An Extensible Clock Mechanism for Computer Architecture Simulations

F. Mallet and S. Alam
Institute for Computing Systems Architecture *
Division of Informatics - University of Edinburgh
email: fmallet@dcs.ed.ac.uk

R.N. Ibbett *
Institute for System Level Integration,
Livingston, UK
email: rni@dcs.ed.ac.uk

ABSTRACT
Just as component re-use is becoming increasingly important in the design of systems-on-chip and parallel computing systems, so too is re-use of components in computer architecture simulation environments. Fundamental to both real and simulated systems is the clocking strategy. In this paper we present a behavioural design pattern to efficiently model the clock mechanism of hardware architectures. We illustrate how this pattern can increase reusability and reduce modifications when the synchronisation policy has to evolve or when components have to be included in larger architectures such as mesh networks.

KEY WORDS
Model development, object-oriented implementation, discrete-event simulation

1 Introduction

Just as component re-use is becoming increasingly important in the design of systems-on-chip and parallel computing systems, so too is re-use of components in computer architecture simulation environments. Many parallel architectures contain large numbers of processor cores integrated into powerful ASICs which are aggregated into meshes to obtain a total computation power of several TFlops. Such high-performance architectures require new design and evaluation tools offering abstraction and reusability, and providing good visualisation mechanisms to analyse aspects of system performance.

The UK Quantum Chromodynamics (UKQCD) collaboration involves the procurement of a high-performance system based on the Columbia QCDOC architecture [1], a system designed to simulate strong interactions between quarks and gluons. This provides a challenge in relating theoretical results about quark and gluon interactions and practical results on mesons and baryons, experimentally observed structures into which quarks and gluons seem to be confined. Individual PowerPC-based processing nodes are interconnected in a 6-dimension mesh with the topology of a torus. We intend to model and simulate such an architecture with HASE, a Hierarchical computer Architecture design and Simulation Environment [2].

HASE has already been used in several projects including the modelling of the Stanford DASH and the DLX architectures [3, 4], the Tomasulo algorithm used in teaching and in the investigation of parallel architectures to support the HPRAM model of computation [5]. These models use a simple two-phase clock implemented using a barrier synchronisation mechanism. Although HASE is an object oriented environment, the current clock mechanism is not well encapsulated and is difficult to extend to mesh architectures. Some new component-oriented and object-oriented inheritance techniques have therefore been added to HASE to satisfy the new requirements imposed by the QCDOC architecture.

Section 2 briefly presents the HASE environment and the Entity Description Language (EDL) used to describe entities, links and configurations. Section 3 presents the original clocking mechanism and its limitations when modelling mesh architectures. Then, we introduce the inheritance mechanism we have added and show how it improves synchronisation and clock mechanism modelling. We advocate that this new clock-pattern can be reused and enhanced to model several synchronisation mechanisms and give some examples. Section 4 briefly presents our mesh description mechanism and shows how it fully benefit from the new clock mechanism. Finally, we conclude with an enumeration of several aspects where models of parallel architectures may benefit from some basic object-oriented extensions.

2 HASE and EDL

HASE is a Hierarchical computer Architecture design and Simulation Environment which allows for the rapid development and exploration of computer architectures at multiple levels of abstraction, encompassing both hardware and software. Within HASE there are graphical entity design and edit facilities, entity library creation and retrieval mechanisms, an animator, and statistical analysis and experimentation tools for deriving system performance metrics.

The architecture is defined using the Entity Description Language (EDL), the screen layout is described by the Entity Layout (EL) description and the behaviour is described in Hase++, a C++-like programming language with primitives to manage discrete-event simulations. The EDL

file is composed of the preamble, which offers a short description of the project, the *parameter library*, which declares the different complex types and links used, the *entity library*, which describes the interface and hierarchy of entities to be used, and finally the *structure* part which defines the configuration, *i.e.* the instantiation of entities and their interconnection through links.

This is the first feature of object-orientation: classification. It allows the description of a set of components as a single entity which can be instantiated several times with different parameter values.

The second feature of object-orientation, which enhances encapsulation and thus reusability, is inheritance. When modelling a system, it often appears that many entities within a system have common properties. In such cases either composition or inheritance have proved to be helpful. Composition is more commonly used to delegate a part of the behaviour or secondary behaviour to another object. However, inheritance is preferred when the primary behaviour is to be identical to an existing object and just some aspects of its implementation are to be modified. In the latter case, the primary behaviour is not to be changed dynamically, although some particular aspects may.

We have therefore added an inheritance mechanism to EDL. In the *entity library* section, it is now possible to define an abstract entity, the primary behaviour of which is described in the corresponding Hase++ description. This primary behaviour may fulfil some particular aspects by calling services which are to be defined or overridden by concrete entities. These services are declared in the EDL and are similar to what are called *virtual methods* in C++ but are closer to *protected methods* in Java.

The following section presents a clock pattern which allows the modelling of several clock mechanisms used in hardware architectures, together with an example to show how this inheritance mechanism helps to implement this pattern. The mechanism to create mesh entities with an EDL description is separately described in section 4. Section 4 also shows that the mesh mechanism would not have been really useful without this new clock mechanism.

3 Abstract entities to model the clock mechanism

3.1 The original clock mechanism

The multi-threaded implementation of HASE means that the order in which entities complete their activities is non-deterministic, though clearly the order in which events are exchanged between entities must be deterministic. In principle a basic synchronisation policy could be built into HASE itself and an experimental version of HASE incorporating such a policy has actually been investigated to run on a Cray T3D. However, this has the effect of turning what is currently a general discrete-event simulator into a clock-driven simulator, where each entity is called once every simulated clock cycle. Although clock-driven simulators are more efficient for models in which every entity changes in every clock cycle, they are inconvenient for the general case and in synchronous architectures it is simple to construct a clock entity which implements a barrier synchronisation policy as part of the simulation model.

The original clock entity used in a number of HASE models contains, in each case, a list of the entities which are to receive clock synchronisation signals. These models operate with a two-phase clock: the first phase is an execution phase, the second is a communication phase. At the beginning of each phase, the clock entity sends a clock tick signal to each entity in the list. Receipt of this signal initiates the activity of the entity (thread). When an entity completes its current activity, it sends a signal back to the clock entity and then waits. When the clock entity has received completion signals from all entities, it generates a new clock signal.

The main restriction of this mechanism is the need to tailor the list of entities in the clock entity for each model. This is not a problem for architectures where the number of modelled entities does not change. However, it is a major issue for mesh architectures, where the size of the mesh and thus the number of entities in the model is one of the important simulation parameters. A built-in synchronisation policy would overcome this problem, but this would be contrary to the philosophy of HASE and so we need some other method of providing an extensible synchronisation mechanism.

A new clock mechanism has therefore been developed and this has provided some unexpected benefits. In particular, it is a more general synchronisation pattern with an explicit specification in the EDL file.

3.2 The clock pattern

Our main objective is to provide a clock mechanism which is reusable, as it must not depend on the modelled system. The proposed pattern consists of two classes, the *clock* class and the *clocked* class. (The term pattern here refers to the category of *behavioural design pattern* defined in [6].) The *clock* class represents the component which emits the *tick* signals and the *clocked* class which represents the basic behaviour of every synchronous component. These synchronous components all receive the tick signal and send back a status information signal (see Figure 1).

Patterns are generally used to reduce design, analysis and implementation time by providing ready-to-use generic schemes. The efficiency of such patterns is greatly increased when a code skeleton is automatically generated by an integrated development environment. Figure 1 uses a Unified Modelling Language (UML) class diagram extended with the Port and Protocol concepts of capsules defined in [7, 8]. The pattern described here is quite specific to the modelling of hardware systems.

The proposed behaviour of such a pattern consists of two parts. First, any *clocked-entity* may register with the

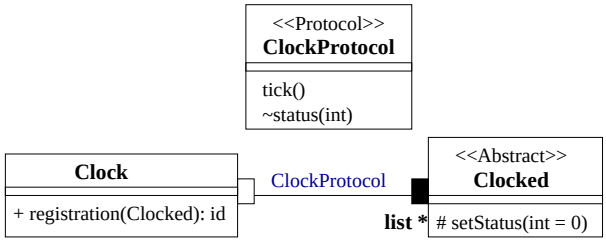

Figure 1. Clock pattern.

clock from which it wants to receive the tick signal. Second, the clock entity sends a tick to each of the registered *clocked-entity* and waits for the reception of every status signal before sending the next tick. Depending on the status of each *clocked-entity*, the clock entity may decide to terminate its activity (*e.g.* error detected in simulation or normal ending after decoding a Stop instruction).

The *clock* entity behaviour can be modelled with the following algorithm:

```
state = Normal
cycle_no = 0 ;

REPEAT

  FOR EACH clocked IN list DO
     SEND tick(cycle_no) TO clocked
  DONE ;

  FOR EACH clocked IN list DO
     RECEIVE status(val) ;
     IF (state == Normal
         AND val == Stop)
     THEN state = Stop ;

     IF (state != Error
         AND val == Error)
     THEN state = Error ;
  DONE

WHILE state == Normal ;

IF state == Stop THEN
  stop() (e.g. send a last tick to
              empty the pipeline)
ELSIF state == Error THEN
  print error information
```

The *SEND* and the *RECEIVE* commands are not necessarily synchronous calls. The behaviour of each *clocked-entity* may be executed in parallel and may depend on the clock scheme chosen. Sending the tick will simply cause a thread associated with the entity to be executed. Synchronisation between entities is performed with signal communications. The only constraint is that a status packet must be sent back to the clock when the task related to the cycle is complete.

Basic component behaviour can be now reused with a low dependence on the synchronisation mechanism. Let us see how the modification of the abstract behaviour which an entity inherits from the *clocked-entity* may determine its synchronisation policy.

3.3 Implementation with abstract entities

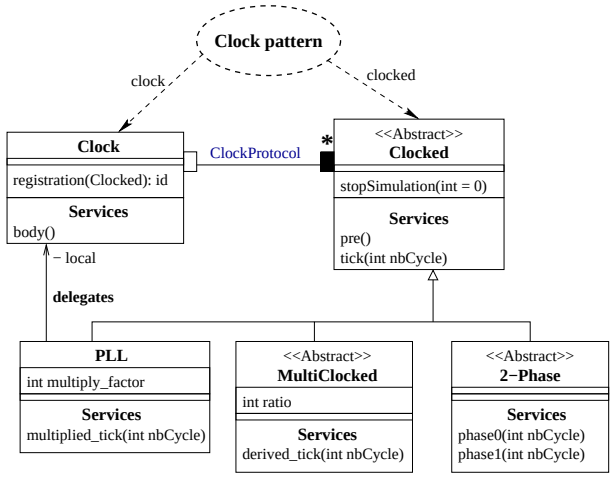

Figure 2. Behaviour associated with the clock pattern.

Figure 2 presents the implementation of the clock-pattern in HASE. The *Clock* class implements the *clock* role of the pattern. Its behaviour is described with a Hase++ description in accordance with the algorithm shown in the previous section. The *Clocked* class implements the *clocked* role of the pattern (section 3.4). Two possible extensions are shown in Figure 2: MultiClocked (section 3.6) and 2-Phase (section 3.5). A further possible extension to model components which are deselected to reduce power consumption when not being used, is described in section 3.7.

3.4 Simple clock mechanism

The simplest clock mechanism (figure 3) consists of registering with the clock, synchronizing by signal communications and executing the tick service in each cycle. Each HASE entity may have a *startup* section and must have a *body* section. The startup section of each entity is executed prior to any execution of the body behaviour. Consequently, registering in this section guarantees that all clocked entities will have been registered before the first tick. The *pre* service is called to allow variable initialisation. Finally, after each event from the clock entity, the tick

service is executed and status information is sent back to the clock.

At any time, the entity may decide to interrupt the execution by calling the inherited method *stopSimulation*. This synchronous call will modify the status information sent back to the clock. Depending on that information, the clock entity may decide to completely interrupt the execution or even to send an error message. The stopSimulation method is the implementation of the setStatus method defined by the pattern. The name has changed because stopSimulation is in that case more explicit than setStatus, though it has exactly the same behaviour.

```
$class_decls

int STATUS;
//entity references
  sim_entity_id clk;
  int clockCycle;

  void stopSimulation(int val=0);

$class_defs
  clockCycle = 0;
  void clocked::stopSimulation(int val) {
     STATUS = val; }

$startup
  clk     = sim.get_entity_id("CLOCK");
  ((clock *)sim.get_entity(clk))-
>registering(get_id());

$body
   pre();
   STATUS = 1;

// predicate
  sim_from_p start(clk);

   while (1) {
        sim_wait_for(start, ev);

        SIM_CAST(int, clockCycle, ev);

// Execute the behaviour associ-
ated with tick
        tick(clockCycle);

        sim_schedule( clk, 0.0, CLOCK,
                      SIM_PUT(int, STATUS));
   }
```

Figure 3. Possible implementation of the clocked entity

3.5 Bi-phase clock mechanism

This is a first possible extension of the clocked entity. The tick service is overriden to call the *phase0* service for the even cycles and the *phase1* service for the odd cycles. The actions of the entity are computed during phase0 and the output signals are updated during phase1. This guarantees that all computations are complete before any output signal is updated. The result cannot depend on the order of execution. Nevertheless this kind of scheme forbids instantaneous loops if no mechanism is provided to compute the fix-point before updating output signals.

The 2-Phase abstract entity may inherit the primary behaviour of clocked and the Hase++ description should only define the tick service as an alternate call to phase0 and phase1 services:

```
$tick(cycle)
   if (cycle%2 == 0 )
      phase0(cycle/2);
   else
      phase1(cycle/2);
```

3.6 Multi-clock systems

When modelling ASICs, different parts of the whole system often have different clock-periods. Each subsystem has its own frequency, while communications between subsystems are still synchronous relatively to the reference oscillator. One way to model a system which contains several subsystems with different clock frequencies is to choose a clock, the frequency of which is the least common multiple of all the subsystem frequencies. Then using a Multi-Clocked abstract entity and varying the ratio parameter will lead to very simple model for such a system:

```
$tick(cycle)
   if (cycle%ratio == 0 )
      derived_tick(cycle/ratio);
```

In this case, all subsystems will run at the greater frequency and only call the *derived_tick* service at the rate $\frac{1}{ratio}$. This implementation should only be used when few entities run at a different frequency, however, because two signals are exchanged between the clock and the Multi-Clocked entity whether or not the derived_tick service is executed.

Typically, in real sytems, a single-reference low-frequency oscillator is distributed to each component in the system where it is multiplied by a local phase-locked loop (PLL) component (figure 4 [9]).

To simulate this arrangement, a PLL entity must be defined. This entity inherits from the basic abstract *clocked-entity* and registers with the main clock. Each component of a subsystem registers with the PLL entity instead of registering with the main clock. Then for each cycle of the main clock, several ticks are sent to each registered component with the same algorithm as for the clock entity.

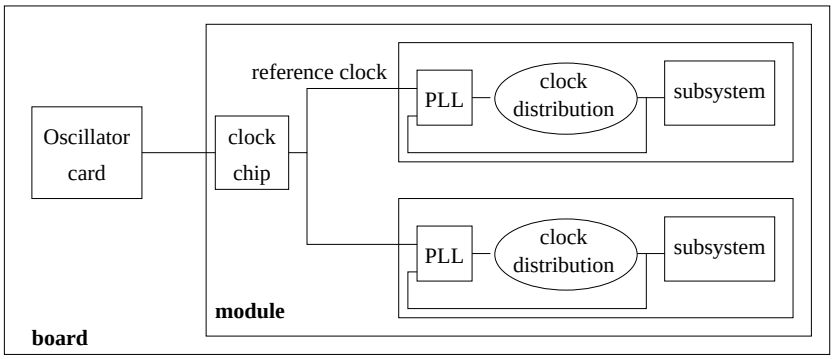

Figure 4. Clock distribution when subsystems have different frequencies.

A multiplication parameter is used to choose the correct frequency in relation to the frequency of the main clock. The abstract *clocked-entity* provides an additional parameter to choose the clock component with which it has to register (main clock or a specific PLL).

The PLL entity collects status information signals from each component of the subsystem and sends back an aggregated status information signal synchronously with the main clock. The reuse of the clock is performed by delegating the whole behaviour of the PLL to a private local instance of the clock entity.

Therefore, the PLL entity implements both the clock and the clocked role of the clock-pattern, the clock role relative to the sub-components and the clocked role relative to the main clock. The clock is a low frequency entity, and this basic frequency is multiplied on demand by PLL entities which feed higher frequency subsystems.

3.7 Deselected components

To reduce power consumption, some parts of embedded systems may be deselected, *i.e.* they do not receive the clock any longer and then do not consume power. To model the deselection of components, we just have to define a new status value: DESELECT. Then, the clock algorithm has to be slightly modified to remove from the list of clocked components the component which has sent such a value. Doing so, this component will not receive the clock signal again until it re-registers. This re-registering has to be performed by a clocked component responsible for such a task or directly by the clock entity. If the task is performed by a separate component, then no modification of the clock is required, but this component should never deselect itself as long as some components may have to be re-registered.

4 Mesh networks

Mesh networks are architectures consisting of the aggregation of several basic components communicating through a specific scheme. HASE provides a mechanism to automatically create n-dimensional meshes based on the definition of a single component.

For example a 4-dimensional mesh can be defined in the entity library of an EDL description as followed:

```
MESH4D archi (
  ENTITY_TYPE (asic)
  SIZE1 (4) SIZE2 (3) SIZE3 (2) SIZE4 (2)
  NO_LINKS(2)
  WRAP(0)
  DESCRIPTION("4D Mesh pattern")
  PARAMS()
)
```

The ENTITY_TYPE attribute identifies the type of the entity to be replicated in the mesh. The SIZEn attributes specify the size of the mesh for each dimension. The NO_LINKS attribute specifies communications either as unidirectional or bidirectional, and the WRAP attribute specifies whether or not the mesh has to be wrapped at the border.

From this description, HASE instantiates enough components (4x3x2x2 in this example) and creates communication links among components through all the dimensions.

Using the original HASE clock synchronisation mechanism, whenever the dimension or the size of the mesh was modified, the list of entity links had to be modified manually in the clock entity file in order for it to feed all the automatically generated sub-components. The new clock-pattern sloves this problem. As long as the base entity (*asic* here) inherits from one of the abstract *clocked-entities*, then each generated component of the mesh will automatically register with the clock entity and behave as specified by the clocked description. This will work for any dimension and any size.

5 Conclusion

We have demonstrated how some basic object-oriented features can enhance models of hardware architectures.

The proposed clock-pattern allows designers to simply reuse existing components and modify the synchronisation mechanism, it also facilitates the generation of mesh networks and shortens the modification time due to dimension or size modifications.

Others points can also be improved with such mechanisms. For example, the use of object-orientation to describe the instruction-set of modelled architecture may facilitate the definition of several addressing modes, and may enhance the reuse of others instruction-set descriptions to create new ones.

More information about HASE is available at http://www.dcs.ed.ac.uk/home/hase.

Acknowledgements

HASE was developed as part of the ALAMO project supported by the UK EPSRC under grant GR/J43295. The simulation study of the UKQCD computer architectures is supported by the UK EPSRC under grant GR/R27129.

References

[1] D. Chen, N. Christ, Z. Dong, A. Gara, R. Mawhinney, S. Ohta, T. Wettig, "QCDOC Architecture" Internal Report, Columbia University, May 2000

[2] P.S. Coe, F.W. Howell, R.N. Ibbett and L.M. Williams: A Hierarchical computer Architecture design and Simulation Environment, *ACM Transactions on Computer and Modelling Simulation*, 8(4), 1998, 431-446.

[3] L. Williams and R.N. Ibbett: Modelling the DASH architecture in HASE, *29th annual Simulation symposium*, New Orleans, July 1996.

[4] R.N. Ibbett, HASE DLX Simulation Model, *IEEE*, Vol 21, Jan-Mar 1999, 24-33.

[5] R.N. Ibbett, P.E. Heywood and F.W. Howell, HASE: A Flexible Toolset for Computer Architects, *The Computer Journal*, 38(10), 1996.

[6] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns – Elements of Reusable Software*, Addison-Wesley, Oct. 1994, ISBN 0-201-63498-8.

[7] B. Selic, G. Gullekson and P.T. Ward, *Real-Time Object-Oriented Modeling*, John Wiley & Sons, inc. 1994.

[8] B. Selic and J. Rumbaugh, *Using UML for Modeling Complex Real-Time Systems*, http://www.ObjecTime.com, 1998.

[9] G.A. Van Huben T.G. McNamara and T.E. Gilbert: *PLL modeling and verification in a cycle-simulation environment*, IBM Journal of Research and Development, 42(5/6), 1999, 915-925.