DCim++: A C++ Library for Object Oriented Hardware Design and Distributed Simulation

Hadi Esmaeilzadeh, Amir Moghimi, Eiman Ebrahimi, Caro Lucas, Zeinalabdin Navabi, and Sied Mehdi Fakhraie

School of Electrical and Computer Engineering, University of Tehran, North Kargar Ave., Tehran 14395-515, Iran

hadi@cad.ece.ut.ac.ir, a.moghimi@ece.ut.ac.ir, e_ebrahimi@mail.utexas.edu, lucas@ipm.ir, navabi@ece.neu.edu, and fakhraie@ut.ac.ir

Abstract—DCim++ is a C++ library developed for object oriented hardware design, modeling and distributed simulation. DCim++ enables C++ to be used as an OO HDL, which supports concurrency in description, inheritance in design and distributedness in simulation. Design simulation results are obtained by running C++ programs on a network of workstations. The Message Passing Interface (MPI) library has been used in the implementation of DCim++ as the basis of communications required for distributed simulation. In our simulation scheme, we have not considered any central management unit in order to defy performance degradation, instead only a coarse-grain synchronizer is used to keep the distributed components synchronized. This paper explores the structure of the DCim++ library and its mechanisms. The process a designer has to go through in order to design a system using DCim++ and conduct its distributed simulation leaving communication complications to DCim++, has also been presented. Finally, the results of our uniprocessor and distributed simulations for ISCAS benchmark circuits show high degrees of performance gains.

Index Terms—Distributed simulation, object-oriented design, high-level hardware specification, parallel computing.

I. INTRODUCTION

OBJECT-ORIENTED and distributed computing concepts have been vastly generalized to all aspects of system design. Besides the ever-increasing application in software engineering, these concepts are now becoming among the main basics of system engineering [2][3]. As a result of the philosophical shift from functionalism to structuralism, the designer needs not focus on performing a task in a top-down manner. The behavior of the total system can be emergent rather than planned. New tasks can be performed as a result of novel collaboration patterns among existing objects of a system. Also, the reusability of the design process can be considerably enhanced by utilizing inheritance. Although hardware engineering has lagged behind the software engineering community in adopting modern bottom-up design technologies with emergent consequences, structuralist thinking is in many respects inherent to the former domain.

Additionally, as the size of the designs to be simulated rapidly grows, the need for fast simulation tools cannot any more be satisfied by sequential simulators [4]. One of the approaches to a solution for this problem consists of parallelizing the system and distributing its execution on more than one processor. This can be facilitated by exploiting the inherent parallelism of the models to be simulated. Hardware systems have many inherent parallel features. This fact together with the large time consumption of the simulation of these systems has led to the utilization of distributed simulation schemes.

In [5], a parallel logic simulation scheme on a network of workstations using a parallel virtual machine (PVM) has been presented. This scheme uses an event-driven engine as its simulation base. Another approach to parallel simulation of digital systems has been by way of distributing sub-tasks of existing simulation-engines such as VHDL which is presented in [6]. There has also lately been work on the distribution of the Verilog simulation engine sub-task for the simulation of ever-growing digital systems [7].

This work proposes DCim++, a C++ library based on Cim++ [1] as a framework for object oriented hardware design and distributed simulation. This framework supports concurrency and inheritance in the phase of design, and eliminates the need for any explicit simulation engine in the simulation phase. A designer using DCim++ utilizes this library without the concern of communications which will eventually be needed for distributed simulation because these are implicitly accounted for in DCim++. This distribution eventually gifts the designer with a gain in time of simulation without complicating his work beyond his own design problem.

DCim++ is extensible in two ways. Firstly because of its C++ nature, it is useful in heterogeneous paradigms such as hardware/software co-design. On the other hand, it can also be
extended to a framework for multi-agent system design.

Section II discusses the Cim++, the base of DCim++, its structure and simulation scheme and design examples. Section III describes the approaches and mechanisms employed in DCim++. Section IV presents the results of applying our method to ISCAS benchmark circuits, and the paper is concluded in Section V.

II. **CIM++, OBJECT ORIENTED DESIGN AND SIMULATION**

At the first glance, it seems that hardware modules are the most important objects of a hardware system, but in fact, such a system is a composition of some wires connected to each other through some operations. Hence, what a system or circuit performs is changing values on these interconnections.

The Cim++ [1], library is a set of classes that encapsulate various kinds of interconnecting wires. Nevertheless, hardware modules cannot be ignored. They are living components, continuously converting values on their incoming wires to values on their outgoing ones.

In Cim++, each hardware module is an object which has some evaluation functions (we call them always-functions) that always map values on the inputs to values on the intermediate and output wires. These objects can be considered as sleeping components waking up on the occurrence of events on their inputs. The wake up strategy is implemented as a loop, which in each of its iterations gives the control to the components of the system to verify their input wires and execute their evaluation function(s) if it is necessary. These new values must be held pending until the next iteration. The main benefit of this scheduling method in which each component is responsible for its functionality is that it can easily be extended for distributed simulation. The library DCim++ described in the next section utilizes this scheduling method as the starting point of adding parallel simulation capabilities to Cim++.

Cim++ uses a C++/Verilog like script for hardware which is intended for simplify the description process and hiding simulation semantics from the designer. This script called CimScript is then translated into Cim++ classes. (See Fig. 1 and Fig. 2 as sample scripts).

![Fig. 1. Register CimScript.](image)

Natively, C++ supports inheritance as a basic object oriented construct, and Cim++ utilizes this existing feature for hardware design, which is illustrated with an example.

Consider that a counter is-a register with counting capability. Therefore for counter implementation, first a typical synthesizable register is implemented using Cim++ (see Fig. 1), then a counter is derived from it (see Fig. 2).

```cpp
class Counter public Register {
    input WireBit cntEn
    always void evaluateData(reset, posedge | clk, posedge) {
        Register::evaluateData();
        if(cntEn == 1) data <= data + 1;
    }
}
```

Fig. 2. Counter CimScript.

III. **DCIM++, VIEWS AND APPROACHES**

DCim++ is a C++ library built on top of the Cim++ library. The aim of this new library is to extensively use the distributed potentiality of the Cim++ simulation scheme. Thus, DCim++ is a combination of a parallel simulation approach based on MPI library and object-orientated property of Cim++. The base structure on which the DCim++ library has been built gives the largest of leads towards defining a distributed simulation bed.

In DCim++, hardware components are all inherited from a class called DCimCore. DCimCore is a pure abstract class which acts as a wrapper around previously implemented CimCore classes. Each DCimCore object is a hardware component configured for distributed simulation, which has the ability to communicate with other DCimCore objects through the Message Passing Interface (MPI) library. The CimCore object inside each DCimCore is intended to take care of internal simulations of that hardware object. Each DCimCore also aggregates an Interface, which is responsible for the communications of that object with its peers.

With the above mentioned terminology the computation and communication of each hardware object have been separated, which is an illustration of the “separation of concerns” concept in object oriented design. As a result, the designer focuses only on the design of each hardware object leaving the communications to be done automatically by DCim++.

The scheme used for simulation of Cim++ objects is not a centralized one. The strategy is implemented as a wakeup loop, in which each component is woken up to observe its inputs and as it is then responsible for any events that might be generated on its inputs, the component generates the appropriate results. This loop keeps iterating in the current time step until there are no more events to be accounted for throughout the system. This aspect of the Cim++ scheme, which is used as the base of the DCim++ library, is what gives this simulation scheme a great potential for distributedness. That is, a system to be put under distributed simulation may be partitioned into discrete subsystems or components with specific interconnections in-between. As each of these components is responsible for its own events, each can be chosen to run on a stand-alone workstation.
Considering the above, the communications of each hardware object are left to be done by the DCim++ library. As mentioned before, Interfaces are responsible for communications of DCimCore objects. Interfaces are actually composed of a number of Channels and also access mechanisms for the CimCore inside the DCimCore object to read/write data from/to the outside world. Channels are where the primitive communications with the outside world are implemented using the MPI library. Channels encapsulate a bunch of wires connecting two peer components.

Synchronization is a matter of concern in any type of simulation, distributed or not. In our distributed environment, the occurrence of any event internal to a hardware component is completely independent of its peers. But, it is necessary to synchronize events happening between the components.

Each process responsible for a particular component of the whole system runs on a workstation evaluating its input values using the functionality defined for it and thus generating possible events on its output wires. In order to be able to safely apply these new values on the wires each component is attached to, it needs to have permission from somewhere in order not to ruin the other components evaluation of their inputs. Synchronization of discrete components running on different workstations is maintained using a light-weight coarse-grain synchronizer object.

Each component writes its results to its output channels every time it evaluates them. When doing so, it also informs the coarse-grain synchronizer of the occurrence of a probable event as a result of this new result. The synchronizer’s duty is to wait for this information from all the components, consequently granting all of them the permission to read their inputs for the next iteration of their simulation loop on the appearance of an event. The simulation loops continue to iterate until no component in the system has any new event to introduce, leaving the system in a stable situation and ready to go onto the next time step.

IV. EXPERIMENTAL RESULTS

The parallel simulation scheme described in the previous section has been implemented on a network of workstations and verified using ISCAS-89 benchmark circuits. All simulations use Intel 2Ghz Pentium IV workstations connected through a switched 100Mbps Ethernet network.

The results of the uniprocessor simulations of a number of ISCAS-89 sequential benchmark circuits are presented in Table I. These simulations were conducted using the Cim++ library, where executable code generated by compiling the C++ description of the circuits was used. These simulations have been performed at the gate level where the clock period was set to 2 time units. We chose this clock period because our aim was to measure simulation times considering full activity and without delay. Random input values have been used as primary inputs except for the clock. Each circuit has been simulated for $2^{16} (= 65536)$ clock cycles.

A number of the experimented circuits with larger gate counts of the ISCAS-89 sequential benchmark circuits were chosen to go under distributed simulation using our method described in the previous sections. These circuits were partitioned using a random method. A CimCore was designed for each of the partitions respectively and then a DCimCore was implemented for each in order to be able to use them in parallel simulation. Finally the distributed simulation was set up and performed in a manner where each DCimCore was set to be simulated on a workstation of its own. The speedup results achieved for these simulations are presented in Table II.

It can easily be observed from Fig. 4 that the speed of simulation increases (i.e., average time to simulate one time step decreases) as the circuits put under test grow larger, but with the addition of more workstations the speedup tends to reach a saturation state. The fact that the overhead for
communication starts to overtake the computational parallelism offered by the increased number of workstations is why the saturation of speedup is seen.

It is considerable that because our design and simulation schemes are different from engine-based simulations in the respect that design is done in C++ and simulation takes place by execution of binary code, our scheme offers much better speedup results with respect to similar engine-based methods.

TABLE I

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Gate Count</th>
<th>Number of Workstations</th>
<th>Time Steps Total Simulation</th>
<th>One Time Step Average Simulation</th>
</tr>
</thead>
<tbody>
<tr>
<td>s400</td>
<td>106</td>
<td>2</td>
<td>25.020</td>
<td>0.191</td>
</tr>
<tr>
<td>s526</td>
<td>141</td>
<td>3</td>
<td>34.574</td>
<td>0.264</td>
</tr>
<tr>
<td>s713</td>
<td>139</td>
<td>4</td>
<td>33.446</td>
<td>0.255</td>
</tr>
<tr>
<td>s9234</td>
<td>2027</td>
<td>5</td>
<td>510.958</td>
<td>3.898</td>
</tr>
<tr>
<td>s15850</td>
<td>3448</td>
<td>6</td>
<td>884.961</td>
<td>6.752</td>
</tr>
<tr>
<td>s35932</td>
<td>12204</td>
<td>7</td>
<td>3.216.166</td>
<td>24.537</td>
</tr>
<tr>
<td>s38417</td>
<td>8705</td>
<td>8</td>
<td>2.215.287</td>
<td>16.901</td>
</tr>
<tr>
<td>s38584</td>
<td>11448</td>
<td>9</td>
<td>3.069.407</td>
<td>23.418</td>
</tr>
</tbody>
</table>

TABLE II

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Number of Workstations</th>
</tr>
</thead>
<tbody>
<tr>
<td>s35932</td>
<td>1.92</td>
</tr>
<tr>
<td>s38417</td>
<td>1.71</td>
</tr>
<tr>
<td>s38584</td>
<td>1.75</td>
</tr>
</tbody>
</table>

Fig. 4. Speedup changes versus the number of workstations showing where and how saturation occurs.

V. CONCLUSION

The DCim++ class library, structure and mechanisms developed for object oriented hardware design and distributed simulation was presented. Support for description concurrency, design inheritance, distributed simulation in DCim++ was also explored. We showed that a designer need only focus on the design of his/her system in an object oriented manner, leaving the communications and synchronization problems in distributed simulation of his/her design for DCim++ to handle. ISCAS benchmark circuits were used to show the gain of our method, and the fact that speedups between 1.7 and 3 were obtained shows the capabilities of this scheme.

Employing an effective and optimum partitioning scheme for our distributed simulation scheme, for instance using genetic algorithms is our future step. DCim++ is also a promising bed for implementing a multi-agent design framework.

REFERENCES