation 3.1.1 *Data structures for the queues of the processes will have to be maintained in both the methods of simulation*

Observation 3.1.2 *A module for the simulator will have to be incorporated into the system description i.e. the simulation can be thought of as the original system specification along with a process for simulation.*

See Fig.3.3 for snapshot of a simulation being run on **Noodles**.

[2]The user is expected to make sure that there is no race condition while updating a shared variable §2.3

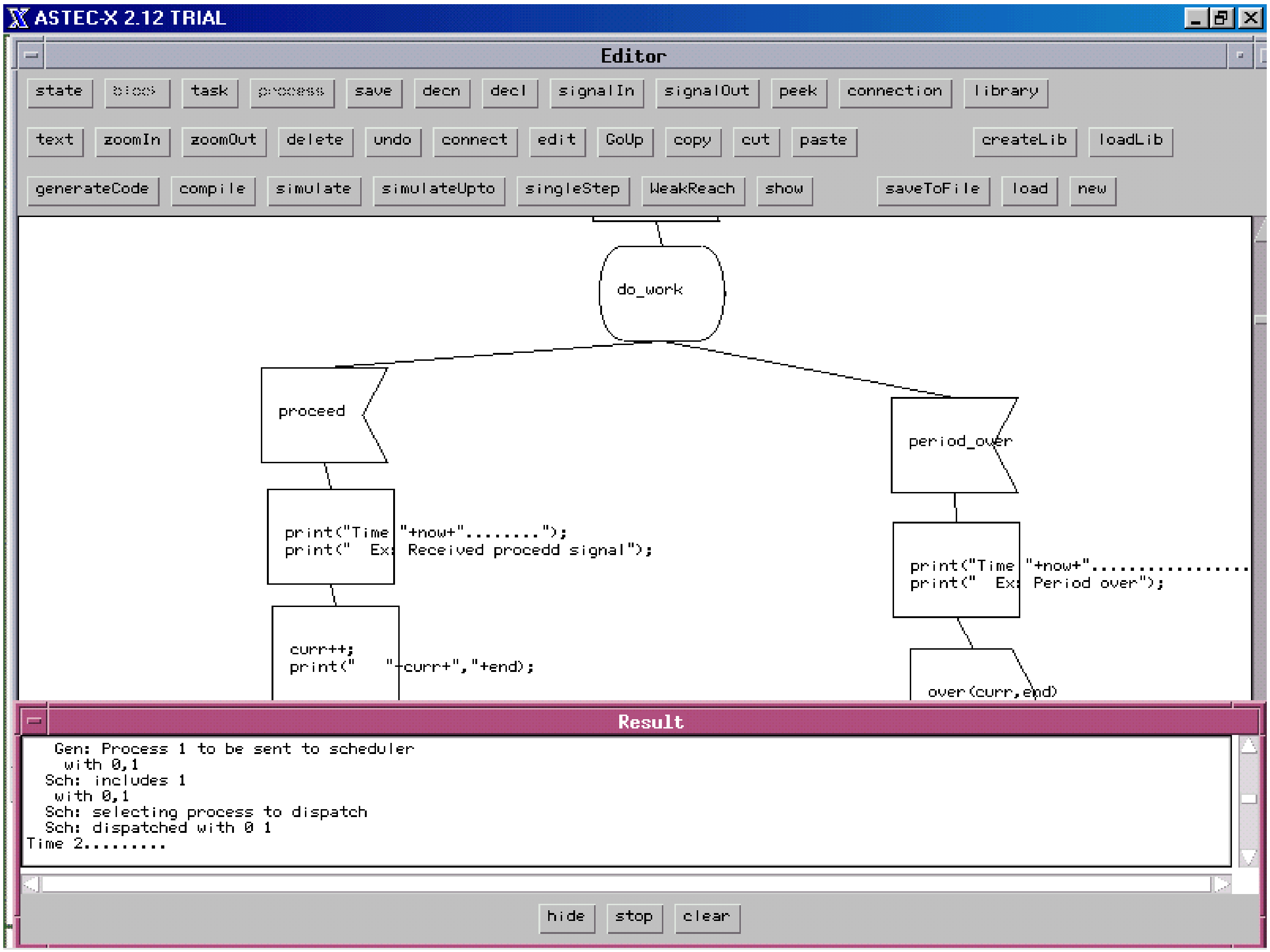

Figure 3.3: Snapshot of a simulation being run on **Noodles**

3.1.4 Debugger

A good debugger is necessary in any design tool(See Fig.3.4 for a snapshot of a debugging in **Noodles**) .The point to note here is that since SDL is a visual description language , the debugger should use the GUI (as was the case while taking input from the user) . The debugger should show the one step execution with results by tracing the execution on the GUI. The debugger can easily be made from the simulator with the following changes (the following discussion is valid if simulation is being done by generating code) :

- In simulation the the atomic transition is between states transition while here the atomic transition is between constructs. However to maintain the concept of cycle simulation , during debugging, a process is given chance till it makes a transition fron one state to another (if any is possible based on the queue confihuration)
- While debugging, after each atomic tranistion is made , the current construct of a process is highlighted.

3.1.5 Verification Tools

Though simulation is a means of verification , for large systems verifications tools are useful. One important problem in a SDL system which can be investigated is that of the reachability problem. However , it is known that this problem is undecidable [4]. Hence, we can hope to tackle this problem in two ways - either use a heuristic or restrict the domain of the problem so that the problem no longer remains undecidable . Both these ideas are used in the verification tools used in **Noodles** . The verification is divided into two parts - dynamic validation and static analysis .

Before we go into the description of these tools we define the notion of *global state* of a SDL system , which will be used in both static analysis and dynamic validation.

Definition 3.1.1 *The state of each process in a SDL system is defined by it's queue configuration , it's current state and it's control variables[3]. The global state of a system is then the set of all process states and the state of the Timer List.*

Dynamic Validation

In dynamic validation the idea is to do some checks while the simulation is going on . The following checks are made while **Noodles** is doing simulation :

[3]Control variables is the transitive closure of the variables in the decision boxes

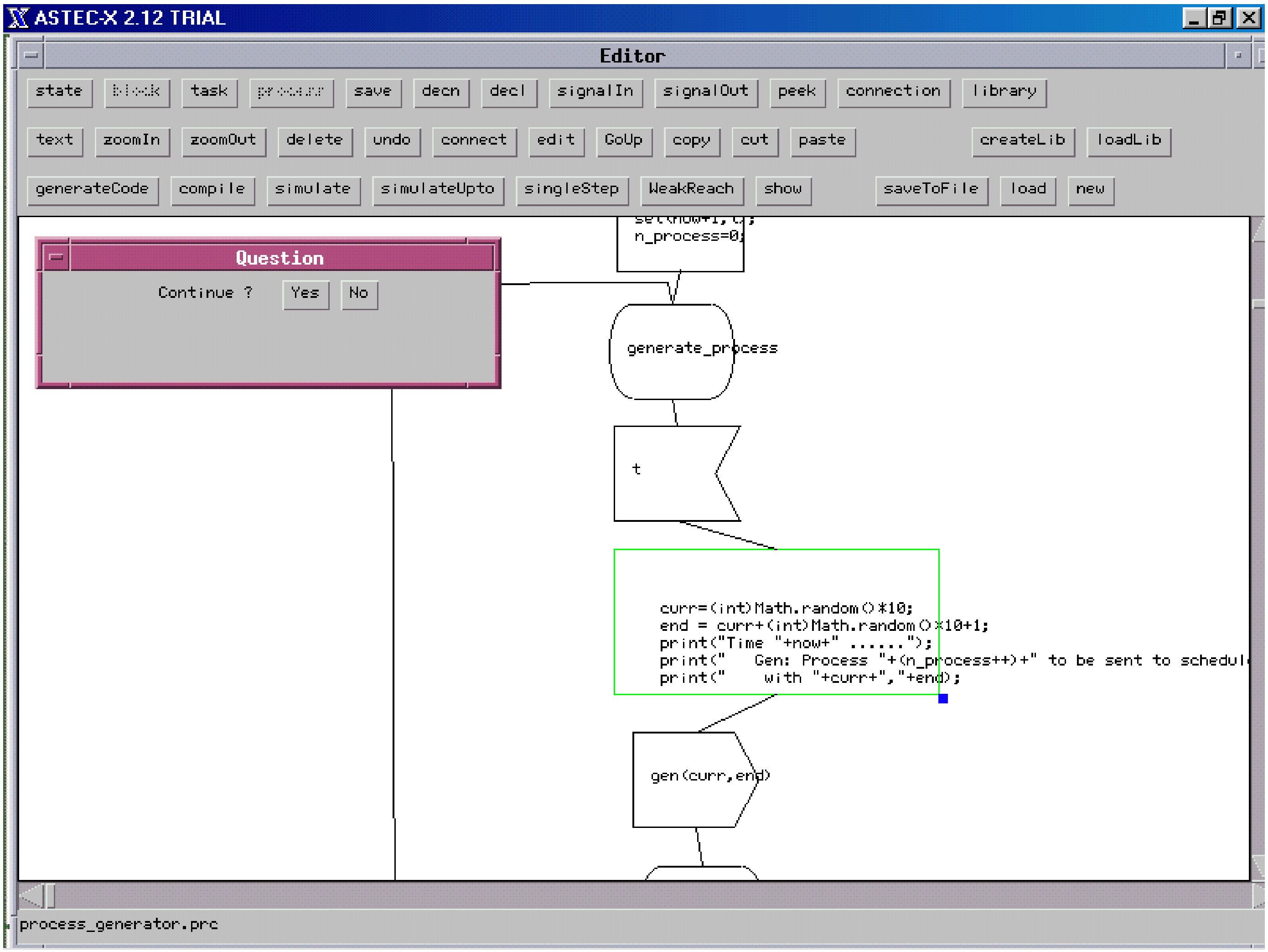

Figure 3.4: Snapshot of the working of the debugger in **Noodles**

- Loop Detection: During simulation the states which have been reached are kept track of . If during simulation , a previous global state is reached , simulation is no longer required to be carried out . The states unreachable at this stage will never be reached. For details see §7.1.

- Adverserial Scheduling: The drawback with the last check is that the check can be made for only one schedule (i.e. the one on which the scheduling is done which is assumed to be the same througout). The basic idea here is to choose a schedule which may lead the simulation to a deadlock - this schedule is called a *adverserial schedule*. The basic assumption is that signals are useful in any designed system. So at each step of the simulation an ordering is chosen for which significant number of signals are rejected. For details see §7.2

- Deadlock Detection: This is simple - whenever in a schedule cycle no transitons are made and the timer list is empty then a deadlock has occured.Note that this detection is for that particular schedule only.

Note that in Adverserial Scheduling we have used a heuristic.

Static Analysis

Although dynamic validation is useful, the user may like to have some static analysis tools for verifying his design . We have developed a static analysis tool for the reachability problem - here the new concepts used are :

- Control Flow Graph: The control flow graph of the process description is used for the analysis. The Control Flow Graph is essentialy this :
  - Only the *state, signal in ,signal out & save* contructs remain.
  - Every *decision* construct is replaced by a choice point (i.e. all the children of the *decision* construct in the process description become the children of the parent(s) of the *decision* contruct in the Control Flow Graph).

- Signal Count: We are replacing the queues of each process by a signal count array. However, the relative position information about the signals in a queue is lost. Note if we are using signals count instead of the queue, our results will be valid for all possible schedules[4].

[4]The ordering among the processes will manifest itself in the relative positions of signals thrown after a scheduling cycle in the process queues.

- Frontier States: In this analysis, a process no longer has a single current state but a set of current states called the frontier state[5].

In the analysis we do a mock simulation in the following manner :

- Transitions: In the mock simulation, we consider each state in the frontier state individually - for each state, a transition is made if the corresponding signal count is greater than zero. All such possible transitions are investigated. For each transition that is investigated , the corresponding signal count is decremented by one. Also in the transition for each signal thrown the corresponding signal count is incremented by one .
- Choice Points: These are the choice points which replace the *decision* constructs in the original process description. Whenever such a choice point is encountered , all the outgoing paths are investigated in parallel.

Initially the frontier states of each process is the *start state*, all the signal counts are zero. The mock simulation is then started . The mock simulation is terminated when the halting conditions are met §6.3.4.

Note that any state found unreachable by this analysis will actually be an unreachable state across all schedules. However, an unreachable state may be reported to be reachable. This is why we call this analysis the *Weak Reachability Approach.* See Fig.3.5 for a snapshot of the results of Weak Reachability Analysis.

For details see Ch.6.

3.2 Additions in SDL Specifications

Though SDL-C provides with many constructs , there are still some which eases the desgin process. It must be noted that these *additions do not add to the sematic power of SDL.*Two such additions which have been implemented in **Noodles** which are described below.

3.2.1 Peeking into the Queue

As a more general case, the user may be allowed to peek into queue — to check if a signal is there at a particular position — without consuming any signal. Consider designing a

[5]Now a process makes a transition from one frontier state to another frontier state.

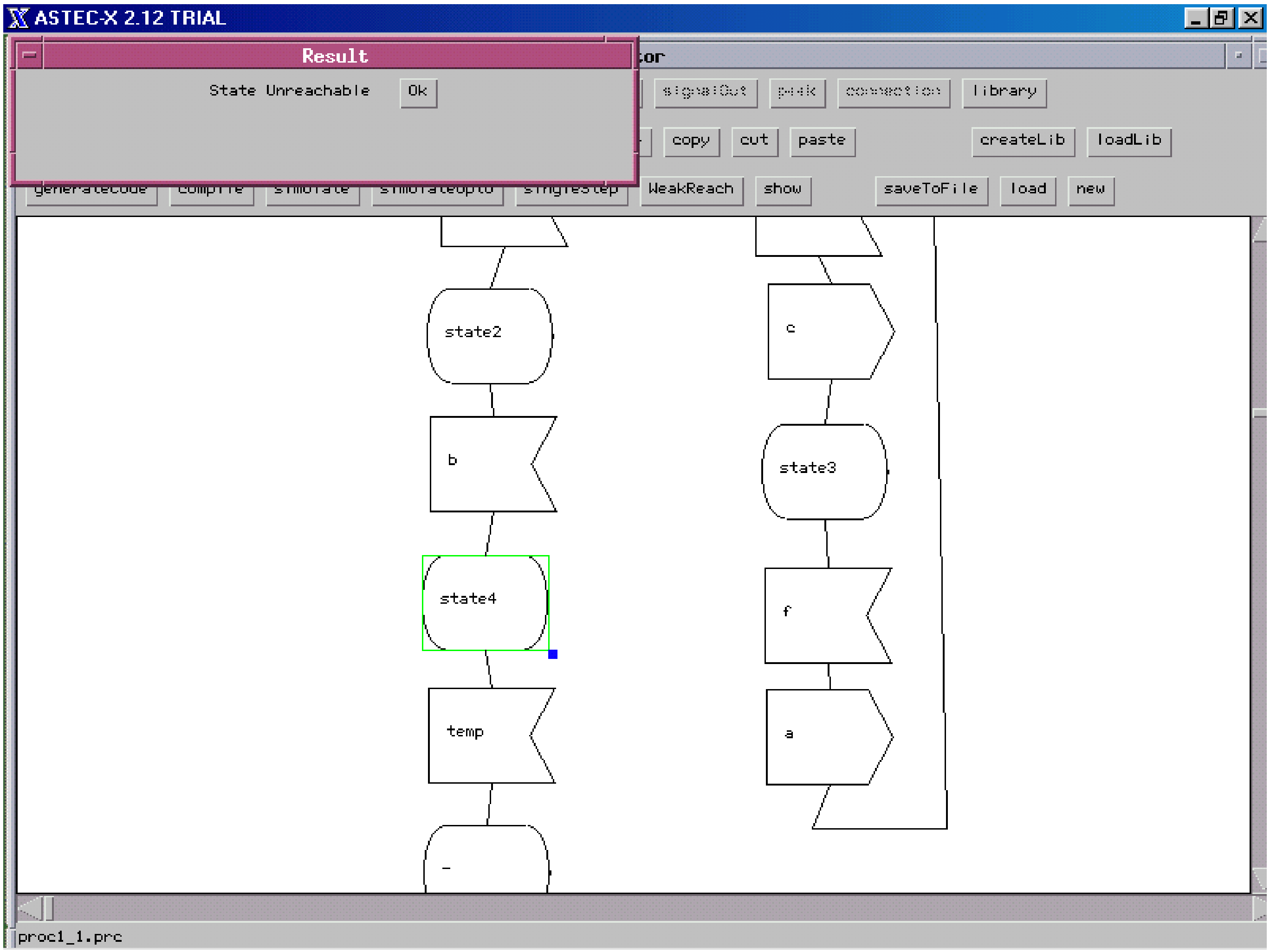

Figure 3.5: Snapshot of the results of Weak Reachability Analysis in **Noodles**

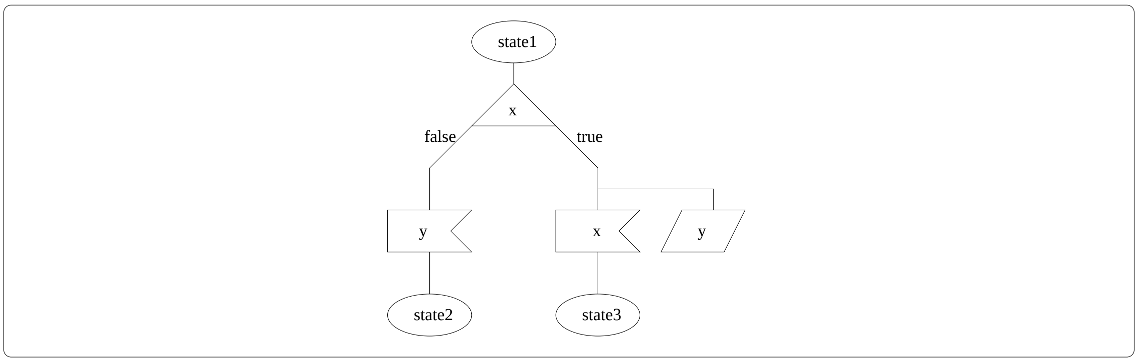

Figure 3.6: Using peek to implement Signal Priority

process scheduler in SDL-C, having a ready queue of processes, which are dispatched in a round robin fashion for fixed time quanta. Between dispatchings, the scheduler checks if any new processes have sent a signal to be included in the ready queue. Note that here the scheduler cannot wait for such a signal indefintely. It has its own task to perform even there is no such signal currently. So it needs to know if a particular signal is there in its queue, or not. That is, it needs to check the absence of a signal.

Hence we suggest a new *peek* construct, which may be added after a state. Each *peek* box is much like a decision box — it checks a condition and returns either *true* or $false$. In fact, one may actually have a binary tree of peek boxes after a state with some of the leaves evaluating to *true*. Those we call the *enabled leaves*. A leaf has to be followed by a *signal-in* construct. At any simulation step, only the paths leading out from the enabled leaves are considered as currently defined transitions from that state. Signals may also be saved in such a state, as usual.

As an example, we design a signal priority using the *peek* construct. Suppose a state s in a process is such that two signals, say x and y, could be present in the signal queue at that simulation step. Suppose signal x has a priority over signal y. Then, as illustrated in Fig. 3.6, if only y or x is present, that is the signal which is handled. But if both are present, then whatever their order may be, x has to handled saving y.

In the example, note that the signals are saved corresponding to a path and that signals which are specified to be saved in paths that evaluate to false, are not saved when exiting via a true path. There are alternative semantics, of course. Signals could also be specified as saved corresponding to a state, which will relieve the user of the burden of saving them explicitly in every path that evaluates to true.

3.2.2 Library

Another feature, whose need was felt time and again during the design phase, is a *parameterised process/block library.* Consider the generic arbiter. All devices are exactly identical except for an unique identifier value. And yet, a library construct is absent then has to go over the same design laboriously every time one needs to add a new device to the system. It would have been much convenient if one could design a generic device with its identifier once and for all and store it in the library. All that need be done when a new device is to be added is an instantiation of it with a particular identifier value as the actual parameter. There will definitely be many such typical blocks in any generic design which need to be *reused* from time to time.

The user should be allowed to add parameterized blocks to a global library or to a personal library of his own. In order to use it at any point of his design, he simply has to add a library construct within a parent block needing nothing more than a button click. He specifies the name of the library component that he is using within the construce, along with the actual parameters (very similar to a function call), and an instance of this library component is added to the appropriate point of his design tree, with the substituted parameters (like a macro substitution).

3.3 Programming Language

Java has been used to implement **Noodles** . The code generated during simulation is also in Java.

Chapter 4

The GUI and the Editor

4.1 Introduction

This chapter goes into the details of the design and implementation of the GUI and the Editor components . The format of the files in which the system decription is stored is given in Appendix A.

4.2 The GUI

Since SDL is a visual decription language, a GUI is necessary for taking input from the user.

4.2.1 Components

The components are described in §3.1.1.

4.2.2 Checks Performed

As mentioned in §3.1.1, some simple checking is done while the user isdone while the GUI is taking the input from the user. Two checks which are performed are:

- **Successor Checking:** The SDL semantics allows only a subset of constructs to follow a construct in the process description (for example, only a *signal in*[1]). This check has to be preformed whenever the user inserts a new construct or a new connection is made between two constructs in the process description. Performing this check is trivial - a global matrix is kept which records the information regarding which subset of constructs can follow a particular construct. Whenver a new construct or a connection is added this matrix is checked.

- **State-less Loop Checking:** SDL semantics does not allow a *state -less loop* in the process description. One way of checking this is to extract cycles from the process description graph and then checking if all the cycles have a *state* construct as one of the nodes. However, extracting all the cycles in a graph is a hard problem[2].

  However, if we attempt to do this checking while the user is using the GUI, the problem becomes much simpler. This claim becomes clear if one notices that a loop can be created in the process description graph only when the user adds a new connection between two existing constructs. So if it is known that the process description graph before adding a new connection is *state-less loop* free then one needs to only checks whether the newly added edge (i.e. a new connection) introduces a *state-less loop* . This check can be done easily in the following manner.

  Algorithm 4.2.1 *Let G_i be the the process description graph at any stage i. G_i does not have any* state-less loop. *Let the new edge added be (v_j, v_k). Do breath first search on G_i starting from node v_k.*

  If there is no path from v_k to v_j then G_{i+1} does not have any state-less loop

  else check if all the paths from v_k to v_j have a state construct as a node in the path. If so then G_{i+1} does not have a state-less loop *else G_{i+1} has a* state-less loop.

  Finally note the G_0 does not have any state-less loop.

4.3 The Editor

4.3.1 Components

Components are described in §3.1.2

[1]The construct *peek* which we have introduced can also follow a *state* construct

[2]Atleast this problem is harder than the Hamiltonian Cycle problem.

4.3.2 Data Structures for the GUI

As mentioned in §3.1.2 , the Editor mantains two lists to cature the full information about the system description which is input from the user:

- Constructs List: This list contains the information about the constructs in the process description graph. Fig.4.1 shows the Constructs List with one element of list shown. The fields of the *Construct* object are :
  - *Bounding Rectangle:* This field captures the geometrical information about the construct. The idea behind using a rectangle to capture this information is to abstract out the individual differences among the constructs. The *Type* field is used to differentiate the constructs (*Bounding Reactangle* and *Type* together are used to draw any particular construct). All the operations on the constructs (for example dragging/moving of constructs, changing the dimensions of constructs) are performed on the *Bounding Reactangle.*
  - *Type:* This field gives the type of the construct.
  - *Selected:* This flag signifies whether the construct has been selected by the user : if this flag is on, then while drawing the construct, it is highlighted.
  - *Text:* This field contains the text inside the construct.
  - *Child List:* This list contains pointers to objects in the Constructs List which are children of the this construct (i.e. constructs which follow the current construct in the process description).
  - *Parent List:* This list contains pointers to objects in the Constructs List which are parents of this construct.
- Connections List: This list contains information about the connections in the process description graph. Fig.4.2 shows the Connections List with one element of the list shown. The fields of a *Connection* object are:
  - *X-coord List:* This is the list of x-coordinates of the polyline representing the connection.
  - *Y-coord List:* This is the list of y-coordinates of the polyline representing the connection.
  - *nPoints:* Number of coordinates in the polyline representing the connection.
  - *Text:* The text present on the polyline (this is present in outgoing edges of a *decision* construct.
  - *Parent:* The source of the connection - this is a pointer to the parent object in the Constructs List.

  - *Child:* The destination of the connection - this is a pointer to the destination object in the Constructs List.

- Hierarchies list: This list stores informatation about the hierarchy in the system designed by the user. Note that this list is actually a tree and all the internal nodes are description of blocks while the leaves are process descriptions. Fig.4.3 shows an example. The list has elements of type *Hierarchy* object which has the following fields :

  - *File Name:* This is the file name in which the description of the *Hierarchy* object is stored.
  - *Parents:* This is a list which has pointers to parents to this object.
  - *Children:* This is a list which has pointers to children of this object. Note that if the object corresponds to a *process* then this list will be empty.
  - *Nodes:* This is the Constructs List for this node in the Hierarchy in the system.
  - *Links:* This is the Connections list for this node in the Hierarchy in the system. Note that if the object corresponds to a *block* this list will be empty.

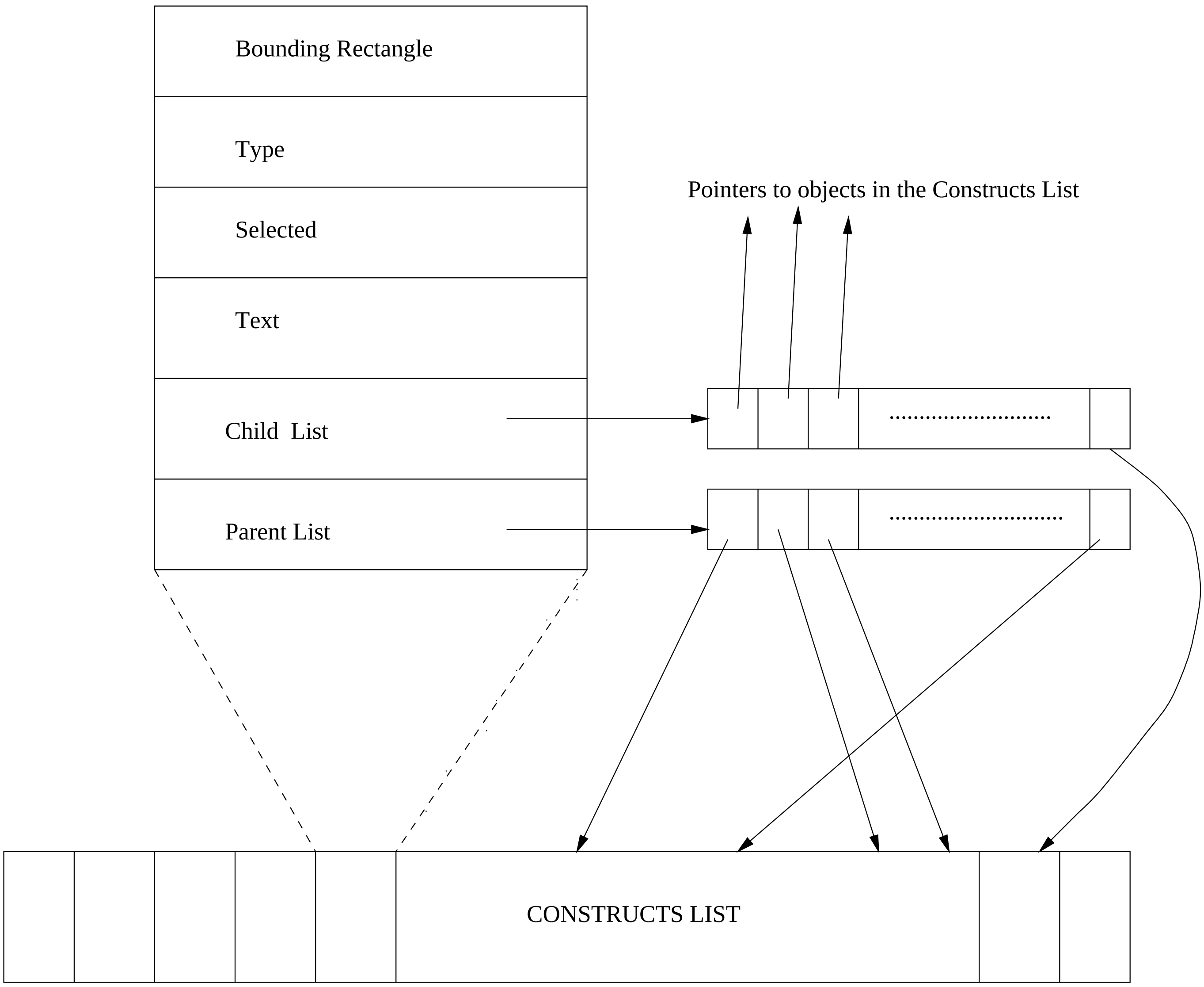

Figure 4.1: *Constructs list* with one *Construct* element shown

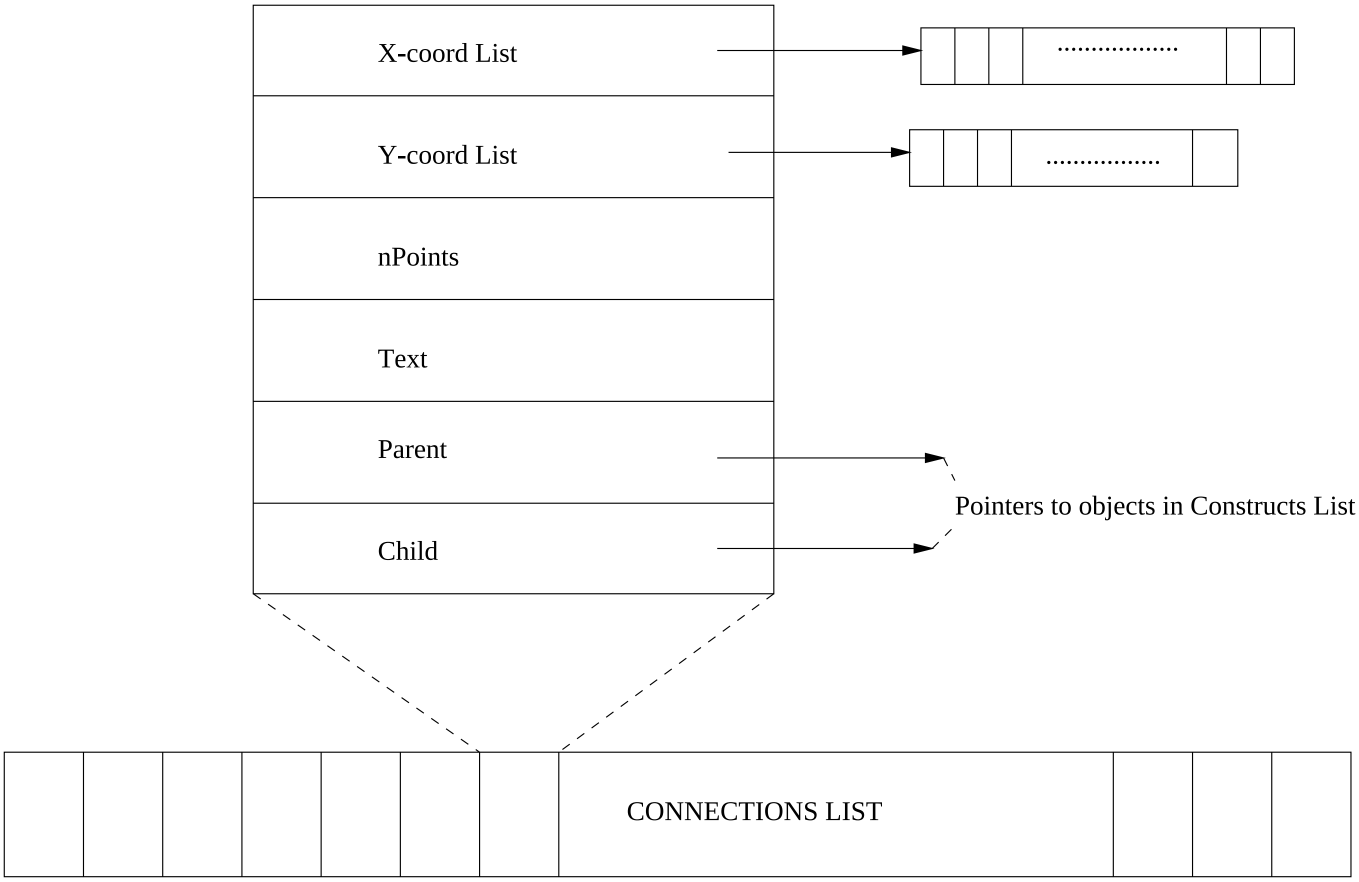

Figure 4.2: *Connections list* with one *Connection* element shown

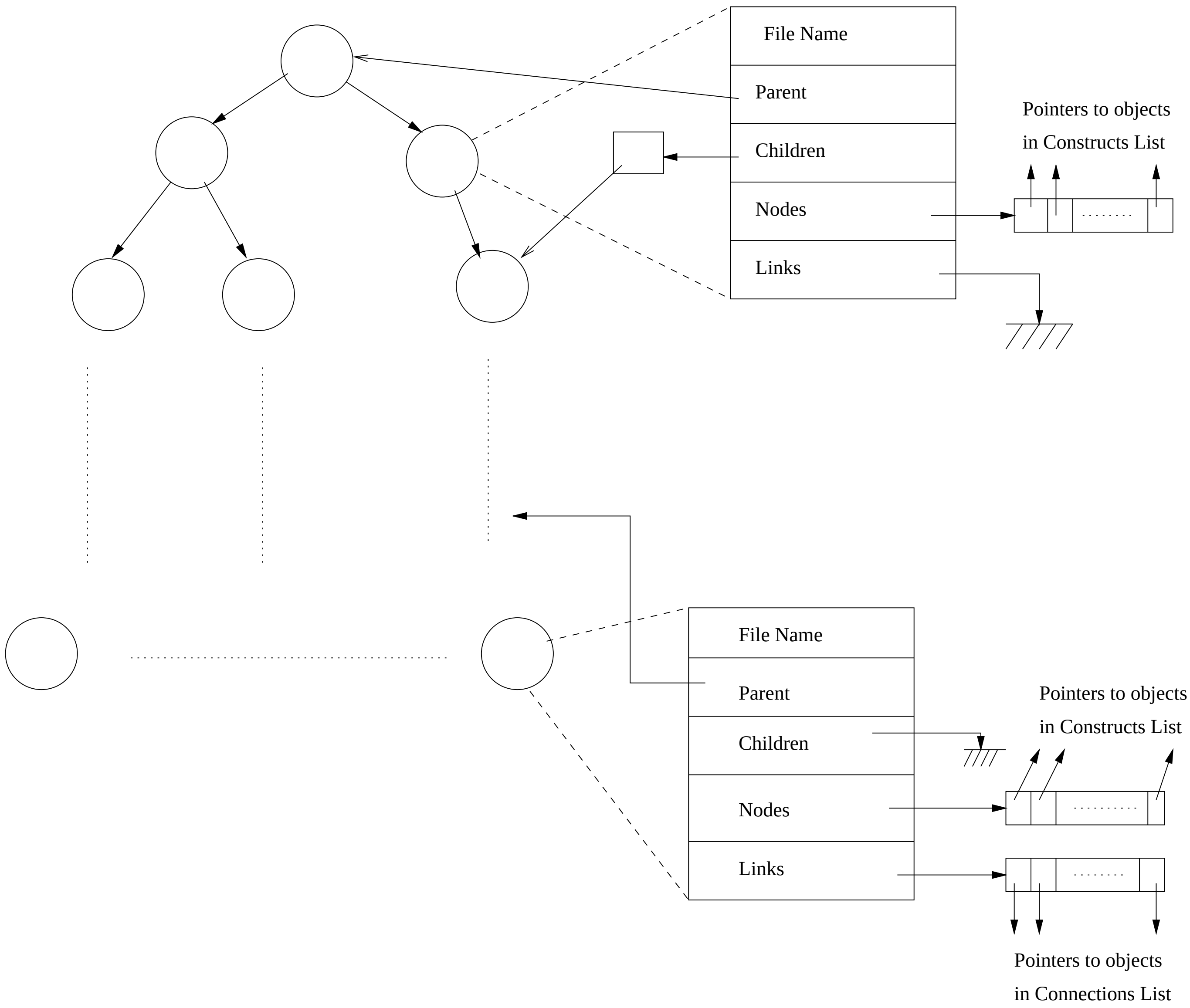

Figure 4.3: *Hierarchies List* with one *Hierarchy* element for one *block* and one *process* shown

Chapter 5

The Simulator and the Debugger

This chapter goes into the details of the design and implementation of the Simulator and the Debugger of **Noodles**.

5.1 The Simulator

5.1.1 Extraction of System design

As mentioned in §3.1.3 , the first step in simulation is the extraction of the system design from the data structures which capture the system in the GUI stage - the process/block description is captured by the *Constructs List* (see Fig.4.1) and the *Connections List* (see Fig.4.2) while the *Hierarchies List* (see Fig.4.3) captures the system hierarchy.

As shown in Fig.5.1 , the information about the system is extracted from *Construct* and *Connection* and stored in a *Process Node.* Note the following points

- Only those information pertaining to the geometry of the constructs and connections are discarded.
- Although in Fig.5.1 *Construct* and *Process Node* share the child list and the parent list, they do not have pointers as elements (in which case they cannot be shared) but have indices as elements (in which they can be shared)
- The text in the *connection* object goes into the appropriate element in the *Conditions* list such that the following hold:

- Let ob_1 be the *Construct* object , ob_2 be the *Connection* object and ob_3 be the *Process Node* object under consideration.
- Let i be the index such that $ob_3.Children[i] = ob_1.Child$.
- Then $ob_3.Conditions[i] = ob_2.Text$.

As shown in Fig.5.2, the system hierarchy information is stored in *Treenode* object. This information is extracted from the *Hierarchy* object in the GUI stage. The correspondence between *Hierarchy* object and *Treenode* object is explained for each element of field of a *Treenode* object :

- *Symbol Table:* This is a hash table which contains information about the variables defined in the current block/process in the system. This information is extracted from the *decleration* construct in the corresponding *Constrcuts* list.
- *Type:* This is flag indicating whether the current node in the system hierarchy is a *process* or a *block*.
- *Process Graph:* This captures the process description - this list has elements as *Process Node* objects.
- *Parent:* This is a pointer to the parent *Treenode* in the hierarchy.
- *Children:* This is the list of children *Treenode* objects in the hierarchy.

5.1.2 Simulation

The following basic steps are performed for simulation:

- Generate code for each node in the hierarchy[1] .
- Generate ordering among processes.
- Perform initialization for the data structures of all the processes.
- Do single step execution[2] of each process according to the ordering obtained in the previous step.

[1]Note that code is also generated for the simulator
[2]indefinitely if no time limit for simulation specified by the user else till the time limit specified expires

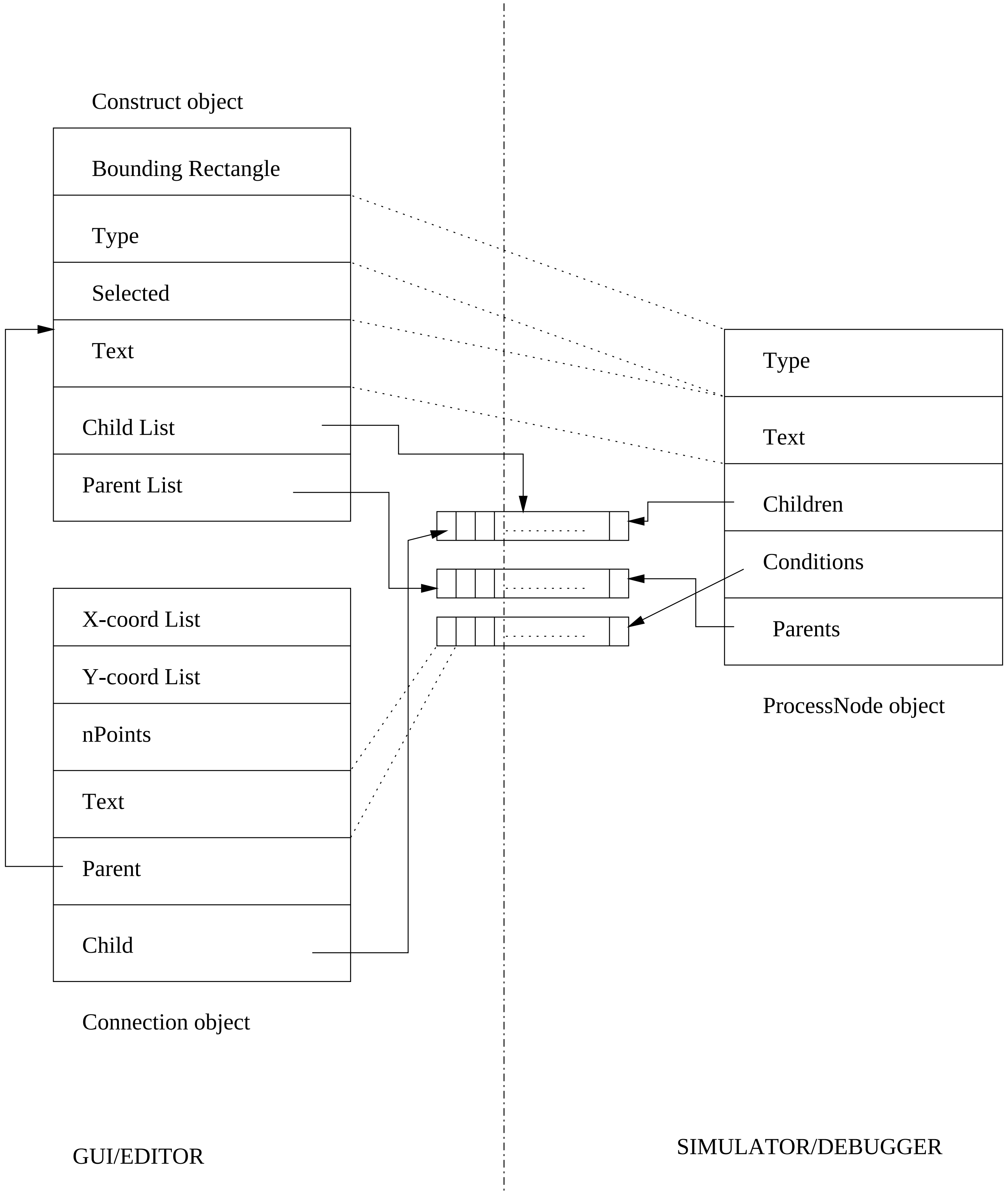

Figure 5.1: *Process Node* composed from a *Construct* object and a *Connection* object

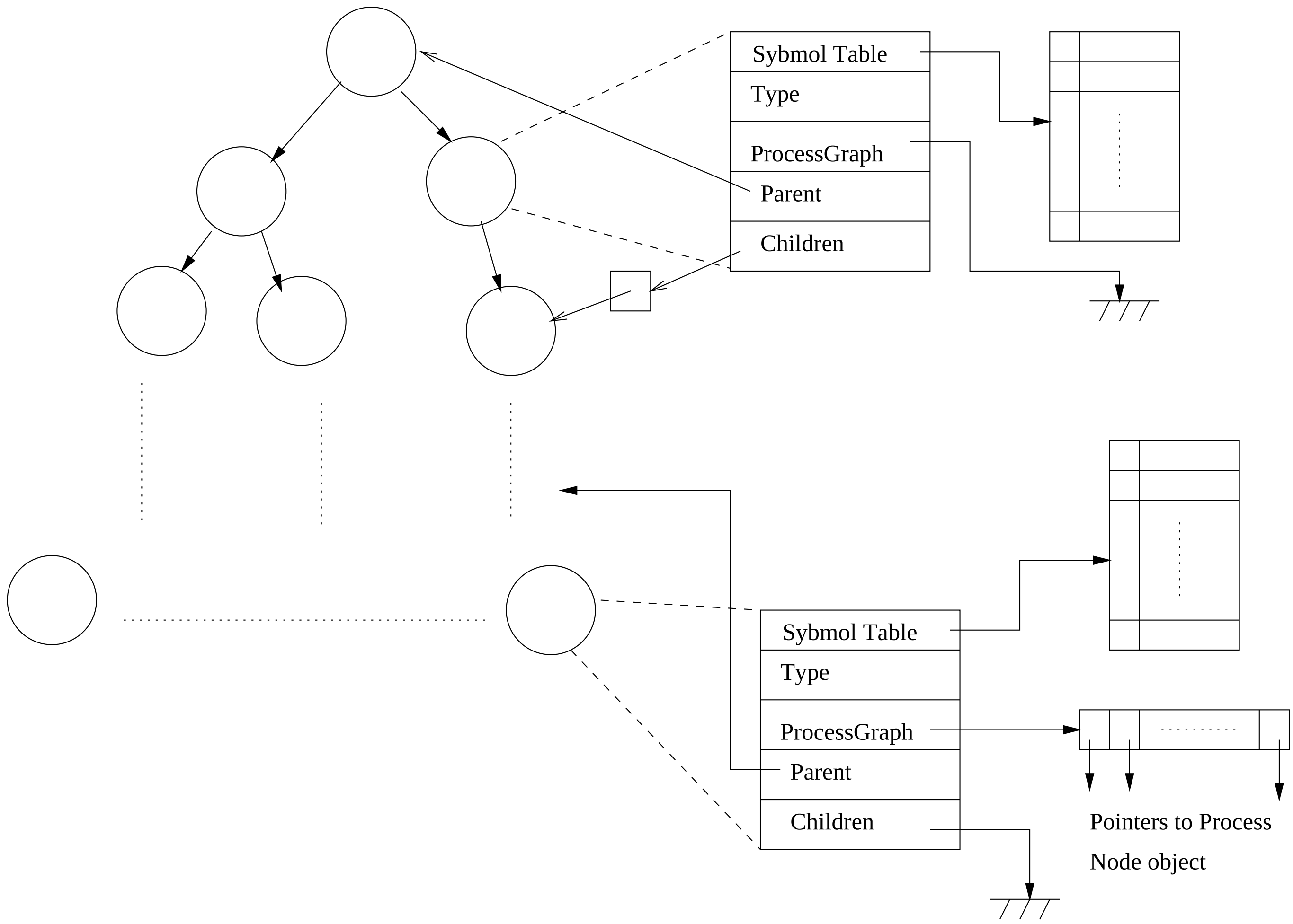

Figure 5.2: *Dependence Tree* with *Treenode* object shown for a *block* and a *process*

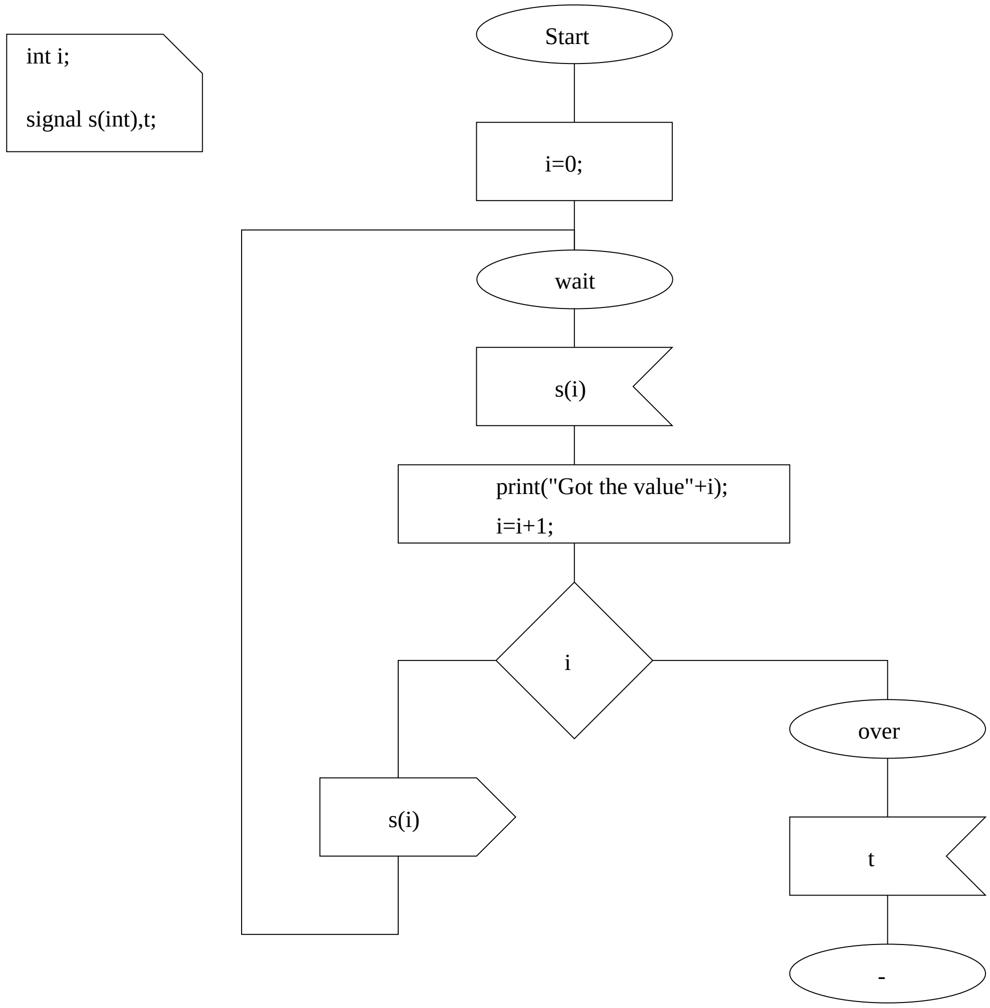

Figure 5.3: A sample *process* description

Fig.5.3 shows a sample process description . The code generated for the above description for simulation is given below.

```
public void oneStepExecution(int nodeNo)
{
  switch(currState)
  {
       case STATE_start :
            Simulator.Transition = true;
            i= 0 ;
            currState = STATE_wait;/* Next state is wait */
            break;
       case STATE_wait :
            transitionMatrix[currState][Simulator.Signal_0]=true;
            /* Tranistion from state wait is possible through signal s */
            /* Code for getting copy of signal s in NODE_s - if no copy then exit */
            Simulator.Transition = true;
              switch(NODE_s.id)
                {
                  case Simulator.Signal_0 :   /* Signal s is present */
                    i = ((Integer) NODE_s.payLoad.elementAt(0)).intValue();
                    /*i has the value of the first payload of signal s*/
                    print ("Got the value "+i);
                    i=i+ 1 ;
                      if(i <10)
                      {
                        thrownSignal = new signal(Simulator.Signal_0,null);
                        thrownSignal.payLoad.addElement(new Integer(i));
                        Simulator.thrownSignalQ.insert(thrownSignal);
                        /* Signal s with i as it's payload is thrown */
                      }
                      else
                      {
                        currState = STATE_over;  /* Next state is over */
                      }
                  break;
                }
            break;
       case STATE_over :
            transitionMatrix[currState][Simulator.Signal_1]=true;
            /*Transition made from state over through signal t */
            /* Code for getting copy of signal t in NODE_s - if no copy then exit */
            Simulator.Transition = true;
```

```
            switch(NODE_s.id)
            {
               case Simulator.Signal_1:/*Signal t is present*/
                  break;/* Remain in the same state */
            }
         break;
   }
}
```

Note the following points in the generated code

- When the signal s is thrown it is not added to the process queue immediately but is stored in a temporary queue *Simulator.thrownSignalQ* [3] from where it is added to queues of the processes at the end of the scheduling cycle.
- The variable *Simulator.Transition* is set if a transition is possible from the current state with the current queue configuration.

As mentioned in Observation.3.1.1 , data structures for the queues of the processes have to be maintained in the code generated for the process description. The following two queues simulate the queue of a process[4]:

- *SavedSignalQ:* This queue stores the signal saved during simulation - while checking if from the present *state* a transition is possible or not, this queue is checked first .
- *NormalSignalQ:* This queue contains the signals thrown by different processes. While checking if a transition is possible or not from the present *state* , this queue is checked after *SavedSignalQ* has been checked.

In addition to the data structures for the processes queue, two matrices are maintained:

- *transitionMatrix:* $transitionMatrix[i][j] = true$ implies that at the i^{th} state transition is possible with the j^{th} signal.
- *storedSignalMatrix:* This stores information about which signals are saved at which state.

[3] *Simulator* is the class in which the code for the simulator is generated

[4] The process queue is different from a normal queue only in the case when a signal saved - in that case the signal retains it's position.

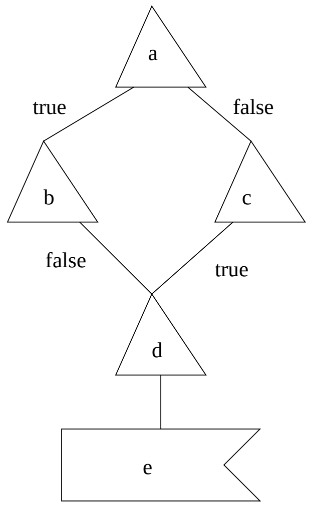

Figure 5.4: A *peek* structure

5.1.3 Handling of the Library construct

The discussion till now did not take into account the generation of code for the library construct §3.2.2. However, this turns out to be quite easy- the following is done to include the libraries in the system description during simulation :

- While building the *Dependence Tree* (see Fig.5.2) , the library component is instantiated (this instantiation may result in a tree structure). This instantiation is then attached to the proper point in the it Dependence Tree.
- All the fromal parameters are replaced by their actual parameters - this is done while the instantiation of the library construct is being done. While constructing the *Process Node* (see Fig.5.1) the *Text* field is scanned for any occurance of the formal parameters - if so they are replaced by their formal parameters (macro substitution).

5.1.4 Handling of the Peek Consttruct

The code generated for the sample process description (see Fig.5.3) did not have to generate code for a peek construct. However, it may seen that a peek construct will only result in a change in the values of the *transition Matrix* elemements §5.1.2.

The peek constructs which can occur only after a *state* construct form a structure which in general will be a Drected Acyclic Graph. The point to note is that a *peek* construct is similar to a *decision* box - so the expression at the leaves[5] will be evaluated as a similar *decision* structure would be . To be more precise, two nodes[6] will be *anded* along a path while two paths which join at a node are *ored*. Consider Fig.5.4 , the evaluated peek expression for signal e is $d \wedge ((a \wedge \neg b) \vee (\neg a \wedge c))$.

[5]Which are *Signal in* constructs.
[6]A node here is a *peek* construct

The algorithm for evaluation the peek structure is a variation of the breath first search algoritm.

Algorithm 5.1.1 *We define an array* Peek_Expression_Array *which stores the peek expression from the state construct to a particular node in the peek structure. Let i be the index to the node under consideration . Now,*

$$\text{Peek\_Expression\_Array[i]} = \bigvee_{j \in \phi} (\text{Peek\_Expression\_Array[i]} \wedge \mathcal{A}\text{Text[j]}^{7}) \tag{5.1}$$

where ϕ is the set of parents of node i and $\mathcal{A}$ is $\neg$ if i is a false *child of j .*

A node is not investigated till all it's parents are investigated[8]*.*

Finally, note that Peek_Expression_Array[s[9]] = true.

After this algorithm is run , *Peek_expression_Array[k]* contains the required peek expression for each leaf[10].

5.2 The Debugger

The debugger is very similar to the simulator - in fact the extraction step §5.1.1 is the same. However the debugger differs from the simulator in the following two points:

- When the code is generated for the processes, the *singleStepExecution* is changed such that now the transition is between *constructs* while in the simulator it was between *states*. Here also there is a module for a 'simulator' is there which does *singleStepExecution* of each process till a new state is reached or a transition with the current queue configuration is not possible.
- The output of the debugger should be such that it is shown on the canvas [11]. Hence an object of the GUI has to be passed to the debugger (note that this is not necessary in the case of simulation where only the output of the *print* statements have to be shown) - hence in the debugger the classes for the processes have to be generated dynamically i.e. code is generated dynamically (this is also done in the case of the simulator) and the corresponding classes are loaded dynamically.

[7] Text[i] contains the condition in node i

[8] In breath-first order.

[9] s is the *state* construct which is the root of the Peek construct

[10] Which is a *signal in* construct.

[11] See §3.1.1.

Chapter 6

Weak Reachability Analysis

6.1 Introduction

The Reachability problem for SDL is undecidable [4]. Hence, it is clear that in order to validate a general SDL design, we shall have to work in a paradigm which approximates the reachability relation in SDL. In this chapter we define a version of the reachability problem as **MultiSet SDL Reachability** and analyse its complexity, finally presenting an algorithm for a **approximate version** of the MultiSet SDL Reachability problem.

6.2 Intuitive Idea

The intuitive idea behind the concept of multiset reachability is that we have replaced the idea of a signal-queue of a process with that of a multi-set of signals. That is, each process now has a count of the different types of signals available. The motivaton for this can be justified if we assume a model for the SDL systems in which the signal routes between processes have certain delays associated with them, and the delays of two signal routes are not necessarily the same. This arbitrary delays associated with the signal routes then results in the signals arriving to a process in a different order than in the order in which they were thrown. The First Thrown First Served strategy of the tool implemented is just a specific case of this model when the routes have all same (or zero delays). As before, a simulation of an SDL system would now proceed by making transitions from states. We refer to the possible current states of a process during simulation as its **frontier states**. A transition from a frontier state may now take place on any of the available signals, which is consumed by decrementing the count of the corresponding type from the multiset of signals. Since the choice of signal is nondeterministic now, a process may have several possible transitions from a frontier state. When a process throws a signal A, say, the signal is added to the multisets of each of the processes for which it is defined. This is simply done by incrementing the count of signal A for the processes.

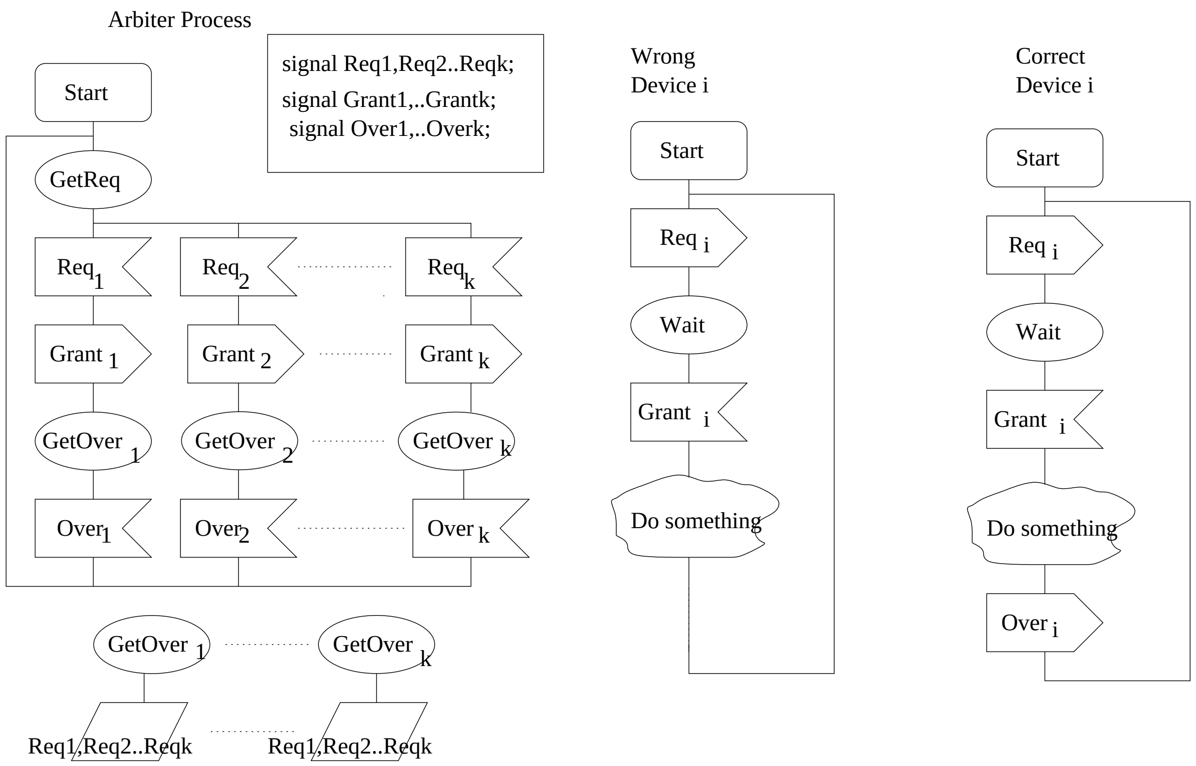

Figure 6.1: An arbiter with a wrong and a correct device

Example 6.2.1 *In the arbiter system shown in Figure 6.1, if we couple the arbiter with the correct devices, all the states will be reachable and all the transitions will occur. But if there is a mistake in the i^{th} device which, say, does not throw an $Over_i$ signal after its job is over, the transition from the state $GetOver_i$ to the state GetReq does not occur in the arbiter. This may easily be captured by our algorithm which proceeds by keeping a count of the various signals available to the processes at each step. The next example takes a more complex system and illustrates how our algorithm proceeds to detect unreachable states and transitions.*

Example 6.2.2 *In Figure 6.2 the available signals to the processes are A, B, C, D, E and F. If we simulate the SDL system of Figure 6.2 under the normal (Virtual Silicon) model of simulation, $state_6$ of $Process_2$ and $state_{10}$ of $Process_1$ are unreachable. $state_6$ remains unreachable even if we relax the model of simulation and move away from the First Thrown First Served model of simulation to a simulation keeping the counts of the signals only, i.e.*

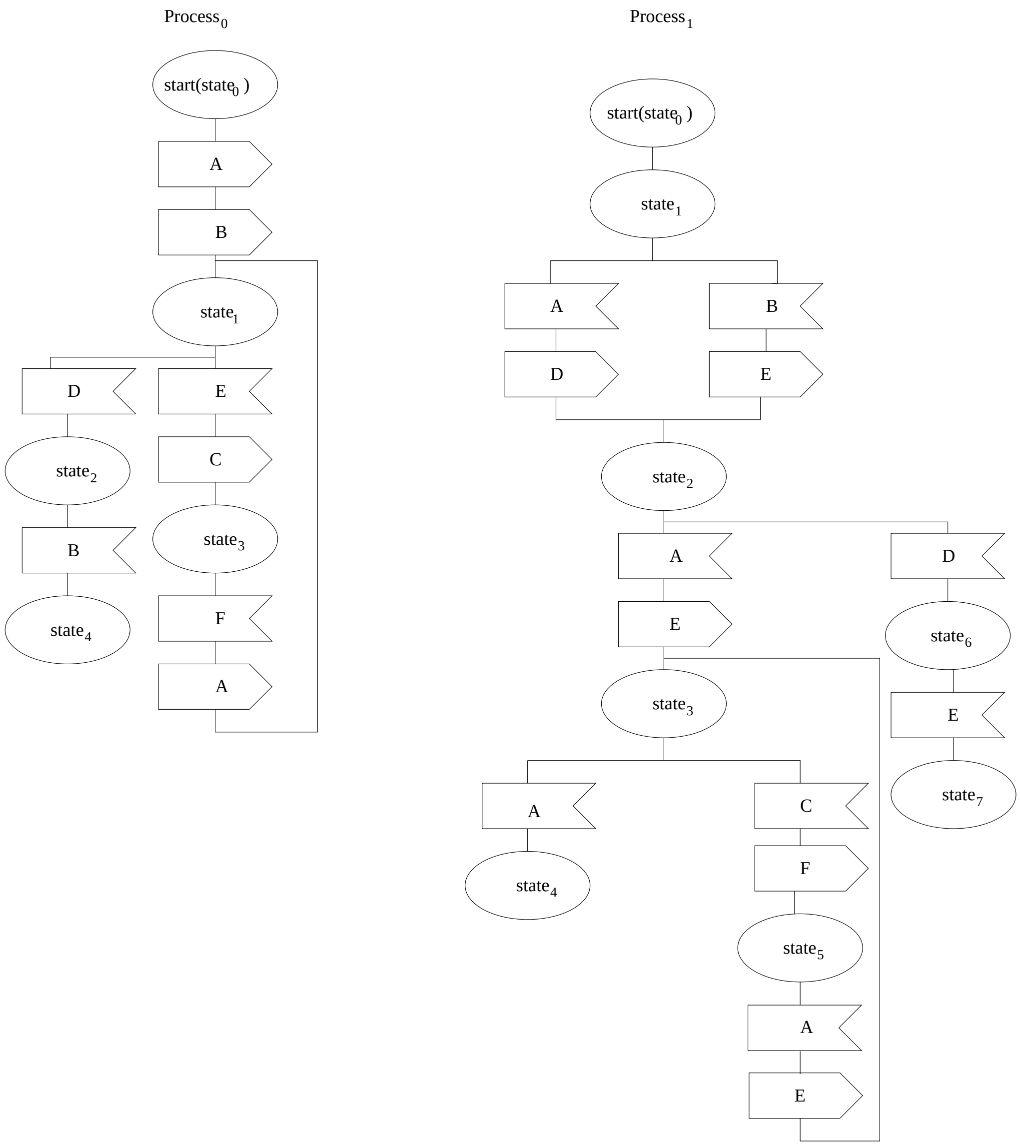

Figure 6.2: An overall example

Multiset simulation. *Suppose we denote the counts of the signal κ as $\kappa^{count(\kappa)}$ where $count(\kappa)$ denotes the count of the signal. A state and its signal counts will be written as $(s, \kappa^{count(\kappa)})$. (s, ϕ) denotes no signals present at state s. A stepwise simulation by our algorithm, the trace of the frontier states will proceed as follows. The first tuple in each step below denotes the frontier states of $Process_1$ and the second tuple the frontier states of $Process_2$.*

1. *Process 1:* $\{\ (0, \phi)\ \}$, *Process 2:* $\{(0, \phi)\}$
2. $\{(1, A^1B^1)\}$, $\{(1, A^1B^1)\ \}$. *This means that after the first step, $Process_1$ is in $state_1$ with 1 copy of signal A and 1 copy of signal B. Similarly for $Process_2$.*
3. $\{(1, D^1E^1)\}$, $\{(2, A^1B^1D^1E^1)\}$
4. $\{(3, E^1D^1C^1),\ (2, E^2)\}$, $\{(3, C^1E^2D^1B^1A^1),\ (6, E^1B^1A^1C^1)\}$
5. $\{(2, F^1),\ (3, F^1)\}$, $\{(5, E^2D^1F^1B^1)$,$(7, B^1A^1C^1)\}$
6. $\{(1, A^1),\ (2, \phi)\}$, $\{(5, A^1)$,$(7, A^1)\}$
7. $\{(1, E^1),\ (2, E^1)\}$, $\{(3, E^1)i,\ (7, \phi)\}$
8. $\{(3, C^1),\ (2, \phi)\}$, $\{(3, C^1),\ (7, C^1)\}$
9. $\{(3, F^1),\ (2, F^1)\}$, $\{(5, F^1),\ (7, \phi)\}$
10. $\{(1, A^1),\ (2, \phi)\}$, $\{(5, A^1),\ (7, A^1)\}$
11. $\{(1, E^1),\ (2, E^1)\}$, $\{(3, E^1),\ (7, \phi)\}$
12. $\{(3, C^1),\ (2, \phi)\}$, $\{(3, C^1),\ (7, C^1)\}$
13. $\{(3, F^1),\ (2, F^1)\}$, $\{(5, F^1),\ (7, \phi)\}$
14. $\{(1, A^1),\ (2, \phi)\}$, $\{(5, A^1),\ (7, C^1)\}$

Given this multi-set paradigm for SDL process simulation, the reachability set under this is defined as usual to be the set of all states that are reachable under any simulation. This set, which we shall denote by **MultiSet SDL Reachability Set** will be a **superset** of the reachability set (**queue SDL Reachability Set**) under the queue paradigm of SDL simulation. We state an important result without any formal proof. The intuitive idea behind the claim is clear enough.

Claim 6.2.1 *The MultiSet SDL Reachability Set for any SDL process will be a superset of the queue SDL Reachability Set. Therefore, when we detect a state to be unreachable under the multi-set paradigm, the state is also unreachable under the queue paradigm. However as shown in Figure 6.3 some states unreachable in queue simulation may be reachable under a MultiSet simulation. The* $state_3$ *of* $Process_1$ *is reachable under the MultiSet simulation but not under the queue simulation.*

Thus our aim will be to find out the a set of states which are unreachable. We show that even the MultiSet SDL Reachability is a very hard problem to compute and thereby develop algorithms for computing a weaker versions of it, which we together name as **Weak Reachability Methods**. Primarily we develop one algorithm for computation of an approximation to the **MultiSet reachability set** based on certain parameters. However, all states that are unreachable in the queue-simulation or even in the MultiSet simulation cannot be detected by our algorithm.

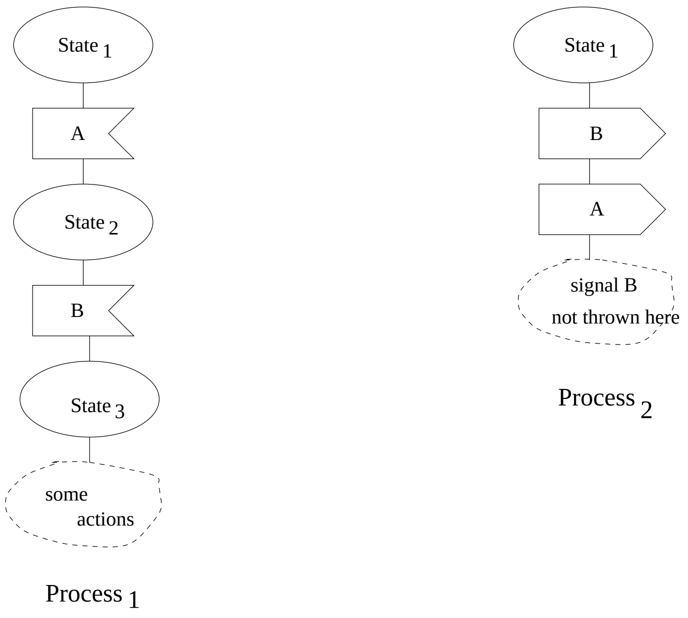

Figure 6.3: Multiset simulation differing from queue simulation

6.3 Complexity, Approximations and Algorithm

6.3.1 Complexity of the problem

The reachability problem, under the multi set paradigm, can be stated succintly as
Multi Set SDL Reachability Given that

a) The available SDL constructs are those of **state, signal-input,** and **signal-throw** and
b) The processes maintain a multiset of signals instead of a queue and the choice of signals for transition is nondeterministic,

the problem is to find out whether a given SDL state is reachable from the Start state.

Under a suitable formalization of the multiset paradigm, we can show that the **MultiSet SDL Reachability** problem is polynomially equivalent to the **General Petri Net Reachability** problem. That is, any algorithm which solves one of them in polynomial time should be able to solve the other within a polynomial blowup of the time required. Thus, an algorithm for the **MultiSet SDL Reachability** can easily be found by applying the reduction first and then an algorithm for the **General Petri Net Reachability**. The detailed algorithm can be easily worked out from the formalization of MultiSet SDL. Now, it is known from [] that **General Petri Net Reachability** is an extremely intractable problem. So, in order to cut down on the complexity and make the algorithm efficient, we have to take resort to some approximations. We present the approximations taken in conjunction with the algorithm itself.

6.3.2 Outline of Algorithm

In this section we first discuss the major data structures that we are to use. Then we describe an outline of the algorithm, pointing out the places where we have applied our approximations. The approximations themselves are discussed in the next subsection.

The first approximation that is imposed is placing a upper ceiling on the count of signals of any particular type available to a process. How this upper ceiling may be calculated is discussed in the approximation schema. We denote the ceiling corresponding to signal sig as $\aleph_{sig}$. Incrementing or decrementing $\aleph_{sig}$ results in $\aleph_{sig}$ only. We could also consider these values $\aleph_{sig}$ as parameters of the algorithm.

The major data structures that are to be used in the algorithm are:

- A (**multi**)set of **frontier states** for each process. These states represent the possible states that the process could be in at the present instant. Right now, we do not put any other restriction on the frontier states and the same state may be present more than once as a frontier state, with different **signal vectors**. The required approximation for the efficient algorithm is discussed in the next section.

- A **signal vector** associated with every frontier state. The signal vector has n positions for n signals and the i^{th} position gives the count of signal i, i.e. the number of copies of signal i present in that state. A signal vector v_1 is said to dominate another vector v_2 if for all the n positions, v_1 has a count which is at least equal to the corresponding position of v_2. Comparison between two signal vectors whose types are not the same is not allowed. Henceforth we shall refer to a any multiset of signals as a signal-vector and use these two terms interchangeably.

- By a **global state** we shall mean the tuple of frontier states for all the processes along with their signal vectors.

  Dominance: One global state g_1 is said to dominate another global state g_2 if the set of frontier states of g_1 is a subset of frontier states of g_2 and the corresponding signal-vectors of g_1 dominate those of g_2.

The algorithm proceeds by means of a mock parallel simulation. Each process maintains a list of frontier states. Proceeding one step for a process consists of proceeding one step along each of the frontiers.

1. The processes are handled in order of any fair schedule (e.g. round robin scheduling).
2. When a particular process is given a chance, update the following
   (a) All the signal-vectors of the frontier states.
   (b) Proceed one step along all the frontier states and update the set of frontier states. If transition is not possible along any frontier state s, because of the required signal not being present, reset all the signal counts, except those saved at that state, to zero. This step is in accordance with the queue semantics of SDL. To prevent an explosive growth in the set size at this point, apply the approximation schema.
   (c) Update the signal vector of each new frontier by the signals thrown during the transition from its parent to itself. Updation of the multisets of other processes is a more intricate issue that will possibly lead to an exponential growth in the problem. We discuss this updation under the approximation issue.
3. Periodically, check with the halting conditions to see if there is any need to continue.

6.3.3 Approximations to MultiSet Reachability

The approximations that have been employed and referred to while discussing the algorithms are as follows

- **Resetting the signal counts:** When a frontier state cannot perform a transition, because of the required signals not being present, we reset the counts of all the signals, at that state to zero, except for the signals which have been saved at that point. This step captures some of the ordering that is present among the signals in the system in the queue model of simulation.

Example 6.3.1 *In Figure 6.2 the* $state_4$ *of* $Process_1$ *is unreachable under the queue semantics, because signal D is available only after* $Process_2$ *consumes signal B in step 1. So, when signal D is consumed in* $state_1$ *of* $Process_1$*, signal B is thrown out from the queue. This is not captuerd in Multiset simulation. But in our algorithm, in step 2 as shown in Example 4.2.1, since signals D or E is not present for transition, all the signal counts are reset, and the* $state_4$ *becomes unreachable.*

- **Imposing a Bound:** We have imposed an upper bound on the number of signals of each type. A signal of type *sig* is assigned an upper bound $\aleph_{sig}$ as stated before. Before the actual algorithm is started, we have to decide the bound that we want to impose on the signal count. This can be done either by interaction with the user or by creating a **requirement matrix**. The position (s, t) of the matrix stores, as a multiset/vector the maximum number of signals that may be required to reach from state s to state t. From the requirement matrix calculate a upper ceiling for each signal beyond which the count is not needed to be incremented.

  Example 6.3.2 *In the Figure 6.2, if the bound on the count of signal A is kept as* 1*, we shall obviously detect the state* $state_3$ *to be unreachable, but if the bound is* 2 *onwards,* $state_3$ *becomes reachable.*

- **Controlling frontier states:** To control the state explosion problem, we impose a restriction on the **frontier states** of a process. If the same state s is present, say, k times in the list of frontier states with the associated signal vectors as $v_1, v_2, \ldots, v_k$ then we replace the k instances by a single instance of state s with the associated signal vector as $max\{v_1, \ldots, v_k\}$.

  Example 6.3.3 *When the above approximation is applied to the simulation of the SDL system shown in Figure 6.4, the state* $state_2$ *of* $Process_1$ *appears twice in the frontier states and is replaced by a single instance of it. As a result, the algorithm decides that the state* $state_2$ *has copies of all the signals and hence states* $state_7, state_8, state_9$ *and* $state_{10}$ *are reachable, when in reality neither of them are.*

- **Updation of signal counts:** If a process has frontier states $s_1, \ldots, s_m$ which perform transitions to move to states $s'_1, \ldots, s'_m$, throwing the multi-sets of signals $v_1, \ldots, v_m$ respectively in corresponding transitions, then since the process should actually be in only one state, we do a pessimistic approximation and take the multiset $max\{v_1, \ldots, v_m\}$ to be the multiset/signal vector by which we update the counts of the signals in all the other processes.

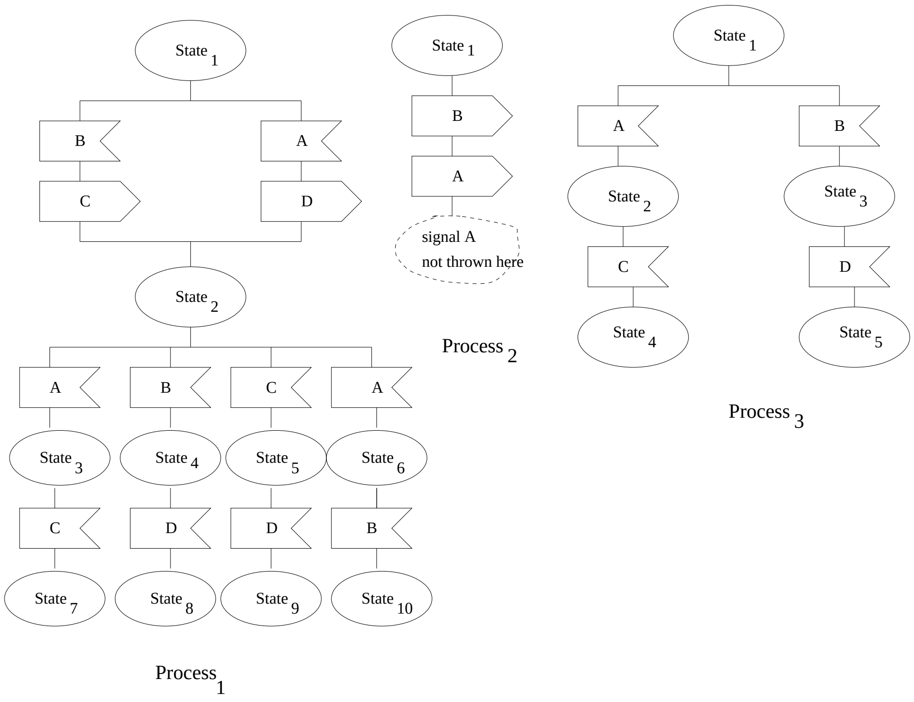

Figure 6.4: Error introduced by the "Updation" approximation

Example 6.3.4 *In the example system shown in Figure 6.4, the states* $state_4$ *and* $state_5$ *of* $Process_3$ *is not reachable under MultiSet simulation. But, because of the "updation" approximation, when the signal counts of* $Process_3$ *are updated in states* $1, 2$ *and* 3*, all the required signals become avalable to it, and states* $4, 5$ *are judged to be reachable.*

The claim that justifies our approximations can be stated, without proof, as

Claim 6.3.1 *If the bounds of all the signals (*$\aleph_{sig}$ *for signal sig) are kept sufficiently high, then any state reachable under the* **queue-SDL Reachability** *paradigm is reachable under the* **Weak Reachability** *paradigm. That is, states detected unreachable by the above algorithm will be unreachable in* **queue-SDL simulation***.*

6.3.4 Conditions for halting

The conditions for halting will be of two types.

- When the system is such that the number of signals goes on increasing, after a certain time the **signal vector** will exceed the **requirement vector** for a state. We can then immediately stop. However, since we are analysing all the paths, if $\aleph_{sig}$ is taken sufficiently high for all the signals, this will be equivalent to all states being reached, a trivial condition for reachability.

- The other case consists of the system coming back to a global state which is dominated by a past global state. That is, the system tends to go in a loop. For detection of such a loop we decide to remember a **checkpost global state**. A checkpost global state is any one chosen at random at first. As the simulation proceeds, we update the checkpost either at certain intervals, or if we find a global state which dominates our checkpost strictly. The simulation stops the moment we find return to a global state which is strictly dominated by the checkpost. The interval that is chosen for updation of the checkpost is increased as simulation proceeds, so that when the length of the interval crosses the length of the loop, we can detect the loop easily. This technique can be modified and the detection of the loop made more efficient by use of various policies of similar to *page-replacement*. We could consider keeping a reasonable sized window of checkpost global states which can be updated at intervals.

Example 6.3.5 *On examining the trace given in Example 6.2.2 of the system of Figure 6.2, we find that the global state in step 9 is dominated by the global state in step 5. The algorithm, may therefore terminate after step 8 and declare the result.*

6.4 Handling SDL-C

When we are dealing with the SDL-C systems, besides the states, signal-in's and the signal-throws, we also have the additional constructs of taskblocks, and decision blocks. So, in order to examine the **weak-reachability set** of an SDL-C system, we abstract out the control-structure of the system by neglecting the task blocks and the decision blocks. An example is given in the Figure 6.6.This means that a transition from a state can now lead to multiple states, but since we are in any case keeping a set of frontier states in our system, this does not matter. However, as we can see from the Figure 6.6 abstracting the control flow based on just the signals would not take us very far, in this particular example, when we neglect the payloads, we are losing the information of the requesting device. Thus, to capture any system meaningfully, we must refine our abstraction technique and our algorithm. One possible idea is to use bounded variables and do a **flow graph analysis** over the taskblocks and **opening up the signals with respect to the payloads**. This means that now the whole SDL-C process will be a structure formed of the signal control flows and the variable data flow graphs. Analysis of this structure should definitly give us a better idea of the reachability property in the SDL process.

Another important issue is that of timers. They are handled in the following mannner : whenever a timer is is come across in the control flow graph, along that path we make

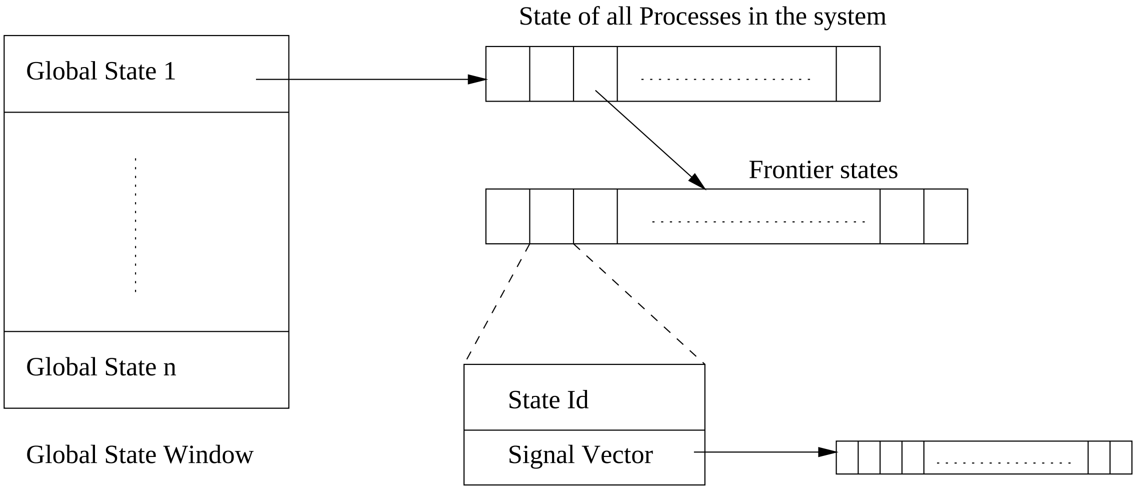

Figure 6.5: Data structure used in Weak Reachability Analysis

the signal count equal to zero. The logic behind this is that a time is increased only when in the last pass no transition could be made i.e. all the signal queues are empty. The point to note is that a timer can be fired only when the current time is increased.

The algorithm for weak reachability which has implemented in the design tool is given below.See figure 6.5 for the data srtucture used.

Algorithm 6.4.1 *The following steps are done:*

- *Extract the control flow graph from the system description. See fig.6.6 for an example.*
- *Perform mock parallel simulation on the abstracted control flow graph in the following manner.*
  - *Every process is given a chance according to some schedule. In the tool the order is generated randomly and round robin scheduling is done.*
  - *For each state in the frontier, explore all possible transitions. All transitions are fired for which the corresponding signal count is greater than zero. After the transition is made the signal count of the corresponding signal is decremented by one. For each signal that is thrown in a transition, the corresponding signal count is incremented by one. Note that for each transition that is being explored,*

separate temporary signal counts are maintained. The final set of new states along with their respective signal counts form the new frontier.

– *If no transition is possible from a state, then the signal counts are set to zero. This captures to some extent, the semantics of the FIFO signal queue.*

– *In each transition, the signals thrown are stored in a temporary count array. After one step has been executed the maximum of all the temporary thrown signal counts are taken and in the next cycle, the signal counts are appropriately updated.*

– *Each decision in the original process description is replaced by parallel paths for each decision path. Whenever such parallel paths are encountered, copies of the signal count are made for each parallel path and the mock simulation continues.*

– *If the same state occurs more than once in the frontier, they are collapsed into one by taking the maximum of their individual signal counts. This is done to prevent explosion in the frontier size.*

– *A maximum bound is imposed on each signal- whenever the count of any signal exceeds its maximum bound, it is frozen at that value, i.e. after it crosses the bound, it can neither be incremeneted or decremented.*

– *Halting conditions (§6.3.4) are checked for tramination. For checking purposes, a window of global states are maintained (See fig.6.5.*

6.5 Performance of Weak Reachability Analysis

Table 6.5 shows the performance of the Weak reachability Analysis on three examples. The third entry is the example of Fig.6.2 modified a bit- an extra process has been added which throws dummy signals.

	Number of Processes	Number of States	Number of Unreachable states Detected	Number of Unreachable states	Time taken (in ms)
Data Comm.	11	30	0	0	110
Arbiter with 10 devices	11	34	0	0	33
Fig.6.2 example	3	15	2	3	1177

System Specs for Performance Evaluation
System : Sun Enterprise 250 workstation. Operating System : SunOS 5.7 System RAM : 256 MB

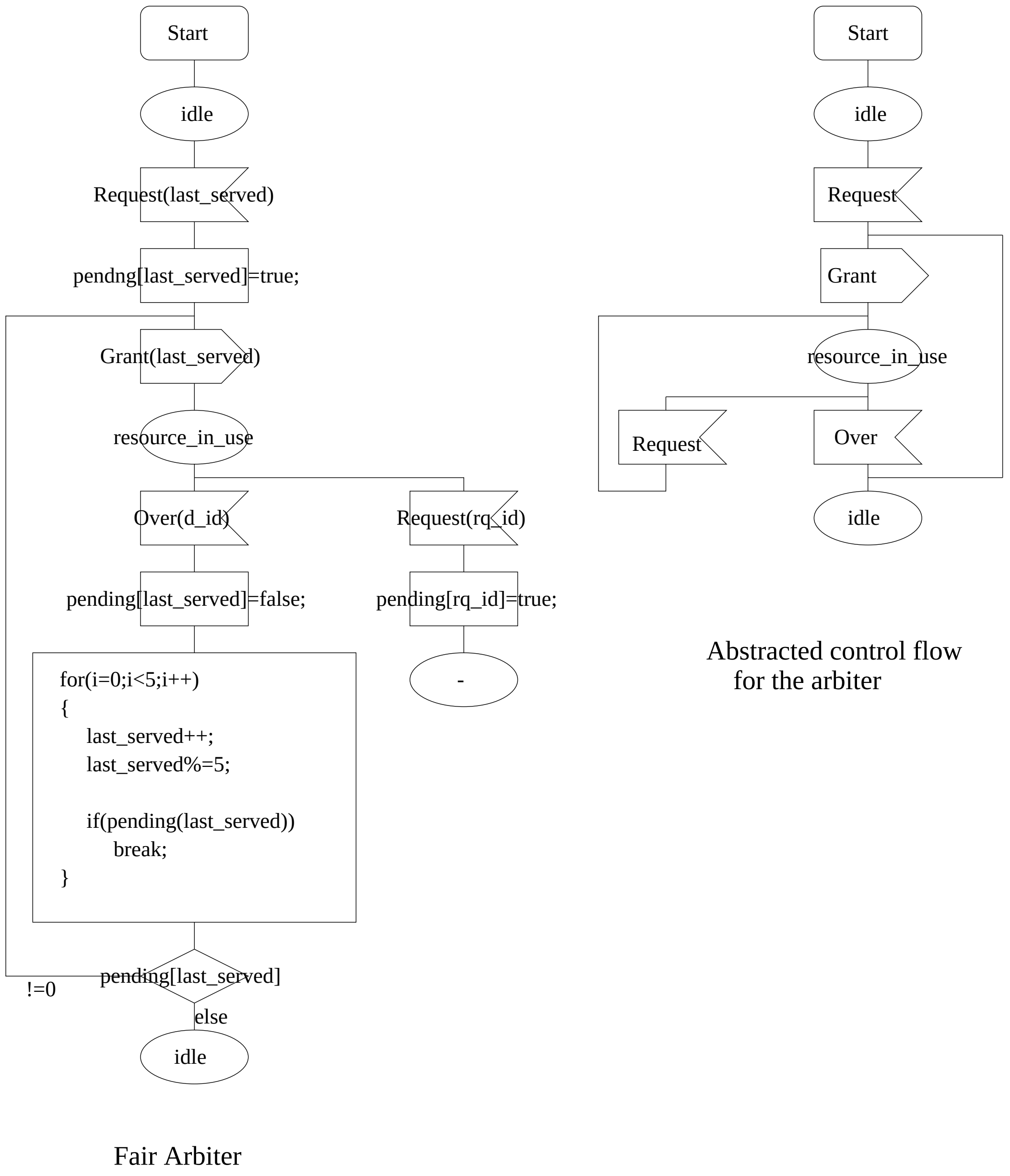

Figure 6.6: A Fair Arbiter and the control flow abstraction of it

Chapter 7

Dynamic Validation

7.1 Loop Detection

The first problem that we look at in this section is that of detection of a loop in the system. First we introduce the concept of a global state of an SDL-system. By a global state of an SDL-system we shall mean a tuple of

- The states of each SDL-process.
- The queue configurations, including the timer status, of each process.
- The value of each control variable, i.e. those variables which are present in decision blocks or affect (directly or transitively) the values of variables which are present in decision blocks.
- The timer list of the system.

A loop occurs if a global state recurrs. The motivation for finding out a loop is twofold:

- Any state that is unreachable till the step where a global state recurrs, will never be reached in future. To this effect, if all states have been reached then the loop checking is discountinued in **Noodles**.
- Once a loop occurs then the future output of the system can be generated without performing simulation any more. Nothing to this effect has been implemented in **Noodles**.

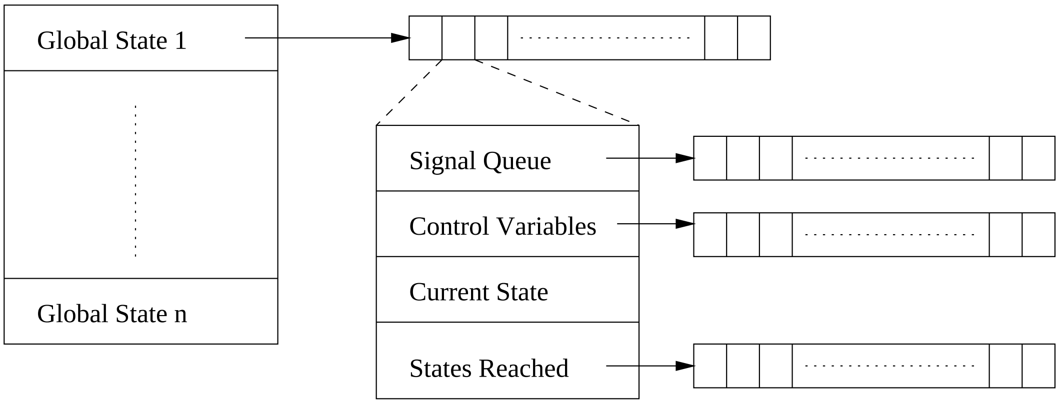

Figure 7.1: Data Structure used in Loop Detection

The algorithm for loop detection is simple. Maintain a window of global states. If at any instance the global state is same as that of any global state in the window, then a loop is detected. The data structure for maintaining the window is shown in fig.7.1. The following fields are maintained for each process-

- *Signal Queue:* This is the queue configuration of the process.
- *Control Variables:* This is the list of values of the control variables of the process.
- *Current State:* This is the state in which the process is in.
- *States Reached:* This is a boolean array which indicates if a particular state of the process has been reached in the current simulation.

The global state window is updated within pre-fixed time intervals.

7.2 An Adversarial Schedule

During normal simulation only one schedule is followed - though this may be alright for simulation purposes , it is insufficient for validation purposes as the property to be validated may hold in the present schedule but may not hold in another.

The main problem in choosing a general schedule for property verification is the large state space of all the possible schedules – factorial of the number of processes to be precise.

One way of tackling this is to heuristically choose an adversarial schedule i.e. a schedule which will force the simulation to take the system into a state where the property under consideration would not hold. Intuitively while choosing an adverserial schedule it would be better to choose one while does not have a fixed ordering among the processes during each step of the simulation i.e. the adverserial schedule chosen should be a combination of many orderings. In this section, we try to create an adversary schedule for analysing the reachability property of the SDL-states of the system. That is, at each stage we try to create a schedule where we can show the unreachability of as many SDL-states as possible.

Inherent with the idea of simulation of a SDL-system is the concept of a **scheduling cycle** of the processes where each process is given a chance once, and can perform at most one transition. But the order in which the processes are given a chance determines the order in which the signals appear in the queue of the processes and may possibly influence the reachability/unreachability of a process. While trying out alternative schedules, we decide to maintain this concept of a scheduling-cycle, and vary the orders of the processes inside the scheduling cycle.

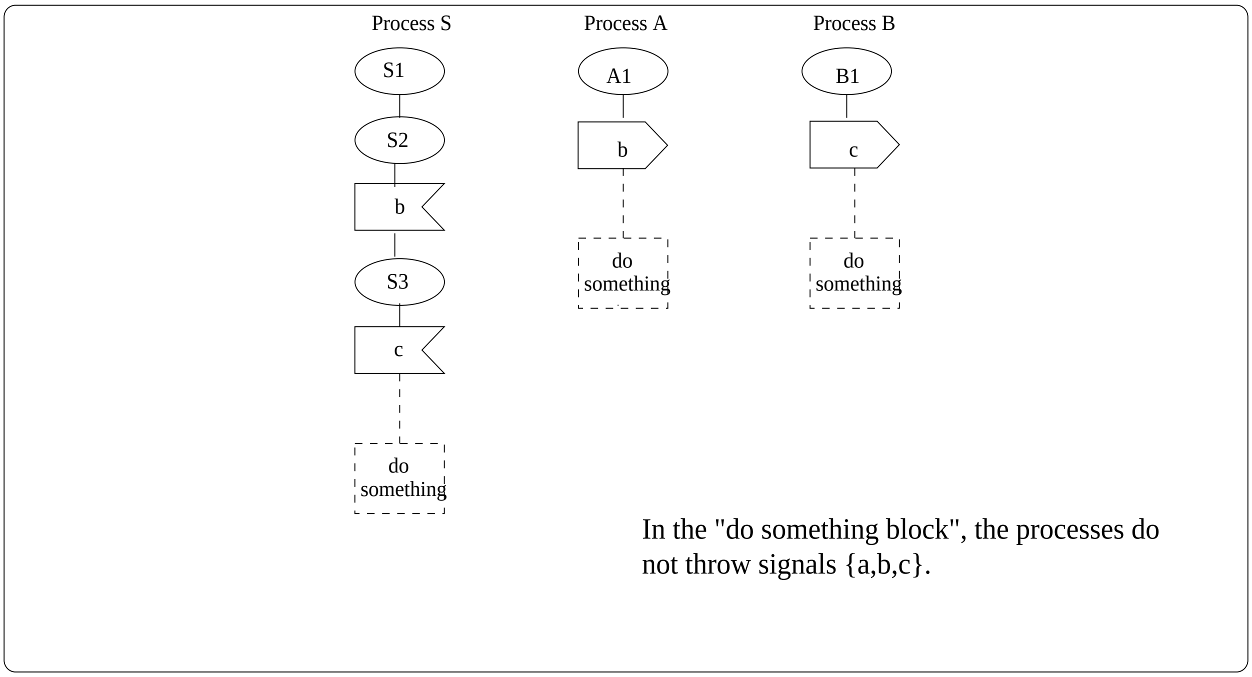

Figure 7.2: A possible deadlock

Example 7.2.1 *In the Figure 7.2 whether the state s of* p_3 *is reachable, depends on the ordering of the processes in the first scheduling cycle which ultimately affects the order in which the signals appear in the queue of each process.*

A simple heuristic for analysing reachability is described below. The major consideration in this heuristic is the assumption that **signals are useful to the system**. We thus intend to create a schedule where as many signals are thrown out (from the queue) as possible.

7.2.1 A Heuristic for choosing an Adverserial Schedule

One heuristic can be to randomly generate an ordering for each step of the simulation. The scheme given below improves upon this idea based on the assumption that *signals are useful in the designed system.* By one step of simlation we shall mean a complete scheduling-cycle. After performing each step of simulation, we try to find out the ordering of the processes for the next scheduling cycle. Based on the assumption of **signals being relevant to the process**, an *ordering between two processes* can be defined as follows : $pr_1 \leq pr_2$ if the following holds - in the next scheduling cycle where pr_1 is scheduled before pr_2 more number of signals are rejected in the simulation .

The heuristic, for each step of simulation generates an ordering among processes using the steps described below.

- Randomly generate a permutation of the processes
- Perform one pass of bubble sort on the permutation generated in the previous step. This is done in the following manner:
  - For $i = 1, 2, ..., n - 1$ where n is the number of processes in the system description.
  - Perform simulation for one simulation cycle on the ordering generated till this step. The $signal\_lost\_count_1$ is calculated as the sum of $signalLostCount(process_i)$ over all $i \in \{1, 2, ..., n\}$. $signalLostCount(process_i)$ can be calculated by looking into the queue confuguration and the current state of $process_i$.
  - Swap $process_i$ and $process_{i+1}$ in the ordering generated till this step. Calculate $signal\_lost\_count_2$ as $signal\_lost\_count_1$ was calculated in the last step.
  - If $lost\_signal\_count_1 > lost\_signal\_count_2$ then swap $process_i$ and $process_{i+1}$ in the ordering.

Observation 7.2.1 *As one pass of bubble sort is performed, this heuristic will be $\mathcal{O}(n)$ slower than the normal simulation, where n is the number of processes in the system.*

In the design tool, the adverserial schedule is generated in the code generated for simulation. Due to this, in the implementation of the above heuristic, the following point must be taken care of- before calculating the lost signal count, the state of the system must be saved. This implies all the variables and the signal queues of all the processes as well as the timer list must be saved.

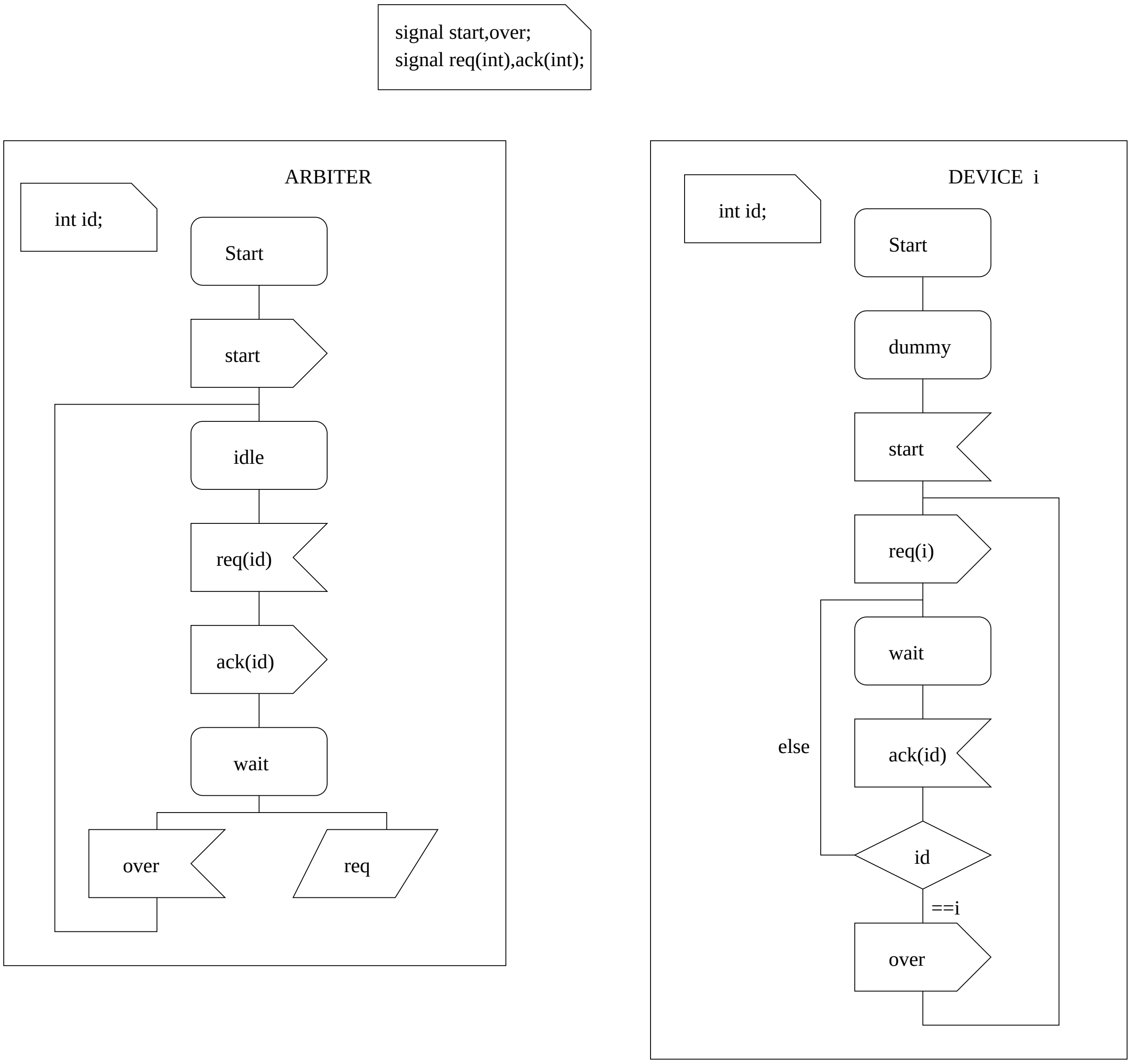

Figure 7.3: An arbiter example

7.2.2 Performance of the Adverserial Schedule

The arbiter system as shown in fig.7.3 was simulated in **Noodles** with and without adverserial scheduling.

Comparision of the times for simulation is shown in fig.7.4 where the simulation time is plotted against the number of devices in the arbiter system (starting from 2 to 10).

Figure 7.5 plots the ratio of the simulation time of adverserial scheduling to normal scheduling against the number of devices in the arbiter system (from 2 to 10). The straight line substantiates Observation 7.2.1.

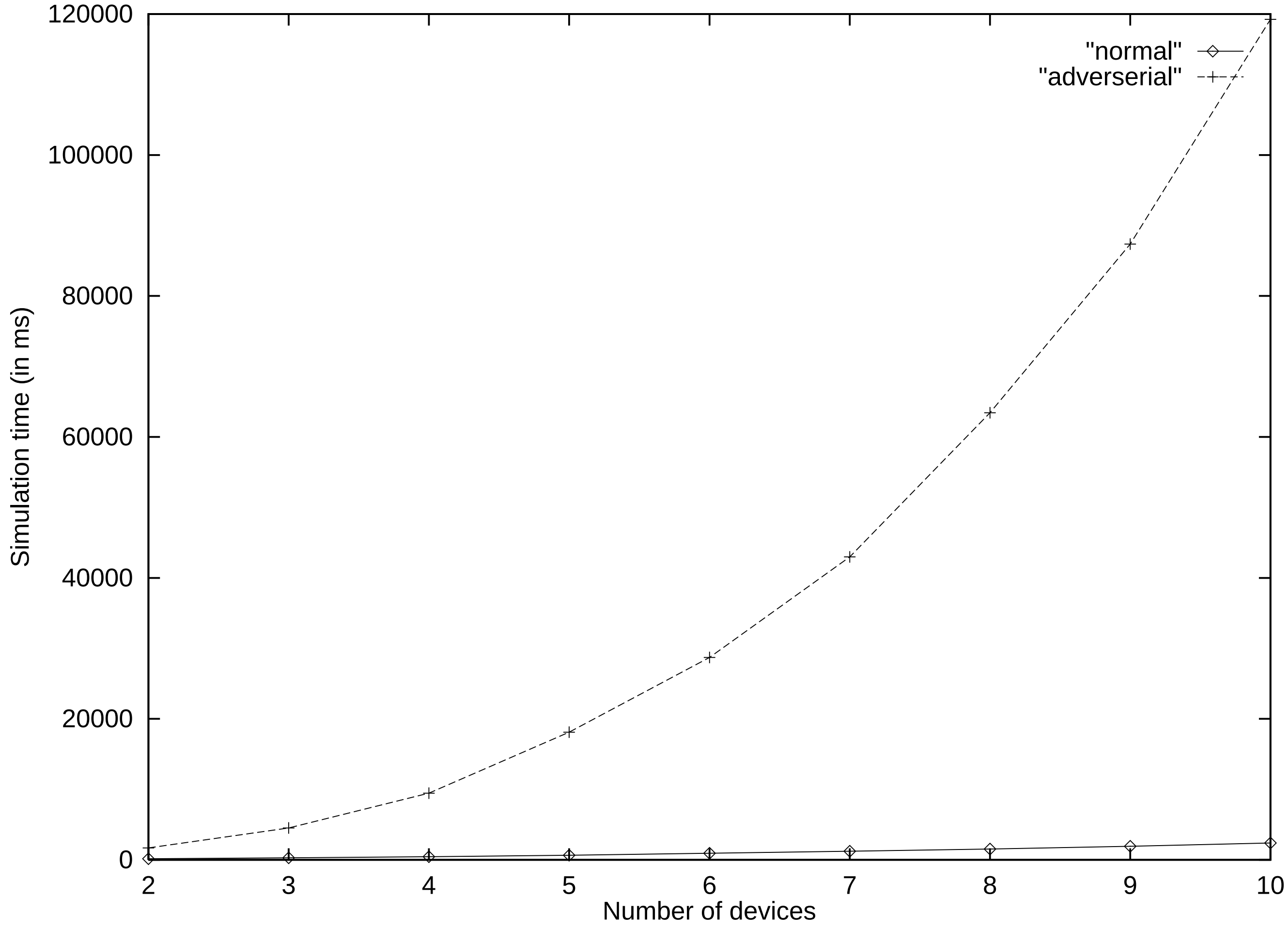

Figure 7.4: Plot of simulation times for normal and adverserial scheduling

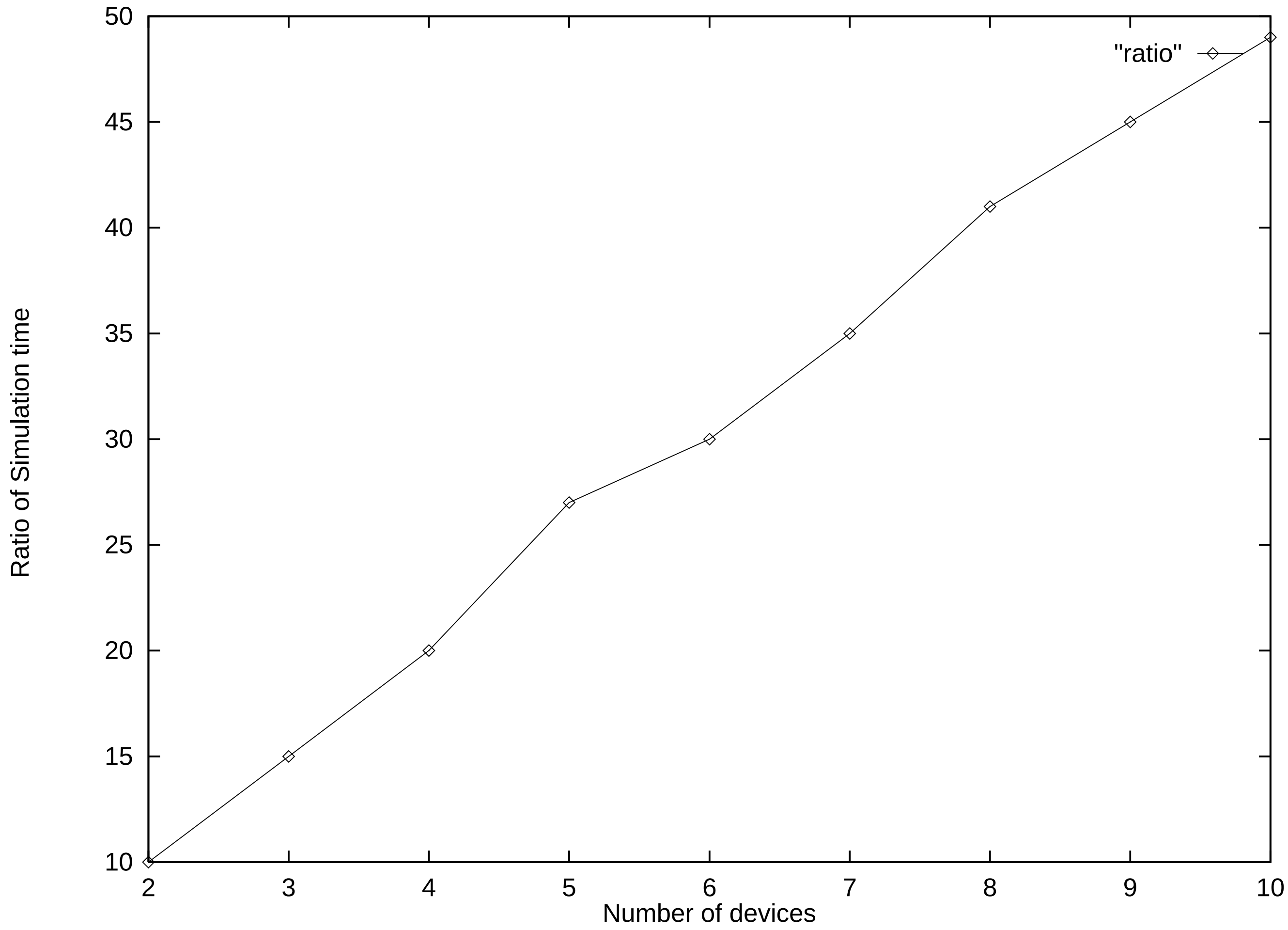

Figure 7.5: Plot of the ratio of simulation time of Adverserial Scheduling to Normal Scheduling

Chapter 8

Conclusions

8.1 Summary of Work Done

The following have been implemented in **Noodles**:

- *GUI :* The GUI has been implemented with the *Java.awt* package of **Java**. However, in the implementation, more efficient algorithms could have been used (For example, instead of the linear search on the process description graph which has been where ever a search is required, some efficient search algorithm could have been used).
- *Simulator:* The simulator is complete- during simulation code for each process is generated in **Java**. The other method of simulation using the process description graph (see §3.1.3) could also have been explored in more details.
- *Debugger:* The debugger has also been implemented. Here also debugging using the process description graph could have been explored.
- *Dynamic Validation:* Loop Detection and Adverserial Scheduling has been done. At present the control varaibles in Loop detection is taken from the user- this also be done by making the def-use graph of the variables to get the set of control variables.Also for loop detection, equivalence of global states is tested .However, instead of testing whether two queue configurations are same, prefix checking could have been done (i.e. if a later queue configuration is a prefix of a previous queue configuration, then also we can say that the two global states are equivalent, other things remaining the same).
- *Static Validation:* A static validation tool (see Chapter 6) has also been added to **Noodles**.

8.2 Scope for future work

Noodles can be augmented further along these lines:

- Assertion Checking: Another direction in which the dyanmic validation can be done is that of checking whether an assertion holds during simulation. An assertion is simply a constraint on the system which must be satisfied *universally* (that is in all possible system configurations) or *existentially* (that is atleast one configuration).
- Formal Verification: Formal verification techniques can be used for adding more verification tools to the existing version of **Noodles**.
- Hierarchical Deisgn: Though SDL provides for a hierarchichal design it is not really so as all the process description is at the leaves of the *dependence tree* (see fig.5.2). Provision may be made for process descriptions to be present at the internal nodes.
- Dynamic process generation: Another direction in which work can be done is that of processes being generated *dynamically*.The situation becomes more complex in this case- while presently all the processes defined are active at any time, here processes will be generated dynamically and the active set of processes will be subset of all the processes defined. Another point ot note here is that more than one 'copy' of a process which has been defined can be active at any particular time.To give a picture of what the design will look like, the new system design will have states which can generate processes- an *exit* construct will have to be added to the process description so that after the execution of the process is complete, control can be passed back to the calling state.

Appendix A

File Format

A.1 File Extensions

The following conventions for file extensions are followed in **Noodles** :

- .atr : These files store the information about the system - essentially it stores the *Hierarchies list* §4.3.2
- .prc : These files store information about a process description - it essentially stores the *Constructs* and *Connections* lists §4.3.2 .
- .blk : These file store information about a block description - it also stores the *Constructs* and *Connections* list : the latter is always empty.
- .lib : These files store the information about a library description.

A.2 File Format

Every entry in the files are terminated by newline. The specific formats are given below.

A.2.1 File Format for .prc & .blk files

The sequence of entries are

- Total number of constructs in the process/block description.

- List of records having the following format
  - Type of constructs
  - X-coordinate of the left top corner of the bounding rectangle
  - Y-coordinate of the left top corner of the bounding rectangle
  - Width of the bounding rectangle
  - Height of the bounding rectangle
  - Number of children
  - List of indices of children
  - Number of parents
  - List of indices of parents
  - Number of lines of text
  - List of text[1]
- Total number of connections
- List of records having the following format
  - Number of points in the polyline
  - List of X-coordinates of the points in the polyline
  - List of Y-coordinates of the points in the polyline
  - Index of the source construct
  - Index of the destination construct
  - Text on the connection

A.2.2 File format for .atr files

This file stores in formation about the hierarchy in the system design. The sequence of entries is :

- Target directory of the generated code
- Total number of entries
- List of records having the following format

[1]This may be necessary for there may be more than one line of text in a task block.

- File Name which stores the process/block description[2]
- Index to parent Hierarchy
- Number of children in the Hierarchy list
- List of indices of children Hierarchies.

A.2.3 File Format for .lib files

The sequence of entries are:

- Number of parameters
- Rest sequence is same as in §A.2.1 without any entry for the target directory

Note that in the library description , the parameters will be as { \$1,\$2,\$3 ... }

[2]in case of a library call the file name will contain the parameters also i.e. like lib1(p1,p2,p3).lib (parsing will have to be done to extract the parameters)

Bibliography

[1] E.M.Clarke and E.A. Emerson and A.P. Sistla. "Automatic verification of finite-state concurrent systems using temporal logic specifications". In *ACM Transactions on Programming Languages and Systems,* 8(2):244- 263, 1986.

[2] Computational Complexity, Christos Papadimitrou, *Addison-Wesley.*

[3] Daniel Brand and P. Zafiropulo. "On communicating finite state machines". *JACM,* 30(2):323-342, 1983.

[4] A. Dasgupta, A.C. Patthak, I. Bhattacharya and Dr. P. P. Chakraborti. "On the reachabilty problem in SDL"

[5] Handout on Virtual Silicon. *http://www.s-magic.com/doc/handout.pdf.*

[6] Ketil Stolen. "Development of SDL specifications in Focus". *Proceedings SDL Forum 95, 1995. http://www4.informatik.tu-muenchen.de/papers/Stolen_PSF91995.html.*

[7] K. Inan, V. Levin, A.Sen and H. Yenigun. "Partial order reduction of SDL verification". *Proceedings of FORTE 97 conference, Osaka, Japan.*

[8] Tutorial on SDL-88, *http://www.sdl-forum.org/sdl88tutorial/index.html.*

[9] Ursula Hinkel. "Verification of SDL specifications on the basis of stream semantics". *Proceedings of the 1st Workshop of the SDL Forum Society on SDL and MSC, 1998, Humboldt-Universitaet zu Berlin. http://www4.informatik.tu-muenchen.de/papers/SAMSDL98.html.*

[10] W. Peng and S. Purushothaman. "Data flow analysis of Communicating finite state machines". A preliminary version appeared in *ACM, PODC '89.*

[11] Documenation on Java, *http://java.sun.com/products/jdk/1.2/docs/api.*