Using reverse circuit execution for efficient parallel simulation of logic circuits

Kalyan Perumalla*, Richard Fujimoto†
College of Computing, Georgia Tech, Atlanta, Georgia, USA

ABSTRACT

A novel technique called reverse circuit execution is presented as an efficient approach towards integrated parallel execution of multiple sequential circuit simulators. The unique aspect of this approach is that it does not require source code modifications to either simulation engines or circuit models, and hence holds appeal in situations where parallelism is desirable but without access to simulator and/or model source code (as in the case of commercial simulators with proprietary code concerns). First, algorithms and methodology are presented for transforming an input circuit into another equivalent circuit that is capable of both forward and reverse execution. Following that, it is shown how the transformed circuit can be used towards optimistic synchronization of multiple circuit simulators. As an end result of using our approach, it will be possible to efficiently co-simulate logic circuits partitioned across multiple commercial simulators, by synchronizing their execution using optimistic concurrency protocols.

Keywords: Circuit simulation, parallel execution, reverse execution

1. INTRODUCTION

Simulation is used extensively in the circuit manufacturing industry to verify circuit correctness and to obtain detailed circuit timing information before an expensive and time-consuming fabrication is performed. Several commercial simulation packages are available in the market for circuit simulation. Most commercially available simulators are limited to sequential execution, i.e., each simulation can run on only one workstation. However, sequential execution can limit the sizes of circuits that can be simulated within the resource constraints of one workstation, and the speed of simulation can become unsatisfactory as circuit size increases. Also, many simulators cannot interoperate with other simulators, due to incompatibilities in models and/or simulation engines. A solution to many of these problems – capacity limitations, reduced simulation speed and interoperability concerns – is in adding an efficient parallel/distributed execution capability to the simulators. Parallel/distributed execution capability will allow multiple instances of the same (or different heterogeneous) circuit simulators to execute on multiple workstations to efficiently simulate different portions of a single circuit, and also will enable efficient integration of digital circuit models with analog circuit models in mixed-signal simulations[1, 6].

While parallel execution capability is desirable, it is important to realize such capability without requiring modifications to the existing sequential simulators or models. This is because it is either difficult or impossible to obtain and/or modify the source code for the simulator or the circuit models due to proprietary nature of the packages. Furthermore, it is desirable to avoid source code changes to the simulators and their models, in order to preserve the validation properties of the existing models.

Here, we present a novel and efficient approach towards parallel execution of existing sequential circuit simulators without the need for modifying their underlying simulation engines and circuit models. We present a new technique called *reverse circuit execution*, and show how it can be used to efficiently execute logic circuit simulators in parallel. This technique can be used to parallelize sequential circuit simulators by integrating replicated instances of the same simulator. Further, it is also useful in efficiently integrating instances of multiple heterogeneous sequential circuit simulators.

We first present the framework and methodology for the generation of inverse circuits from sequential circuits. Using that as building block, we present a technique for transforming the input circuit into another equivalent circuit that is

* kalyan@cc.gatech.edu; phone 1 404 385-0596; fax 1 404 385-2295; http://www.cc.gatech.edu/~kalyan;
† fujimoto@cc.gatech.edu; phone 1 404 894-5615; fax 1 404 385-2295; http://www.cc.gatech.edu/~fujimoto;
College of Computing, Georgia Tech, 801 Atlantic Dr NW, Atlanta, GA, USA 30332-0280

capable of both forward and reverse execution. The equivalent circuit is then driven forward during optimistic execution, and later driven in reverse direction, as appropriate, whenever portions of the optimistic execution need to be rolled back to undo incorrect execution. This scheme is independent of any particular simulator.

1.1 Background

Circuit simulators, in general, employ discrete event simulation techniques in which circuit behavior is modeled in terms of events executed in time-stamp order. Traditional methods[5, 9] towards parallelizing circuit simulators can be categorized into two distinct approaches – "conservative" and "optimistic".

In the conservative approach, the simulators execute in a tightly coupled fashion with respect to simulation time, ensuring that the simulators never violate global timestamp ordered execution[7]. However, the tight coupling can severely limit execution concurrency, and could potentially degenerate the integrated execution into a sequential one.

In the optimistic approach, individual simulators execute with looser coupling, but synchronize as needed by detecting and rolling back incorrect portions of their execution. The optimistic approach, thus, hold the potential to dynamically extract higher concurrency of execution, improving the overall execution time. However, a downside to traditional approaches to optimistic synchronization of circuit simulators has been that it requires modifications to the source code of the simulators and their circuit models, making it less appealing as a parallelization method for existing simulators, especially commercial simulators. Nevertheless, optimistic parallel execution can be appealing if the requirement for source modifications can be relaxed.

The novel *reverse circuit execution* technique presented here addresses the aforesaid problems associated with the traditional optimistic parallelization approach. The technique is designed as an efficient rollback mechanism in an optimistically synchronized parallel execution of the simulators. The key idea behind this technique is that the input circuit can be augmented to execute in both forward and reverse directions. The circuit can then be driven in forward direction during normal execution and driven in reverse direction during rollback. The original sequential simulators themselves are oblivious to the changes to the circuit, and execute as usual, just as they would execute any normal circuit sequentially. Both the simulator engine, as well as the circuit model source code, remains unmodified – modifications are only limited to the user input circuit.

Here, we present algorithms and the associated simulation methodology for automatically generating inverse circuits of a given circuit, and using that inverse circuit in conjunction with the original forward circuit in rollback-based optimistic synchronization. With this approach, multiple commercial simulators can be integrated in a co-simulation using optimistic protocols, such as Time Warp.

1.2 Optimistic simulation

Optimistic parallel simulation permits potentially incorrect computation to occur, but provides a mechanism to undo or roll back such computation after it has been discovered that the computation is in fact incorrect. The "computation" in discrete event simulation applications is viewed as a sequence of event executions, where each event execution may modify memory items, called the state, and may schedule new events (send messages) into the simulation's future. Hence, to rollback an event, it is sufficient to undo the changes to state variables and message sends caused by the event.

Usually the state variables are recovered by checkpointing the state or incrementally saving the state, a process referred to as state saving. The potential problem with state saving is that it can consume a large amount of memory when there are many state variables and/or a long history of state modifications that needs to be remembered (e.g. Analogy Inc.'s implementation of Calaveras algorithm for mixed-mode simulation[8]). An alternative rollback technique is reverse computation, which reduces memory requirements and greatly reduces the time to save state.

1.3 Rollback with reverse execution

In reverse computation, the model computation is so arranged that a one-to-one mapping exists between the current and previous state, such that the previous state can be recovered from the current state. This eliminates the need to save a state history (at the reduced cost of saving some "control" history). More detailed discussion about reverse computation can be found in Carothers, et al[4].

Reverse computation has not been previously explored as a technique for simulating digital logic circuits. In fact, the use of reverse computation for simulation in general has been limited. Recently, reverse computation has been applied to *communication network* simulations[4], by using a special compiler to generate reverse code from given model source code, and invoking the reverse code to restore state during rollback. Clearly, the source code modification approach requires access to the original source code of the simulator and/or the models. In this paper, we present a new reverse computation approach to digital circuit simulation. The focus of our work is in extending existing commercial simulation software to exploit reverse computation techniques in such a way that we do not require access to the source code of the simulator.

Our approach entails performing reverse computation at the circuit level, rather than simulation code level. Here, we focus on the applying this technique to gate level circuit simulation, and limit the discussion to correctly simulating the logic circuit's behavior.

In the context of reversible computing in general, and recently in relation to quantum computing, theoretical studies focused on mapping circuits made of irreversible logic gates to circuits made of reversible logic gates[2]. However, since we are constrained by existing circuit simulation models, we focus on making circuits reversible by using only conventional (irreversible) gates.

2. OPTIMISTIC PARALLEL CIRCUIT SIMULATION

We will first address the problem of making a given synchronous digital circuit reversible and also generating its inverse circuit. We then turn to the question of how to make combined use of the forward and reverse versions of the input circuit in an optimistic parallel simulation setting.

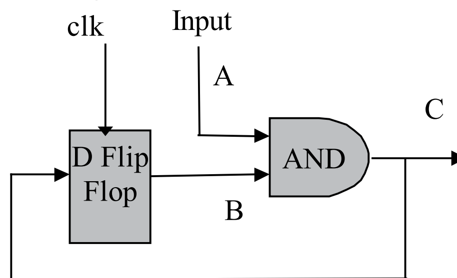

Figure 1: Example sequential circuit.

For illustration, consider the circuit fragment given in Figure 1, with an external input **A**, an internal input **B** from a flip-flop, and an output **C** that is also fed back to the flip-flop. It is a simple sequential circuit that always generates **0** if it receives at least one **0** in its input anytime in its past, but generates **1** until such time. Let us focus on how we can transform this circuit in such a way that it not only preserves its original circuit behavior (during forward execution), but additionally, the circuit can be made to execute in reverse direction as well. We can then use the resulting modified circuit in place of the original circuit, and drive it in forward or reverse direction, as appropriate, within an optimistic simulation run.

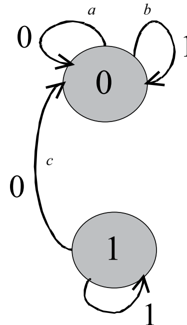

Figure 2: State diagram for circuit shown in Figure 1.

The state diagram for the circuit is given in Figure 2, and its equivalent truth table is given in Table 1. The states in the state diagram represent values of the flip-flop memory element of the sequential circuit, and the arcs are labeled with the external input bit value. An arc from state **CS** to state **NS** with label **In** represents the transition of the flip flop from current state **CS** to next state **NS** upon receiving **In** as input, after one clock cycle.

Table 1: Truth table for circuit in Figure 1.

CS	In	NS
0	**0**	**0**
0	**1**	**0**
1	**0**	**0**
1	**1**	**1**

The same information is encoded in the truth table in a different form: the current state **CS** in combination with the external input **In** constitute the input bits, and the next state **NS** forms the output bits in the truth table.

From the state diagram, it is clear that the circuit is not reversible as it stands. In other words, by only examining the current state, one cannot recover the previous state. This is because of ambiguity about to which state to transition back when the current state is **0**. Since the state **0** has multiple incoming edges for the same input value **0** from different sources (one from state **1**, and the other from state **0**), it is not possible to recover the previous state without additional information. This problem is more clearly evident in the truth table, in which more than one combination of **<CS, In>** input bits map to the same **<NS>** output bit value, making it difficult to uniquely recover the inputs given the output.

In general, a sequential circuit is not reversible if, in its state diagram, at least one state exists that has multiple incoming arcs with same input value but emanating from different source states. We can render such circuits reversible by augmenting the circuit to generate additional output bits that are necessary to uniquely recover the previous state from any given current state.

2.1 Augmenting input circuit for reversibility

In the example circuit, it is clear that two additional output bits are needed in order to accurately preserve information about any state transition (i.e., two extra bits are necessary and sufficient). These two additional bits, **E1** and **E2**, serve to record information about the most recent transition – the received input value (1 bit), and the source state (1 bit to resolve ambiguity among the two source states for transitions to the **0** state). The values assigned to the additional bits can be arbitrary, so long as the values satisfy the requirement of recovering the input and previous state uniquely. Note that we choose to preserve the original behavior of the circuit, and hence not alter its original output vector. This is to preserve the original characteristics of the input circuit unchanged for the benefit of the user. Our modifications only add additional gates and input/output signals to the otherwise intact input circuit.

In general, the minimum (optimal) number of output bits that are necessary to be added is given by $\lceil log_2 F \rceil$, where F is the maximum among all ambiguous in-degrees of states in the circuit state diagram. An algorithm is given next to find the number and values of bits to be added. Using this algorithm, the state diagram and consequently the original circuit can be modified to make the circuit reversible.

2.2 Reversible truth table generation

We now present a simple algorithm that takes a truth table as input and finds the optimal number of additional bits needed to make a given truth table reversible. It also generates a correct assignment of values to the added bits, thus generating a complete reversible version of the original input truth table. The reverse truth table is then trivially obtained by swapping the input and output vectors of the augmented forward truth table.

This algorithm is designed to be applied on the truth table of the original input circuit. The resulting truth table then gives the minimum number of extra bits needed to make the circuit reversible, and also assigns a unique combination of values to the added bits.

In general, given any truth table with n input bits and m output bits, it is necessary to analyze the truth table to check if more than one input vector maps to the same output vector. If so, the output vector to which the maximum number, Q, of input vectors map determines the number of bits q to be added to the output vector. The q bits added to the output vector are called *free* bits. These bits will be assigned unique values to distinguish among the input vectors incident on any single output vector. The resulting truth table then becomes reversible.

In the best case, no output vector is mapped to more than one input vector, in which case the truth table is already reversible, and hence no additional bits are needed. In the worst case, all input vectors map to exactly one output vector, in which case, $q = \lceil \log_2 n \rfloor$ bits need to be added to the output vector.

An algorithm is given shown in Table 2, which is based on the preceding observations to transform any truth table into an equivalent, but reversible, truth table. The input and output vectors of the original truth table are preserved in its reversible truth table. Additionally, the reversible truth table possesses the capability of recovering the input vector of the original truth table, given only the output vector of to the reversible truth table.

Table 2: Algorithm for generating a reversible truth table from an irreversible truth table.

Input: Original truth table with n inputs & m outputs
Output: Reversible truth table with n inputs, $m+q$ outputs, where $q = \lceil \log_2 Q \rfloor$, and Q is the maximum among in-degrees of all output vector values in the original truth table.

1. Expand all *don't care* values in the input vector of truth table.
2. Find maximum in-degree Q of output vector values.
3. Add $q = \lceil \log_2 Q \rfloor$ free bits to the output vectors of truth table, and make them all *don't care* values.
4. For each output vector state with in-degree q' ($0 <= q' <= q$):

    Distinguish the input states by assigning q' distinct values to the first q' bits among the (*don't care*) free bits in the output vector.

The resulting truth table is used in a straightforward way to generate the reverse truth table: Swapping the input vectors with their corresponding output vectors of the truth table results in its reverse truth table.

2.3 Augmented the reversible circuit

By applying the reversible truth table generation algorithm (Table 2) on the truth table of the original circuit (Table 1), we obtain the reversible truth table given in Table 3. The corresponding reversible state diagram is shown in Figure 3. The original circuit is thus enhanced to generate two additional outputs **E1** and **E2**. Note that these two bits encode sufficient information to not only recover the previous state from the most recent transition, but also recover the input to the most recent transition.

Table 3: Truth table of example circuit, made reversible by augmenting with two additional output bits, E1 and E2.

CS	**In**	**NS**	**Extra**	
			E1	**E2**
0	**0**	**0**	**0**	**0**
0	**1**	**0**	**0**	**1**
1	**0**	**0**	**1**	**X**
1	**1**	**1**	**X**	**X**

The corresponding state diagram augmented to generate the two output values for every transition is shown in Figure 3. To realize this reversibility at the circuit level, an auxiliary circuit is added to the original circuit to generate the specified pattern of values to **E1** and **E2**. In this example, the assignment is designed such that the auxiliary circuit is a trivial "pass through" of the flip-flop and external input values. This simple assignment also has the advantage that no additional circuitry is introduced for the production of these extra bits, thus preventing the addition of (parallel) simulation overhead, when compared to the original circuit. In general, such simple assignment may not be possible, and hence might involve the generation of a non-trivial auxiliary circuit to generate the extra output bits. Standard Boolean circuit generation algorithms can be applied to obtain the auxiliary circuit corresponding to the truth table of the extra bits.

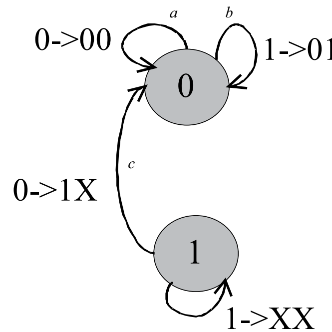

Figure 3: State diagram of example circuit, made reversible by adding two extra output bits.

The reversible version of the original circuit augmented with the additional output bits is shown in Figure 4.

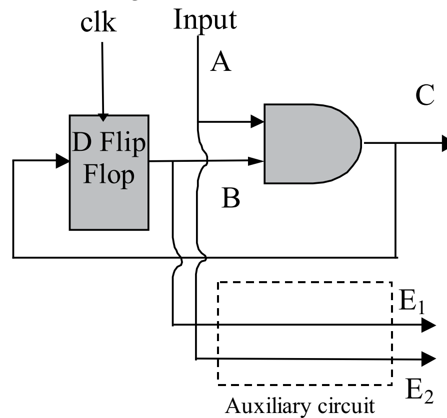

Figure 4: Original example circuit made reversible using an auxiliary circuit and adding outputs E1 & E2.

Although in this case, the number of bits required for reverse execution is the same as the number of bits for state savings, it is in general not true. The number of bits added for reverse execution is only on the order of *log(k)*, where *k* is the maximum in-degree of states in the state diagram.

2.4 Generating reverse circuit

Since we now have a reversible forward circuit, we can now generate the reverse circuit that can be used to undo (rollback) the execution of the original circuit. We will later use this reverse circuit in conjunction with the augmented reversible circuit to design a combined circuit that can be driven in forward and reverse directions at will.

Table 4: Truth table for inverse circuit of example circuit.

CS	E1	E2	NS	In
0	0	0	0	0
0	0	1	0	1
0	1	X	1	0
1	X	X	1	1

The truth table for the reverse circuit is trivially obtained by swapping the input and output vectors of the augmented reversible circuit, as described in the reverse truth table generation algorithm. Using this approach, the truth table for the inverse circuit of our example circuit is shown in Table 4. Using the reverse truth table, standard Boolean circuit generation algorithms can be used to generate an equivalent circuit, which will constitute our desired reverse circuit.

2.5 Combined forward and reverse circuits

Using the reversible and reverse circuit, we are now ready to assemble the complete circuit that can be made to simulate our original circuit in either forward or reverse directions. First, a new signal **F/R** is added that will be used to choose between forward and reverse execution modes for the circuit. Next, a stack circuit is added to buffer the output vectors generated by the auxiliary forward circuit. This last-in-first-out output stack serves to feed the reverse circuit with the most recently generated values in correct reverse order during rollback. A separate first-in-first-out stack is added to buffer those inputs that were consumed by the original circuit during optimistic execution, but undone and recovered by the reverse circuit. This input stack serves to buffer and feed (to the original circuit) those past input values that were accepted ahead of time during optimistic forward processing.

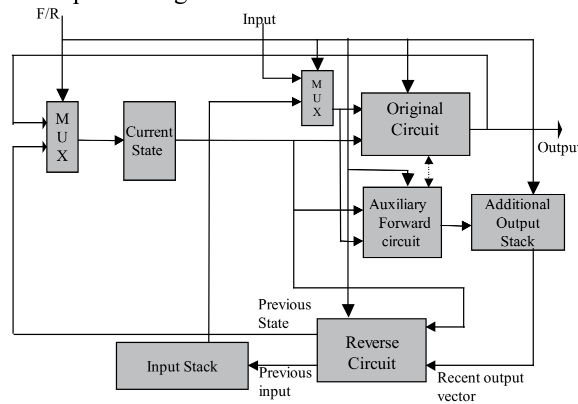

Figure 5: Generalized circuit architecture for executing original circuit in both forward and reverse directions.

A multiplexer is introduced to choose between values from the input stack and actual external input – external input is admitted through the multiplexer only when the input stack becomes empty. Another multiplexer is inserted before the flip-flop (current state) vector to choose between the outputs generated by the original forward circuit or the output from the reverse circuit. This multiplexer is driven by the **F/R** signal, which makes it choose the forward output during forward processing mode, and reverse circuit output during rollback processing mode.

2.6 Driving the combined circuit

During optimistic simulation, initially, the circuit is executed forward, and hence the **F/R** signal is turned on (representing forward mode). When a signal arrives from another simulator with a timestamp less than the current simulation time of the circuit, the **F/R** signal is turned off and the (reverse) circuit is executed in order to roll back the portion of forward computation that was done optimistically. After rollback is completed, the **F/R** signal is turned back on, and normal execution is resumed. For proper functioning of this scheme, the timing of reverse circuit needs to be carefully designed using only delta-delay processing to keep the native simulator time in synchrony with parallel optimistic simulation system time.

2.7 Inter-simulator synchronization

For parallel simulation of a large circuit, the circuit can be partitioned into parts, with each part simulated in a different simulator, with one simulator per processor. Signals spanning across simulators can be transferred as time stamped events exchanged among the simulators. For example, in Cadence Verilog-XL simulations, the Verilog Procedural Interface can be used to realize inter-simulator synchronization (e.g., see [3, 10] for more details).

3. OPEN ISSUES

3.1 Automation and usability

Clearly, for production use, the reverse circuit execution approach requires automation of some of its component steps, such as automatic circuit augmentation, and automatic reverse circuit generation. Additionally, simulator-dependent pre-processors are required to translate circuit configuration scripts to transform input circuits to augmented reversible circuits. While this creates additional development burden introduced by this approach, we don't see any fundamental technical difficulties for such automation in a production setting.

3.2 Simulator clock vs. optimistic time

Since the simulator executes both forward and reverse modes of the augmented circuit as though it was executing normal forward mode of the augmented circuit, the simulator's clock will be ahead of the actual simulation time due to rollback. One way to address this problem is to use "infinitely fast" reverse execution. In other words, very small timestamp increments can be used for all rollback events (similar to delta delay processing). However, such operation could potentially be simulator-dependent, and hence this issue needs to be more completely resolved before production use.

3.3 Intermediate circuit stack sizes

Since the output bit stack sizes are finite and set at initialization (circuit configuration) time, they need to be carefully chosen to achieve maximum simulation runtime efficiency. The larger the stack size, the farther in simulation time an individual simulator can optimistically process the circuit. However, large stack sizes increase the size of the simulated circuit, and hence can add to the simulation overhead. Smaller stack sizes lower the simulation overhead, but can limit the amount of optimistic processing, and consequently the amount of concurrency.

3.4 Minimizing the augmentation overhead

In the augmentation method for making the input circuit reversible (Section 2.3), it is important to minimize the size and complexity of added circuitry. Thus, it is desirable to find the minimum number of circuit elements that need to be added to obtain the auxiliary bits added to the output bit vector. Doing so will minimize the amount of additional simulation work that the simulator needs to execute during forward execution of the augmented circuit. The minimization procedure should consider not only the minimum sized independent circuit that obtains the required bits for reversal, but also the potential reuse of some of sub-circuits of the original input circuit in generating the reversal bits. This remains to be addressed.

3.5 Rolling back statistics

Here, we have addressed the problem of correctly synchronizing circuit behaviors. However, additional work is needed to consider the effect of reverse circuit execution on circuit statistics models and correctly update the statistics during rollback and optimistic processing.

4. CONCLUSIONS

We presented a novel approach called reverse circuit execution for efficient parallel optimistic execution of multiple circuit simulators. Towards applying this approach, algorithms and methodology have been presented for the generation of inverse circuits for sequential and combinational circuits. Building on that, a technique has been described for transforming an input circuit into another equivalent circuit that is capable of both forward and reverse execution. It has been then shown how the equivalent circuit can be used in a simulator-independent fashion to efficiently synchronize optimistic parallel execution of multiple simulators. The circuit is driven forward during optimistic execution, and later driven in reverse direction, as needed, whenever portions of the optimistic execution need to be rolled back to undo incorrect execution.

Using the approach described here, it will be possible to efficiently interoperate multiple commercial simulators, and synchronize their execution using optimistic protocols, such as Time Warp. This approach has the potential for automatically extracting the inherent parallelism in a composition of circuits simulated across multiple simulators. The novel aspect of our approach is that access is not required to the source code of the simulator or the model. This has been a major challenge because traditional approaches to parallelization typically require access to source code of either the simulator or model or both.

ACKNOWLEDGEMENTS

This work has been funded in part under the Yamacraw mission of the State of Georgia. Liang Xiao contributed to discussions while he was working with the authors as a graduate research assistant at Georgia Tech.

REFERENCES

1. Fujimoto, R.M., Perumalla, K.S., Xiao, L., Casinovi, G., Swaminathan, M., Dalmia, S. and Mao, J. Parallel Simulation Backplanes for Mixed Signal Circuit Design, Technical Report Yamacraw Research Center at Georgia Tech, Atlanta, 2000.
2. Rieffel, E. and Polak, W. An Introduction to Quantum Computing for Non-Physicists. *ACM Computing Surveys*, *32* (3). 300-335(2000).
3. Fujimoto, R.M., Perumalla, K.S. and Xiao, L. Implementing Parallel Verilog-XL Simulation, Technical Report Yamacraw Research Center at Georgia Tech, Atlanta, 2000.
4. Carothers, C., Perumalla, K.S. and Fujimoto, R.M. Efficient Optimistic Parallel Simulations using Reverse Computation. *ACM Transactions on Modeling and Computer Simulation*, *9* (3)(1999).
5. Frey, P., Parallel Synchronization of Continuous Time Discrete Event Simulators. *International Conference on Parallel Processing* (1997), 227-231.
6. Shmerler, S., Tanurhan, Y. and Muller-Glaser, K.D., A Backplane for Mixed-Mode Cosimulation. (1995), 499-504.
7. Buck, J.T., Ha, S., Lee, E.A. and Messerschmitt, D.G. Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems. *International Journal of Computer Simulation*, *4* (Special Issue on Simulation Software Development). 155-182(1994).
8. Analogy Inc. *Mixed-Signal Simulation Handbook*. Web book, http://www.analogy.com/Pubs/Guide/toc.html, 2002.
9. Fujimoto, R.M. *Parallel and Distributed Simulation Systems*. Wiley Inter-science, 2000.
10. Sutherland, S. *The Verilog PLI Handbook: A User's Guide and Comprehensive Reference on the Verilog Programming Language Interface*. Kluwer Academic Publishers, 1999.