CHAPTER 1

HIERARCHICAL ARCHITECTURE SIMULATION ENVIRONMENT

F.W. Howell and R.N. Ibbett

1.1. INTRODUCTION

The Hierarchical Architecture Simulation Environment (HASE) is a tool for modelling and simulating computer architectures. Using HASE, designers can create and explore architectural designs at different levels of abstraction through a graphical interface based on X-Windows/Motif and can view the results of the simulation through animation of the design drawings. This chapter describes the design and animation facilities of HASE, compares it with other simulation systems and concludes with suggestions for future tools based on several years' experience using HASE within the University of Edinburgh department of computer science.

1.1.1. The Motivation

Advanced simulation tools are available for low level electronic design, such as Spice for analogue circuits, and VLSI layout tools. However, tools for rapid prototyping of architectural ideas are less well established. Simulation languages can be used to model computer architectures, but the user has to be an expert on simulation. This is also the problem of general purpose simulation tools (e.g. SES/Workbench), where icons represent 'queues', 'servers' etc., and the link between a queueing model of an architecture and the architecture itself is not immediately apparent to the engineer not fluent in queueing theory.

Conventional languages (C, C++) are often used to construct simulators, but this approach involves starting from scratch for each new project. User interface aspects are often neglected as the tool will be thrown away with the next architecture. This is very wasteful, as many aspects of computers are constant between different architectures. The object oriented approach offers a solution. Standard components (such as memories, microprocessors and interconnection networks) can be held in a library. They can be constructed and linked together graphically on screen to create a simulation of an architecture, in much the same way that standard components can be wired together in a semi-custom VLSI tool. The difference is that the simulation is not fixed to low level wires and chip pins, but is free to choose the appropriate abstraction level.

HASE was designed to provide the flexibility of a raw programming language with the user interface advantages of a graphical tool.

1.2. DESIGN OF HASE

1.2.1. Overall operation

The HASE tool acts as a graphical front end to SIM++[1], a discrete event simulation extension of C++. SIM++ is used to describe the behaviour of basic components of a simulation. It provides a `sim_entity` class from which user components may be derived. Entities run in parallel and may schedule messages to other entities using SIM++ library functions. The user can link icons corresponding to entities together on screen and HASE produces the SIM++ initialisation code necessary for simulating the network. New components can be constructed by linking together standard components. Each component can be simulated at any level of abstraction. A register transfer level simulation will produce the most accurate simulation results; behavioural level simulations run more swiftly. The tool allows different parts of the simulation to run at different abstraction levels, so the user can 'zoom in' to specific parts of the design to simulate that at a low abstraction level and run the rest of the design at a high level of abstraction. Figure 1.1 shows how the parts of the system fit together.

1.2.2. Internal design of Hase

Each project built using HASE has its own directory for storing the SIM++ code. This directory may be used for building and running the simulation

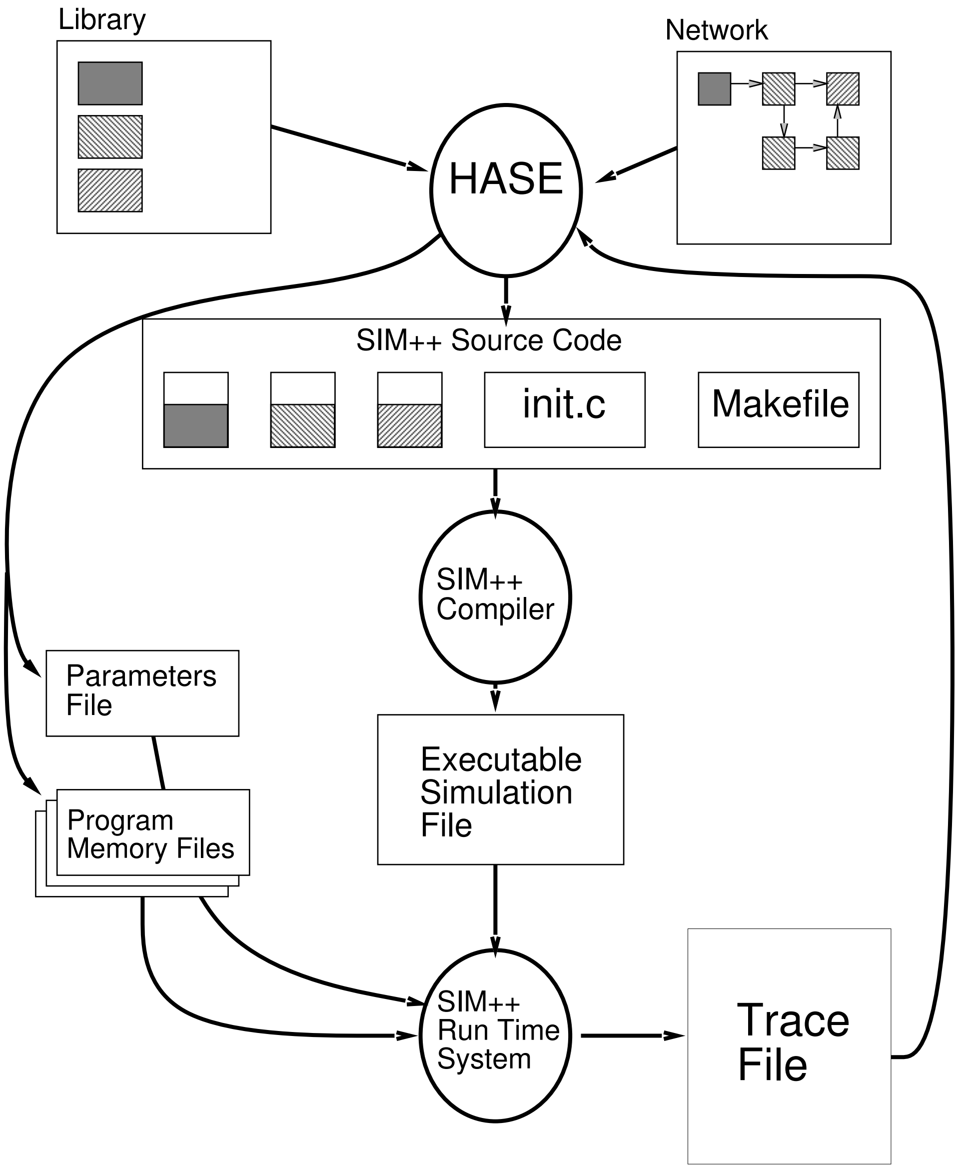

Figure 1.1. The top level design of HASE.

outwith the HASE environment using command line tools like `make`, giving the full flexibility of the SIM++ programming language. Alternatively the simulation process may be controlled from the HASE front end. HASE itself was written using C++, and a project is represented within HASE by four main classes; the **entity**, the **parameter**, the **link** and the **port**.

- **Entity.** This object stores a single component (or 'entity' in SIM++ terminology). The SIM++ code defining the behaviour is held in a file which has the same name as the entity. Within the object are stored details of the entity's ports and parameters. In addition, it holds the name of the bitmap file used for display and animation.

- **Parameter.** An entity may have many parameters. Details of these are stored within HASE along with instructions for their animation.

- **Port.** An entity sends messages to other entities via 'ports'. A port has a name, an icon and position relative to the entity's icon. The simulation code for an entity is written using sends and receives to and from these ports rather than directly to and from other entities. This constraint on SIM++ (which allows direct communication between any entities in the simulation) means that reusable components may be constructed with a defined interface.

- **Link.** This holds a link between two ports, drawn as a line on the screen. The object includes mechanisms for animating packets sent between entities.

1.2.3. Hierarchy

A subdivided entity may be defined in terms of a network of lower level components. Sometimes this is purely to make the design more manageable on screen, with the simulation still being performed using the low level components. It is also possible to provide simulation code for this higher level component and choose to use this one object rather than the low level network in order to obtain faster simulation time and less detailed results.

This choice of simulation level may be made at run time and is made by toggling a switch associated with the object. The external interface of the high level component is defined to be the same as that of the lower level network. This allows the simulation level of each object in the simulation to be set independently. Figure 1.2 illustrates two subdivided components connected by their external ports.

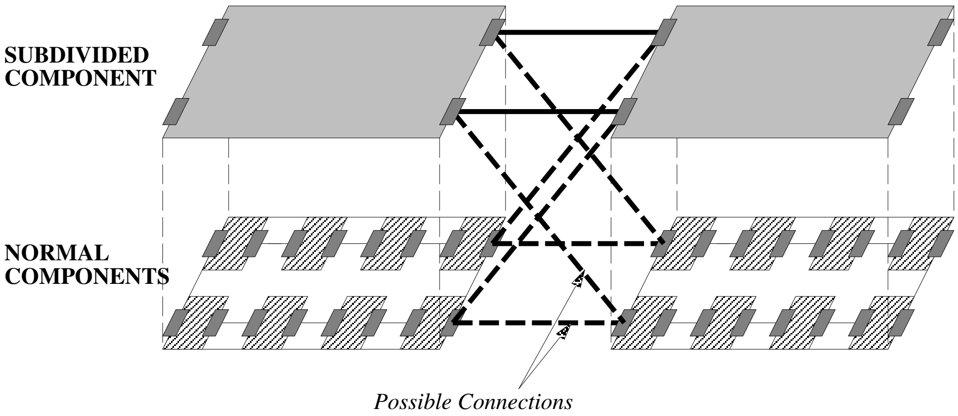

Figure 1.2. Two subdivided entities are connected by their external ports.

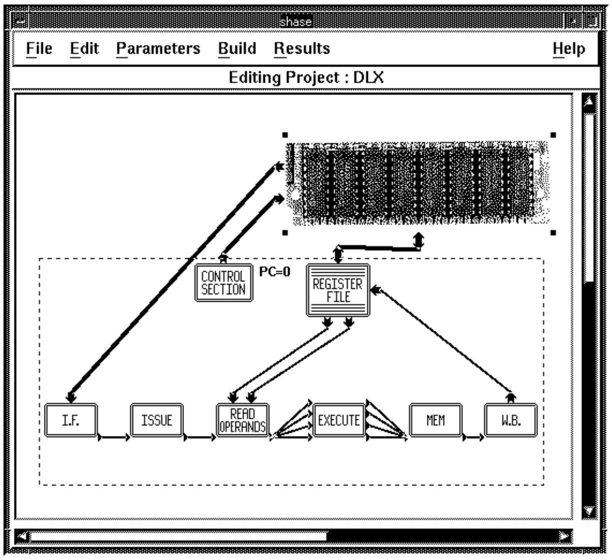

Figure 1.3. The HASE user interface.

1.2.4. Parameter Types

HASE parameters are the crucial link between the simulation code and the animation. They form the internal representation of each entity's state and include integers, floats, enums, structs and arrays. Once a parameter has been defined for an entity within HASE, that parameter is available to the simulation code as a normal C++ variable. The initial value of the parameter may be set using a Motif dialog and changes in the parameter's value may be recorded in the trace file at simulation run time, ready to be picked up by the animator (see section 1.4.1. for more details). Array variables are initialised at run time by reading in a text file. This process is powerful enough to allow streams of instructions (for example consisting of `COMPUTE <time>`, `SEND <proc#>`, `RECV <proc#>`) to be parsed and read in to a component's memory.

1.2.5. Templates

Templates for building common structures such as arrays and meshes of components are included. The user can slot any component into the template, set the dimensions and all the required components and links are produced. Current templates include a linear array, a 2D mesh, an omega network and a 3D torus.

1.2.6. Output Approaches

Simulations are renowned for producing vast quantities of raw data; transferring this into useful information is no trivial task. The result of a single simulation run is a trace file with timestamps showing when all changes in state and messages occurred. HASE includes two visualisation tools to make sense of this information; an animator (see section 1.4.) and a timing diagram display. The hierarchy is used to control the amount of information displayed on the timing diagram and logic-analyser style measurements can be taken.

Used in conjunction, these two tools show in detail what is actually going on during a simulation run, which is very useful when developing models. For very low level debugging purposes it is sometimes necessary to resort to looking at the trace file itself. Once a model has been developed, it is natural to stretch it with heavy workloads. This can rapidly generate unmanageably large trace files, so there is a mechanism in Hase for controlling how much trace information is produced (section 1.4.1.). For the largest runs it is usual to garner a small number of statistical measures from the model. These measures are taken using classes provided in SIM++ for histograms, counts and accumulated averages.

Repeated runs are required to investigate how a model behaves using a range of parameters[2]. These runs are typically controlled by a Perl script and graphs produced using the GNUplot program.

1.2.7. Recycling Simulation Objects

One of the major benefits ascribed to object oriented techniques is that software components may be reused by others instead of being recreated from scratch.

This ideal has nearly been attained by hardware simulation systems; hardware components have well defined inputs and outputs so designs may be constructed by gluing together off-the-shelf components. The ideal is only “nearly” attained in this case as effort is still required to package components for others to use, so a certain amount of reinvention still occurs.

The situation isn’t so rosy with object oriented software. This is partly because software is inherently more flexible than hardware. It becomes more difficult to define interfaces between objects when they aren’t constrained to N physical wires, but may instead be composed of data types, interdependent methods, global variables and so on. It requires a significant investment in time and effort to document and prepare objects so others may use them[3]. As a result, few objects are generally shared between people, and most people only reuse code they have written themselves.

It was an early design aim of the Hase system to encourage object re-use as much as possible. This has met with some success in practice (but not as much as was hoped for). The interface to most Hase objects is by typed messages to ports, which makes reuse of objects simpler than the general C++ case (but not quite as straightforward as low level hardware models). Objects which play by these rules may be included in a project with no problems. However Hase does not enforce this model; it is possible for objects to use SIM++ techniques to communicate using global variables or to bypass the ports. This makes it more complicated to simply slot such an object into a project. Practicalities such as proper documentation being provided for objects also affect reuse.

The Hase library system has been designed to address these issues. Rather than storing a set of *components*, it stores a set of *projects* each of which includes a list of components, the parameter and message type declarations and the global variables.

1.2.8. Object Oriented Databases

There has been substantial commercial and academic interest in object oriented databases recently. One common type of object oriented database is an extension to an object oriented language (such as C++) which provides for *persistence* of the objects. This approach is advertised as being suitable for storing the complex objects common in CAD systems, and providing desirable facilities such as version control and checkpointing of designs.

To investigate this approach to managing designs, Hase objects were made persistent by using the ObjectStore[4] database system. The experience was not without its problems. All HASE source files had to be preprocessed by the ObjectStore compiler before seeing the C++ compiler, which lengthened compile times. General run time performance became sluggish as all standard C++ pointers were replaced with persistent pointers, which could potentially result in a disk access. Any changes to class definitions made all previous database files unreadable (unless they were processed using a command line tool). Substantial source code modifications were required to be compatible with ObjectStore assumptions, and more modifications were later needed to obtain reasonable performance.

The conclusion from this experiment with object oriented databases is that the technology isn't yet mature enough for this type of CAD system. The general idea of allowing persistent objects within a language (without requiring I/O code) is a good one to be greeted with enthusiasm; in practice, however, adding an object oriented database requires much more effort than it would take just to write I/O code.

1.2.9. Limitations of graphical simulation systems

Die hard hackers sneer at graphical tools in general since they may never be as flexible as a programming language. This lack of flexibility is indeed a problem with *entirely* graphical tools which construct models at all levels by joining icons. At the lowest level of design, a description in a programming language is often best. However, there are also limitations with *entirely* textual descriptions; hardware and software designers usually use pictures to explain a system in terms of its subsystems. A compromise is therefore in order.

HASE is an inherently graphical system; if no pictures are needed, then there is little point in using it. However it does not impose a graphical approach to the specification of individual objects. These are described in SIM++ and the full power of SIM++ is available to the programmer.

This compromise is finely balanced and it typically changes during the life cycle of a simulation project. Initially when the design is fluid, animation and graphics are very important for communicating ideas between researchers. Later, when the design solidifies, the important aspect is simulation run times for collecting experimental data.

1.3. OTHER APPROACHES

1.3.1. VHDL

VHDL has become established as the standard hardware simulation language. It enjoys support from all major EDA companies and provides for simulation at levels from behavioural down to gate level. This section compares the VHDL approach with using a C++ based simulation language for simulating hardware systems.

1.3.1.1. Why use anything other than VHDL? High level simulations incorporating software are usually written in C or C++ since these are the languages used by programmers. It is possible to link code from different languages, but the process is never entirely painless as interface routines have to be written to convert between the different data formats. The ideal is to use one language throughout. McHenry[6] uses VHDL for high level system modelling, and Swamy[7] describes object-oriented extensions to VHDL to make it more suited to system modelling.

VHDL incorporates very powerful features for modelling hardware; there are explicit constructs for wires `signals` and detailed timing information may be included. It's possible to detect glitches and other low level hardware problems.

At the software level, good support is also included for concurrent processes; e.g.

```
architecture behavioural of component is
  signal w : bit := '0';
begin
  proc1: process is
  begin
    w <= 1;
```

```
      wait for 10 ns;
      w <= 0;
      wait for 10 ns;
    end;
    proc2: process is
    begin
      wait until w = '0';
    end;
  end behavioural;
```

Concurrent processes may be included *within* the description of a component. In SIM++, the unit of concurrency is the *entity* object. These entities communicate by sending and receiving *events*, which may contain data objects themselves. There is no concept of a *wire* as there is in VHDL, and no concept of a hierarchy of components (all entities are equal and may send messages to any other entity). The hierarchy is imposed on SIM++ by the Hase concept of ports. Programming in SIM++ is akin to programming a message passing parallel program.

The primary advantage of C++ based simulation languages (such as SIM++) over VHDL for system simulation is that linking to software libraries is significantly more straightforward. Basing communication upon messages passed between components rather than upon asserting signals allows a higher level view of the system, with the ability to send a data object at any abstraction level. VHDL on the other hand has much better tool support and standardisation than the various C++ simulation systems and includes direct support for modelling low level wire behaviour.

1.3.2. SIMULA / DEMOS

Another popular simulation approach is based on SIMULA and the discrete event package built on top of it (DEMOS). The original version of HASE was based on DEMOS[5]; the switch to SIM++ was motivated by the higher performance of C++ and the desire to interface to existing C and C++ libraries of code. Interaction between objects is based on shared *resources* which may have several operations defined, such as `wait`, `coopt` (a synchronisation).

1.3.3. Ptolemy

The Ptolemy project at Berkeley is a wide ranging simulation effort with a focus on signal processing[8]. It is a framework encompassing many different simulation styles, including a discrete event domain. The package includes support for animations written manually using the Tcl/Tk toolkit.

1.3.4. Commercial Tools

Several commercial tools are available for network modelling and general system simulation, an example being BoNeS[9]. These tools present a slicker and more complete interface than research prototypes like HASE, but as their source code isn't freely available they are less suited to playing with new ideas and adding new features.

1.4. ANIMATION

Watch the cogs and pistons of a steam engine for a while and you get a feel for the workings of the machine. This is trickier with electronic systems; although they are many times more complex than the steam engine, they just appear to sit and work their magic without effort (bar the odd flashing light and smoldering component).

An animation of a simulation model can generate a similar intuitive feel for how an electronic machine works. This often suggests 'obvious' improvements and highlights design flaws which may be concealed by a flat diagram or descriptive paragraph. It is also fun to watch a complex design coming alive on screen and behaving as intended (or, as is more likely, *not* behaving exactly as intended).

The main reason that animation isn't usually an integral part of the design process is the amount of effort involved in building one. The problem with creating an animation separate from the main design is that changes to the design have to be made to the animation code as well. This makes the animation diverge from the actual design and become unusable.

Hase addresses this by making animation an *integral* part of design. Simple animations are generated automatically, based on the state changes of components and the messages which are passed between them. More complex animations may be customised to include `GIF` colour icons.

1.4.1. The Approach

Animation is based on the changes in value of a component's parameters. These may be dragged onto the screen using the component editor (figure 1.4); once this has been done, any time that parameter's value changes it appears on the display.

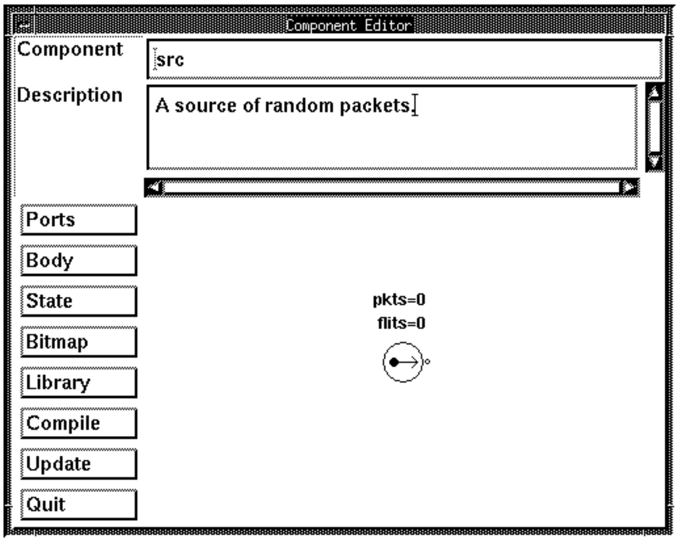

Figure 1.4. The component editor allows the state variables of an object to be dragged onto the display for animation.

The way a parameter is shown may be varied. `Value` just shows the value in screen (e.g. `123` for integers, `1.234` for floats, `BUSY` for enums). `Name+Value` shows the variable name as well (e.g. `curr_state = BUSY`).

Enumerated parameters may be displayed as icons instead of text; the icons are read in from bitmap files with the same name as the state (e.g. `BUSY.btm` or `IDLE.gif`). This is a simple but powerful technique for state animations; by simply providing the bitmaps for the corresponding states a customised animation is generated. These bitmaps may be displayed alongside the entity, or alternatively may be used to set the entity's bitmap.

`struct` parameters are displayed by drawing a box around the constituent elements (each of which may be displayed as above).

Thus far attention has been focussed on animating single parameters; any number of a component's parameters may be dragged onto the screen to be shown during animation, or they may be left hidden. It is also possible to define array parameters. The contents of these may be displayed on screen in a list box with a scroll bar and any updates or reads from the array are highlighted during the animation. Such updates are written to the trace using the `MEM_READ()` and `MEM_UPDATE()` macros in the SIM++ code. This technique has proved useful for displaying register contents and instruction buffers.

A simulation is not solely composed of state changes; there are also the messages sent between components. These messages may contain any form of data or handshake signal. The basic icon for a "message" may take any of the forms of static state parameters outlined above. This icon is animated by moving it down a link from one entity to another. The requisite line in the trace file is generated by the `send_DATA()` function in the SIM++ code, and the animation of the message is performed *at the time the message is sent*. Note that this is not necessarily the same as the time the message is acted upon by the receiving entity, as every SIM++ message is queued until the receiver is ready for it.

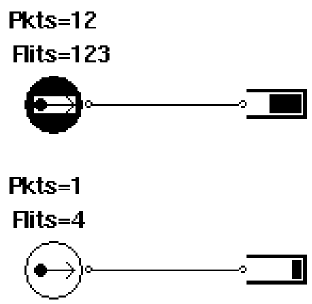

Figure 1.5. Changes in a component's state may be displayed on screen.

To show how the simulation code relates to the animation, figure 1.5 shows a `src` object connected to a `queue` and the following fragment shows part of the corresponding SIM++ code .

```
// excerpt from src.sim
  Pkts++;
  Flits++;
  if (ok_to_send)
    state=SRC_OK;
  else
    state=SRC_BLOCKED;
  dump_state();
  DataPkt d(123);
  sim_hold(1.234);
  send_DATAPKT(out,d,0.0);
  sim_wait(ev);
```

An example shows the format of the trace file which is generated on running the simulation and read in by the animator:-

```
// example trace file generated at run time
  u:src0   at 0.000: P SRC_BLOCKED 12 123
  u:queue0 at 0.000: P FULL_6
  u:src1   at 0.000: P SRC_OK 1 4
  u:queue1 at 0.000: P FULL_1
  u:src0   at 1.234: S out 123
```

Sometimes protocols require several messages to be exchanged between entities; in these cases it would be messy to animate all the acknowledge packets, so it is possible to send messages without generating any trace information. For large scale simulations, it is also often useful to avoid animating messages altogether and just show the state changes, so the "trace level" may be set to control which types of trace information are generated. The levels are:

comments and line numbers	1
message sends	2
memory updates	3
state changes	4
summary	5

Table 1.1. The levels of trace generation.

Setting the trace level to 4 (say) includes state updates and summary information in the trace, but not messages, memory updates or comments.

1.4.2. An example

Figure 1.6 shows an animation of a crossbar interconnection network with input and output queues. When the inputs block the icon is highlighted; it is possible to see the individual flits moving down the links and the queues grow and shrink dynamically.

1.5. APPLICATIONS

Architectural simulation work using the DEMOS prototype version of HASE is detailed in[5]. In 1992 work began on the current SIM++/Motif version which has been used in many MSc and final year honours projects, including simulation of multiprocessor WAN bridger/routers, simulation of the DLX processor and simulation of the DASH multiprocessor[10]. More details of

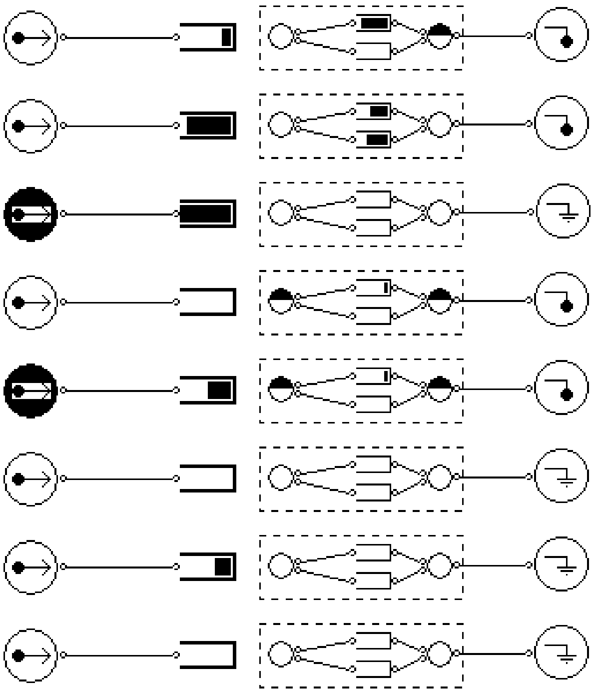

Figure 1.6. An interconnection network with input and output queues demonstrates the HASE animation facilities.

projects using Hase are given in[11]. Currently the main focus is on simulating multiprocessor interconnection networks and parallel MPI software. Many of the projects have involved linking simulation code to substantial existing libraries of C or C++ routines.

1.6. CONCLUSIONS

1.6.1. Important Messages

Animation has proved to be the most appealing feature of the Hase tool. The way in which it is incorporated into the design process allows swift construction of animation models and encourages communication and debate between designers. These advantages couldn't be obtained with an animation tool separated from the main design environment as there would be a problem maintaining consistency between the animation model and the one used for simulation.

The combination of an efficient threaded C++ with messages to communicate between objects is a powerful and intuitive programming model for software and hardware systems. It has also been useful that Hase imposes no restrictions on using SIM++ features.

The final message is that no simulation system will encompass all the needs of all projects. Many of the Hase features were included by students "extending" Hase to cope with the particular requirements of their project and this has proved the ultimate in flexibility, and a major advantage of having the source code and design available (which wouldn't be the case with commercial tools).

1.6.2. Future of the approach

New directions for the tool currently being investigated are closer tie-ins with an object oriented version of VHDL (to strengthen the links with hardware). VHDL itself is an attractive language for modelling hardware, but needs the addition of messages to model systems at a higher level. For software systems, it is very convenient to use a C/C++ like language since this makes it easy to include existing libraries of software.

Use of a parallel simulation language has been considered since the start of the Hase project and SIM++ originally had a timewarp version, but in projects to date the bottleneck hasn't been the simulation run time of individual runs, but rather the time to construct simulations. The lengthy simulations have been successive runs with different parameters which have been run simultaneously on different workstations. We are currently experimenting with our own implementation of SIM++ to run on the Cray T3D to map out the performance of a model over a large area of the input parameter space in parallel.

REFERENCES

1. JADE INC, Sim++ User Manual, (Jade Simulations International Corp., Calgary, Canada, 1992).
2. J. HILLSTON, A Tool To Enhance Model Exploitation, Technical Report CSR-20-92, Dept. of Computer Science, University of Edinburgh, 1992.
3. B. STROUSTRUP, The C++ Programming Language (Addison-Wesley, 1991), 382-384.
4. OBJECT DESIGN INC, ObjectStore Release 3.0 User Guide, (Object Design Incorporated, Burlington, MA, 1993).
5. A.R. ROBERTSON and R.N. IBBETT, "HASE: A Flexible High Performance Architecture Simulator", in Proc HICSS-27 (IEEE, Hawaii, 1994).
6. J.T. McHENRY and S.F.MIDKIFF, "VHDL Modeling for the Performance Evaluation of Multicomputer Networks", in Proc MASCOTS-94, (IEEE Computer Society Press, New York, 1994).
7. S. SWAMY, A. MOLIN and B. COVNOT, "OO-VHDL: Object-Oriented Extensions to VHDL", IEEE Computer, **28:10**, 18-26 (1995).
8. J. BUCK, S. HA, E.A. LEE and D.G. MESSERSCHMITT, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems", Int. J. Comp. Sim., **4**, 155-182 (1994).
9. S.J. SCHAFFER and W.W. LaRUE, "BONeS DESIGNER: A Graphical Environment for Discrete-Event Modelling and Simulation", in Proc MASCOTS-94, (IEEE Computer Society Press, New York, 1994).

10. L.M. WILLIAMS, Simulating DASH in HASE, (MSc Dissertation, Department of Computer Science, University of Edinburgh, 1995).

11. R.N. IBBETT, P.E. HEYWOOD and F.W. HOWELL, "HASE: A Flexible Toolset for Computer Architects", to appear in The Computer Journal, (1996).

Index

Animation, 11

BoNeS, 11

C++, 2
Crossbar network, 14

DEMOS, 10

Ptolemy, 10

SIM++, 13
SIMULA, 10

VHDL, 9