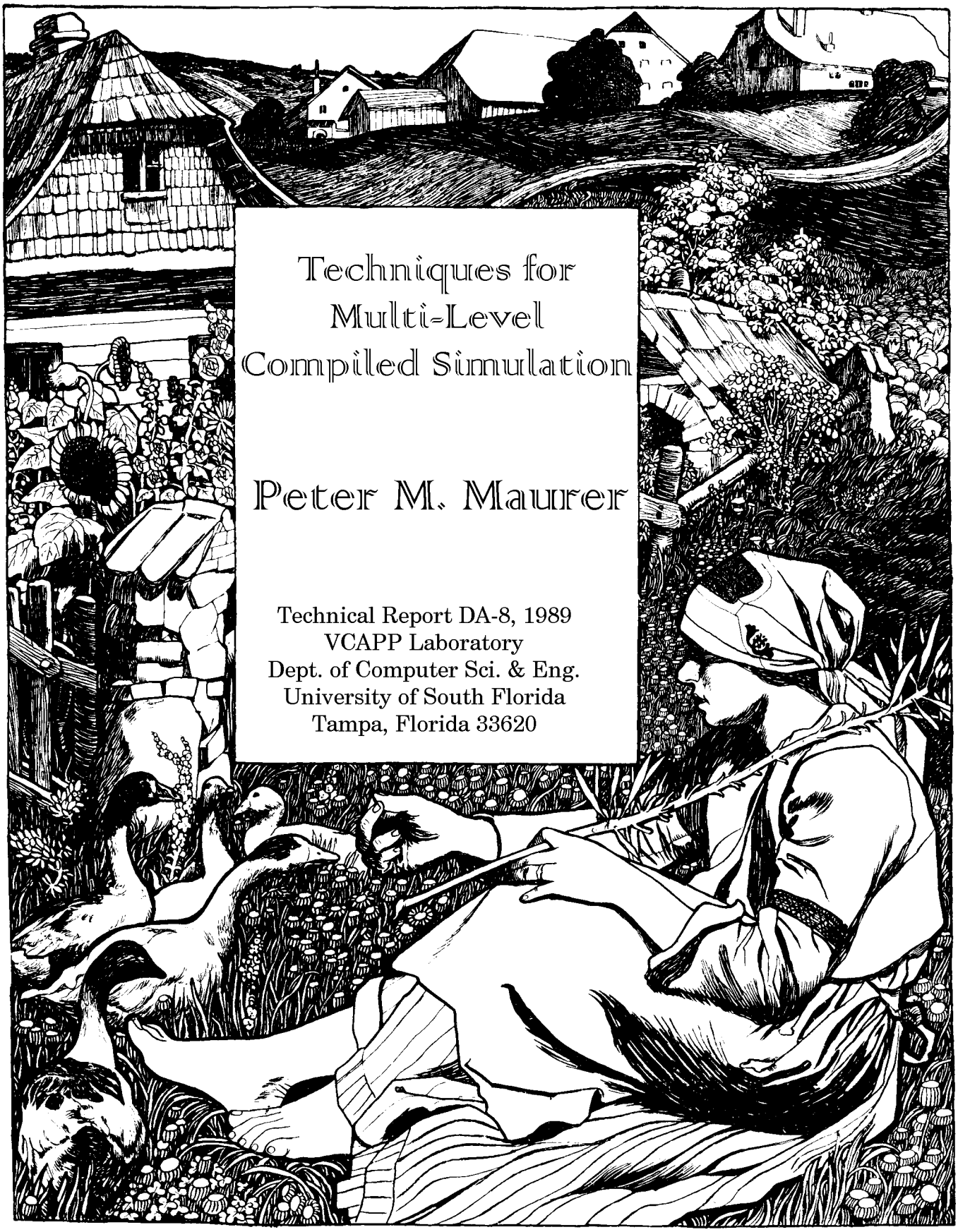

Techniques for Multi-Level Compiled Simulation

Peter M. Maurer

Technical Report DA-8, 1989
VCAPP Laboratory
Dept. of Computer Sci. & Eng.
University of South Florida
Tampa, Florida 33620

TECHNIQUES FOR MULTI-LEVEL COMPILED SIMULATION

Peter M. Maurer
Zhicheng Wang
Craig D. Morency
Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620

ABSTRACT

This paper begins attacking the problem of multi-level compiled simulation by presenting a solution to the problem of independent compilation of subcircuits. The solution to this problem is an interface data-structure that contains a public section for I/O ports and a private section that contains internal signals and maintains internal states. One data structure is created for each instance of a subcircuit. The data structures are retained by the circuit-simulator that *contains* the instance, but are created by the simulator *for* the instance. This technique prevents changes in the subcircuit from forcing the recompilation of all circuits containing instances of the subcircuit. Since subcircuits can be compiled independently of one another, they can be compiled using different compilers. Furthermore any subroutine that complies with the interface specifications can be incorporated into the simulation. Translation tables are used to move one logic-system to another. Finally, the paper discusses compilation techniques for cyclic circuits.

TECHNIQUES FOR MULTI-LEVEL COMPILED SIMULATION*

Peter M. Maurer
Zhicheng Wang
Craig D. Morency
Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620

EXTENDED SUMMARY

1. Introduction

Recent papers have described complied simulation of circuits at various different levels [1][2][3]. Compiled simulation is appealing because it has the potential for providing much faster simulations than can be obtained using conventional interpretive methods. On the other hand, because circuit compilers are much slower than the simple parsing algorithms used by conventional simulators, changes in the circuit take more time. The problem is compounded by the fact that most existing circuit compilers must recompile the entire circuit even if only a small portion of it changes. Although independent compilation of subcircuits would reduce the time needed to make changes, it increases the complexity of the circuit compiler. Because of the complexity of circuit compilation, it has made sense to avoid the additional complexity of independent compilation. Although there are still many problems to be solved, circuit compilation is now well enough understood that it is time to begin focusing both on independent compilation and the problem of integrating different levels of compiled simulations.

Multi-level simulation is appealing because it allows a circuit to be designed piece by piece, and allows new portions of the circuit to be verified as they are completed[4][5]. Conceptually one could begin with a behavioral simulator to "prove the concept," and gradually replace parts of the simulation with logic-level or switch-level models. Currently our compiler supports the behavioral level, the algorithmic state-machine level, the RTL level, and the logic-gate level. At this time we support the behavioral level by allowing

* This research was supported by the Defense Advanced Research Projects Agency under grant 2114-033-LO and by the University of South Florida Center for Microelectronics.

subroutines written in conventional programming languages to be integrated with the output of our compiler, but we are in the process of designing a true behavioral-level language. At the RTL level we support microprogramming of both ROMs and PLAs. We also support simulations at the switch level by integrating the output of the COSMOS compiler[2] with our output. Other extensions are planned.

Two less crucial issues that affect the design of a compiled simulator are the choice of source and target languages. Although the choice of source language is somewhat a matter of taste, two natural choices are VHDL[6] and EDIF[7], because they are published standards. In spite of this, we chose instead to design simpler more compact language to speed up laboratory work and reduce the time needed to teach the language. We have designed translators to transform our language into EDIF, and we have designed our software so that we could easily adapt it to some other source language should it be necessary to do so.

The target language should be a machine language or some high-level language such as C or PASCAL. Compilation will probably be faster if a machine language is chosen, but the compiler will not be portable. If a high level language is chosen, code generation will probably be easier and the circuit compiler will be portable, but the high-level language must provide the bit-level operations needed for simulation. Using a high-level language has the additional benefit of reducing the number of optimizations done by the circuit compiler. Such things as eliminating consecutive NOT operations and performing constant arithmetic can be left for the the high-level language optimizer. We have chosen to use C as our target language because of its bit-level operations and because of the nearly universal availability of UNIX† and C.

In the remainder of this paper we discuss the problem of interfacing the output of separate compilations, and the integration of the output of different compilers. We also discuss the problem of generating code for various different levels of circuit descriptions.

2. Independent Compilation of Subcircuits

As stated above, the compilation process is simpler if one does not permit independent compilation of subcircuits. In this case it is the usual practice to flatten all hierarchically specified circuits thereby allowing the code generator to deal with a single circuit. Even if circuit-flattening is not performed, the fact that the compiler can see the entire circuit at once simplifies the passing of data between various parts of the circuit. For example, one can

---

† Unix is a Registered Trademark of AT&T

use global variables instead of parameters to pass data to subcircuits. On the other hand, every change to the circuit, no matter how small, forces recompilation of the entire circuit.

Our initial experimentation with just such a compiler has pointed out other benefits of independent compilation. We have found that flattening large circuits sometimes creates programs that are too large for our C compiler. Furthermore, when we began adding more and more different levels, we found that the compiler was becoming unmanageably complex. In fact, we first began looking at independent compilation as a method for integrating the output of several different compilers. (Although we use several compilers, we are still able provide the illusion of a single compiler by splitting a multi-level circuit specification into blocks and passing each block to the appropriate compiler.)

To see why independent compilation is difficult, consider the circuit whose block diagram appears in Figure 1.

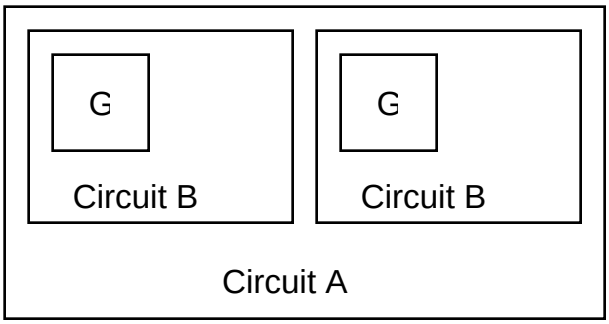

Figure 1. A Hierarchical Circuit Description.

Circuit A contains two instances of circuit B, and each instance of B contains a flip-flop named G. Let us assume that the state of G is maintained in a variable named G_value. The simulator for circuit A must maintain two copies of G_value, one for each instance. Now suppose we compile circuit B independently of circuit A, and suppose we add a second flip-flop to circuit B. The simulator for circuit A must now keep track of two variables per instance instead of one. Nevertheless, it is desirable to avoid recompilation of circuit A unless the *interface* of circuit B changes.

Now, suppose it is necessary to monitor the output of gate G in one instance of B, and suppose that the output is not a primary output of circuit B. In once sense this output is a local variable of circuit B, but our need to monitor it forces it to be globally accessible. Compilers that do not allow independent compilation typically solve this problem by flattening the circuit and assigning all monitorable signals to global variables. For independent compilation, a different solution is needed.

Our solution to these problems is the data structure pictured in Figure 2. We use one such structure to represent each instance of an independently compiled subcircuit.

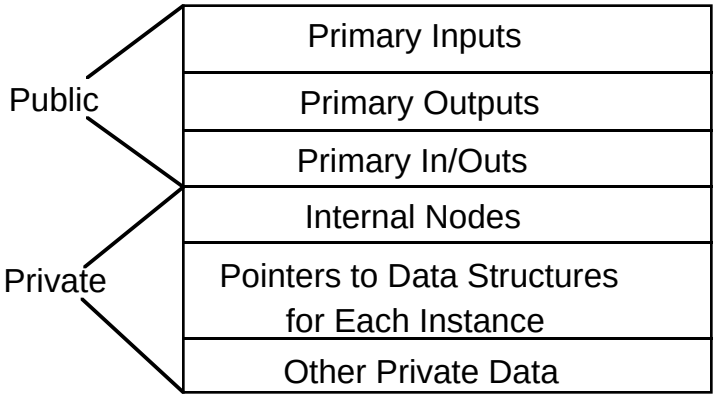

Figure 2. The Data Structure Representing an Instance.

The compilation of a subcircuit produces a single subroutine which must be called to simulate each instance of the subcircuit. The address of a data-structure of the type pictured in Figure 2 is passed as an argument to the subroutine. Before calling the subroutine, the caller copies the values of the inputs into the data structure, and after the subroutine returns the caller retrieves the values of the outputs from the data structure. The caller does not touch the private section of the data structure. The private portion of the data structure may contain many different kinds of data depending on the needs of the subroutine.

To avoid forcing a recompilation of the caller when the private section of the data structure changes, the *called* subroutine is given the responsibility for creating and initializing the data structure. When the subroutine is called with a NULL argument it allocates and initializes a new data structure instead of performing a simulation. The address of the new data structure is returned to the caller.

To illustrate, suppose the circuit pictured in Figure 1 is being simulated. A simulation kernel will be used to read user commands and test vectors, and to call the simulation subroutine of circuit A. During initialization, the simulation kernel calls the subroutine for circuit A with a NULL argument to create a new data structure for circuit A. The subroutine for circuit A allocates a new data structure, and calls the subroutine for circuit B with a NULL argument twice to allocate the data structures for the instances of circuit B. Pointers to these data structures will be stored in A's data structure. Finally, the address of A's data structure is returned to the simulation kernel. Because the states of all internal nodes of all instances are accessible through the data structure, the simulation kernel can monitor the state of any internal node.

To provide a "map" between the name of a particular node and the location of the node's value in the data structure, the compilation of a circuit produces a dictionary file that contains the name of all inputs, outputs, and internal signals. It also contains instance names and subcircuit names for each independently compiled subcircuit used by the circuit. The dictionary files are linked into a master dictionary which becomes part of the simulation kernel. The simulation kernel uses the dictionary to locate the current value of a node in the data structure.

In any multi-level simulation, it is possible for different levels of the simulation to use different sets of logic values. At the behavioral level, two-valued logic will usually suffice, while at lower levels three, four, or more values may be used[8]. It is necessary for different levels to communicate even though they use different sets of logic values. To solve this problem we allow the user to include an empty definition of the subcircuit that declares the logic-system to be used by all instances of the subcircuit. If such a definition is omitted the compiler will assume that the logic-system of the subcircuit is identical to the logic-system of the containing circuit. When the logic-systems differ, the compiler provides a translation arrays to map logic values between the two systems. These arrays are used when copying input values into an instance's data structure, and when output values are retrieved from the data-structure. Some loss of accuracy must be expected when translating between different logic-systems.

3. Integrating Our Compiler with Existing Compilers

Although our data-structure method is quite general, it requires us to write a new compiler for each simulation level, even when there is an existing compiler for that level. This entails an enormous amount of work, which we naturally wish to avoid. We would prefer to integrate the output of existing compilers with the output of our own compiler, but there are several problems that complicate such integration. First, the compiler may generate code containing global variables, making it impossible to create more than one instance of the circuit. Second the global variables may have identical names for different circuits and the compiler may generate two different but identically named subroutines for two different circuits. This makes it impossible to include instances of both circuits in one simulation. Third, the compiler may include code to parse test vectors and print output which would interfere with our own input/output routines. Nevertheless, we have developed procedures for integrating the output of "foreign" compilers with our own.

Our first step is to identify that portion of the compiler that generates code for simulating *one test vector*. We retain this portion of the compiler and strip out everything else. Next, we force the output of the compiler to be placed in a single output file and

modify the compiler to add a prefix to the name of each global variable. We then change their declarations to make them static *local* variables. We add a library subroutine that contains uninitialized declarations of the original global variables. This enables the code for several circuits to "time-share" the global variables. We can't simply eliminate the global variables because they may be accessed by some common library routines. Next, we modify the declarations of all generated subroutines to make them local subroutines instead of global. (Since they are now all in one file, their names will be known to one another.) Finally we add a global driver routine that will be called to simulate the circuit on one test vector. The driver routine performs a dynamic translation from our data-structures into the form required by the generated code and takes care of time-sharing global variables.

As with subroutines generated by our own compilers, the driver routine allocates and initializes a data structure when it is called with a NULL argument. In addition to the usual contents, the data structure contains copies of all global variables used by the generated code. When the global subroutine is called with a non-NULL argument, the global-variable values from the structure are copied into the actual global variables. The data structure is used to create one test vector, and the circuit is simulated once using this vector. The output values and the values of all global variables are then copied back into the data structure.

We have used this procedure to integrate subroutines generated by COSMOS[2] with output from our own compilers.

4. Interfacing with Non-Compiled Simulations

It is our intent to provide methods for integrating our compiled simulations with non-compiled simulations of certain subcircuits. This will enable us to take advantage of the features of some widely available simulators while retaining the benefits of multi-level compiled simulation. We have not yet implemented these procedures, so they remain tentative. Although these procedures are straightforward they require modifications to the non-compiled simulator, which tends to preclude their use with commercial simulators.

The compiled simulator must create a circuit description acceptable to the non-compiled simulator, and must generate a driver routine similar to the one we use to integrate "foreign" compiled code. When it is called with a NULL argument, this routine initiates a new process to run the non-compiled simulation, and creates a new data structure.. When it is called with a non-NULL argument, the routine will create a new test vector, perform the necessary interprocess communication to pass the input vector to the non-compiled simulation and retrieve the output vector from the simulation. The non-compiled simulator must also perform the necessary interprocess communication to receive its input vectors

and transmit its output vectors. Because the most sophisticated facilities for interprocess communication vary widely from one version of UNIX to another, use of these facilities may limit the portability of the simulator and the compiler.

Lee and Feng [14] have used a simular technique technique to integrate SPICE analog simulations with a switch-level digital simulator.

5. Generating Source Code

At most levels, generating code for individual objects (gates, equations, etc.) is straightforward. The primary problem is scheduling the code for the various objects, i.e. determining the order in which to generate code for the objects. A technique known as levelized scheduling has been used for logic-gate and timing simulation[1][9]. Another technique known as data flow analysis has been used for logic-equation and switch-level simulation[3][10]. We have chosen to use levelized scheduling, but either technique will give acceptable results. We have also developed a technique called SCS scheduling that gives more optimal schedules than either levelized scheduling or data-flow analysis for certain types of circuits. See [11] for details.

Although these algorithms are well known, they are not complete solutions to the scheduling problem. In their purest form, they can be applied only to acyclic circuits. For high-level circuit descriptions it may be acceptable to simply prohibit cycles[3], but in most cases such a restriction would make the compiler practically useless. The major problem in generating code is finding suitable techniques for handling cyclic circuits.

6. Generating Code for Cyclic Circuits.

Cyclic circuits can be handled in several different ways. The simplest is to treat flip-flops as primitive elements and insist that every cycle contain at least one flip-flop. The circuit is then broken at each flip-flop and levelized scheduling (or data flow analysis) is used on the result. The inputs of each flip-flop are treated as primary outputs, and the outputs are treated as primary inputs. For this procedure to work properly it is necessary to prevent the states of the broken flip-flops from changing until the entire circuit has been simulated. To see why this is so, consider the circuit illustrated in Figure 3.

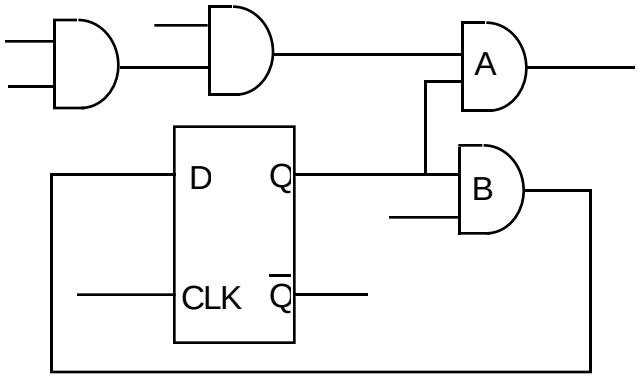

Figure 3. A Cyclic Circuit Broken by a Flip-Flop.

If the output of gate B is allowed to change the state of the flip-flop, the gates A and B will see different values for Q. (This is because A is further from the primary inputs than B.) To keep the state of the flip-flop from changing at unpredictable times is necessary to use two variables to represent the state. One variable contains the current state of the flip-flop and is visible to all gates that use the flip-flop's outputs. The other contains the future state and is invisible to all other gates. When the flip-flop's state changes, the change is reflected in the future state variable, not in the current state variable. After all gates have been simulated the future state variable is explicitly copied to the current state variable.

This procedure will produce correct simulations except when the new state of a flip-flop needs to be immediately visible to another flip-flop (as in an asynchronous counter, for example). One solution to this problem is to provide two types of flip-flops, one that breaks and one that doesn't. The non-breaking flip-flop is treated as an ordinary gate, and is allowed to propagate its output immediately. Since a non-breaking flip-flop cannot be used to eliminate cycles, any cycle must contain a breaking flip-flop.

Another problem is that some of the outputs of the circuit may appear to change one test vector late. The correctness of the simulation is not affected, but it may be difficult to compare output vectors with those obtained from a more accurate simulation. We are currently designing modifications to the scheduling algorithm that will correct this problem.

The flip-flop breaking technique produces fast simulations, because no gate is simulated more than once per test vector. However because feedback is not simulated directly, there may be some loss of accuracy. In particular, glitches, race conditions, and oscillations may not be detected. One method for providing a more accurate simulation is to include an event queue and scheduler and perform an event-driven simulation as is done by the COSMOS compiler[2]. We are investigating a different technique which we

believe will give us similar results, but will eliminate the overhead of handling the event queue. This procedure is not yet fully operational, so these remarks are tentative.

We treat the circuit as a directed graph, find the set of strongly connected components, and schedule the components using the levelized scheduling algorithm. If a strongly connected component contains more than one element (an element is a gate or a net) we flag all back-arcs in the component, and schedule the gates as usual, ignoring the back-arcs. (A back-arc is a net or a portion of a net that goes from a gate on level x to a gate on some lower level y. The lower the level, the closer the gate is to the primary inputs.) The code for the component is placed inside a loop that tests the values of the back-arcs and simulates the component until no back-arc changes, or until a predetermined limit is reached. To eliminate unnecessary evaluations, we place embed portions of the code inside "if" statements that test for changes in the inputs. Much work remains before this technique becomes fully operational.

7. Generating Code for Higher Level Circuit Descriptions

At the RTL and behavioral levels the most important issue seems to be the design of the source language(s) rather than the procedures for generating code[12]. We have designed several RTL-level languages[13], and have found most of the code generation to be straightforward. We encountered a few problems in generating code for algorithmic state machines and in generating efficient code for PLAs. We solved the problem of generating code for algorithmic state machines by using a recursive procedure to generate the code for each state. This procedure traces the flow of the state diagram, and calls itself recursively to generate code for the conditional branches. We are still investigating the problem of optimizing the code for PLAs.

8. Conclusion.

This paper has described methods for doing multi-level compiled simulations and independent compilation of subcircuits. We have presented a data-structure method that allows the output of independent compilations, as well as the output of many different compilers to be integrated into a single simulator. Extensions of this technique can be used to integrate existing compiled and non-compiled simulations with the code produced by our compiler. We have discussed techniques for generating code for cyclic circuits. These techniques, or some combination of them, can be used at many different levels. We have used our compiler both in research projects and in the classroom with great success. We believe that compiled simulation is an attractive alternative that will be widely used in the

future, and we believe that the techniques described here will be of value to anyone who wishes to construct a multi-level compiled simulator.

REFERENCES

1. L. Wang, N. Hoover, E. Porter, J. Zasio, "SSIM: A Software Levelized Compiled-Code Simulator," Proceedings of the 24th Design Automation Conference, 1984, pp. 473-478.

2. R. E. Bryant, D. Beatty, K. Brace, K. Cho, T. Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits," Proceedings of the 24th Design Automation Conference, 1987, pp. 9-16.

3. C. Hansen, "Hardware Logic Simulation by Compilation," Proceedings of the 25th Design Automation Conference, 1988, pp. 712-715

4. M. H. Doshi, R. B. Sullivan, D. M. Schuler, "Themis Logic Simulator - A Mix Mode, Multi-Level, Hierarchical Interactive Digital Circuit Simulator," Proceedings of the 21st Design Automation Conference, pp. 24-31, 1985.

5. C. F. Chen, C-Y Lo, H. N. Nham, P. Subramaniam, "The Second Generation Motis Mixed-Mode Simulator," Proceedings of the 21st Design Automation Conference, pp. 10-17, 1985.

6. "VHDL Language Reference Manual," IEEE Standard 1076, IEEE Computer Society Press, Los Amitos, CA, 1987.

7. "Electronic Design Interchange Format Version 2 0 0" Recommended ANSI Standard EIA-548, Electronic Industries Association, Washington D.C., 1988.

8. D. R. Coelho, "VHDL: A Call for Standards," Proceedings of the 25th Design Automation Conference, 1988, pp. 40-47.

9. J. K. Ousterhout, "A Switch-Level Timing Verifier for Digital MOS VLSI" IEEE Transactions on Computer Aided Design, Vol CAD-4, No. 3, July 1985, pp. 336-349.

10. V. Ashok, R. Costello, P. Sadayappan, "Modeling Switch-Level Simulation Using Data Flow," Proceedings of the 22nd Design Automation Conference, 1985, pp. 637-644.

11. Z. Wang, P. Maurer, "Scheduling High-Level Blocks for Functional Simulation," Proceedings of the 26th Design Automation Conference, 1989.

12. F. Catthoor, J. van Sas, L. Inze, H. de Man, "A Testability Strategy for Multiprocessor Architecture," IEEE Design & Test of Computers, Apr. 1989, pp. 18-34.

13. P. Maurer, "FHDL Tutorial," University of South Florida, Department of Computer Science and Engineering Technical Report Number CSE-88-00011, Tampa, FL 33620, 1988.

14. E. S. Lee, T. F. Fang, "A Mixed-Mode Analog-Digital Simulation Methodology for Full Custom Designs," Proceedings of the IEEE 1988 Custom Integrated Circuits Conference, pp. 3.5.1-3.5.4.