Hybrid Analysis of SystemC Models for Fast and Accurate Parallel Simulation

Tim Schmidt, Guantao Liu, and Rainer Dömer
Center for Embedded and Cyber-physical Systems
University of California, Irvine, USA

Abstract— Parallel SystemC approaches expect a thread-safe and race-condition-free model from the designer or use a compiler which identifies the race conditions. However, they have strong limitations for real world examples. Two major obstacles remain: a) all the source code must be available and b) the entire design must be statically analyzable. In this paper, we propose a solution for a fast and fully accurate parallel SystemC simulation which overcomes these two obstacles a) and b). We propose a hybrid approach which includes both static and dynamic analysis of the design model. We also handle library calls in the compiler analysis where the source code of the library functions is not available. Our experiments demonstrate a 100% accurate execution and a speedup of 6.39x for a Network-on-Chip particle simulator.

I. Introduction

The increasing complexity of embedded systems slows down the design process of new products. Designers use simulation as a tool to validate prototypes. However, the dramatically increasing simulation time has been identified as a bottleneck in the design process. Various approaches have been made to optimize the simulation performance. For instance, the simulation reduced level has been decreased and communication has become more abstract. Although state-of-the-art PCs have multi-core processors, most simulations are still executing sequentially.

SystemC [1] is a widely-used tool for simulating and modeling embedded systems. We advocate an advanced approach to simulate SystemC models fully in parallel without losing accuracy. This is in contrast to other limited techniques. For instance, techniques like time decoupling have been proposed to boost the simulation performance. However, this method results in inaccurate simulation results [2].

Our Recoding Infrastructure for SystemC (RISC) compiler infrastructure analyzes a given design, identifies potential race conditions, and transforms the sequentially written model into a parallel executable design. The transformation happens automatically and the designer has no burden of partitioning the model. We propose a hybrid analysis to consider all possible aspects of the design including 3rd party libraries.

A. Problem Definition

SystemC is the de facto standard library for modeling and simulating embedded systems. The official simulation kernel performs the simulation in a sequential fashion. In other words, only one simulation thread is active at any time. If several threads could be simulated in parallel rather than sequentially, then we would obtain a significant decrease in the simulation time.

First attempts have been made to run the simulation in a parallel fashion. A dedicated compiler is used in [3] to analyze the design hierarchy and to identify the potential race conditions among the individual threads. However, in order to perform these advanced simulation techniques, two criteria must be satisfied. First, the entire source code must be available for the static analysis. It cannot be partially provided in a library. Second, the design must be statically explorable to identify the design hierarchy.

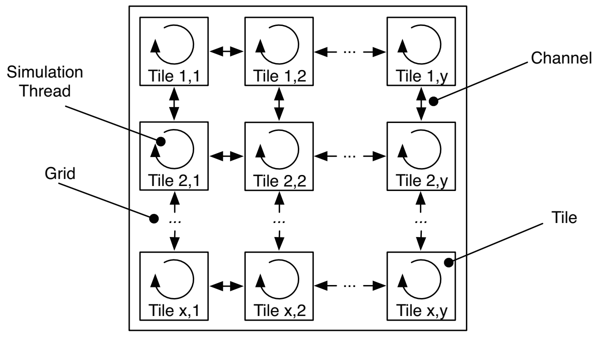

Fig. 1. General structure of a Network-on-Chip design model

The Network-on-Chip (NoC) paradigm is a widely-used design pattern. However, it violates these parallel simulation requirements. Figure 1 shows a typical NoC which is assembled through a hosting grid and tiles which are organized in rows and columns. A tile contains intellectual property (IP) and communicates with other tiles located to the north, west, south, and east of it. A tile can be a user-defined or a 3rd party IP element which is encapsulated in a library. During the design space exploration, architects test various combinations of these grid sizes and tile components. The number of rows and columns is often defined through command line parameters. Thus, the tiles are allocated in two loops with the `new` operator which is not statically analyzable. Evidently, these two limitations prevent the analysis for an efficient parallel simulation.

In this paper, we propose a solution in order to simulate designs in an accurate parallel fashion. These models can include 3rd party libraries and code which is not statically analyzable. In the new design flow, we perform a dynamic analysis which is followed by a static analysis. Additionally, we provide the designer with the opportunity to annotate 3rd party libraries. Such annotations are taken into account for the advanced design analysis and allow the simulation of library code in a parallel fashion.

B. Related Work

Parallel simulation of discrete event simulation is a well-studied subject. Initial work has been contributed by [4].

The concept of a Segment Graph for parallel simulation was first introduced in [3] for synchronous and out-of-order parallel simulation. Later, the Segment Graph infrastructure was used for may-happen-in-parallel analysis for safe ESL models in [5]. Both contributions require a complete design model with a statically analyzable module hierarchy. In contrast, our approach supports 3rd party libraries and non-statically analyzable module hierarchies in designs.

Time decoupling is a widely-used method that speeds up the simulation of SystemC models. Parts of the model execute in an unsynchronized manner for a user defined time quantum. However, this strategy is associated with inaccurate simulation results [2]. [6] and [7] propose a technique to parallelize time-decoupled designs. This technique requires the designer to manually partition and instrument the model in parts of time-decoupled zones. In contrast, our approach supports a 100% accurate simulation of designs. Also, our compiler automatically instruments the model.

A tool flow for parallel simulation of SystemC RTL models was proposed in [8]. The model was partitioned according to a modified version of the Chandy–Misra–Bryant algorithm [9]. In contrast, our conflict analysis considers the individual statements of threads. Also, our solution is not restricted to only RTL models.

An approach of static analysis for simulation of SystemC models on GPUs was provided in [10]. In contrast, our approach combines static and dynamic analysis to obtain information for the conflict analysis which is only available at run time.

II. Hybrid Analysis

The transition from a sequential simulation towards a parallel simulation is an extensive process. The design must be analyzed and prepared for potential race conditions to avoid unpredictable data corruption. This requires a full understanding of the module hierarchy. One option is to statically extract the module hierarchy and analyze the individual threads. However, in most cases not all of the information can be explored statically. For instance, during the design space exploration, designers test various prototypes to explore an optimal design. Design parameters are passed via the command line to define the number of modules, channels characteristics, and other needed information. The instances of modules, channels, and ports are created through loops in a dynamic fashion. However, the essential parameters are only available at run time, so they cannot be statically analyzed. The result would be an incorrect model transformation which produces wrong simulation results.

Figure 2 shows our proposed design flow which supports command-line parameters and the dynamic analysis for a fully accurate parallel simulation. Essentially, the tool flow is split up into two major stages. The first stage performs a *Dynamic Design Analysis* and collects the design hierarchy. The second stage allows the *Static Conflict Analysis* to integrate the obtained data from the previous stage. As a result, the design is transformed into a thread-safe design for fast parallel execution.

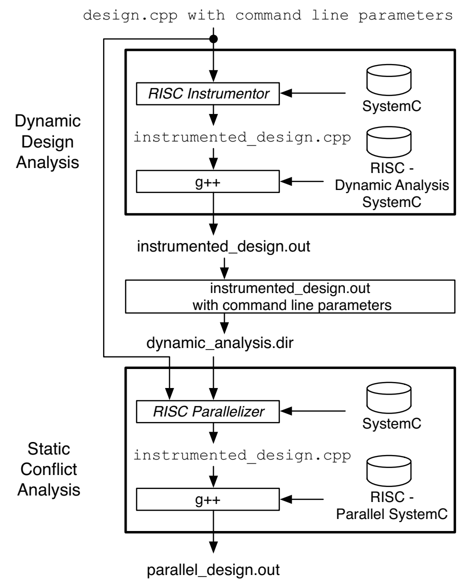

Fig. 2. Tool flow for the proposed hybrid analysis of design models

A. Dynamic Design Analysis

The purpose of the *Dynamic Design Analysis* is to provide data for the *Static Conflict Analysis* which can only be obtained at run time. This data is structural design information which, for example, is dependent on command-line parameters, or hidden deeply in nested loops. The results of this step are provided in a dynamic internal representation (DIR) file and then used as a lookup table in the *Static Conflict Analysis*.

In detail, the DIR file includes the module hierarchy, the port mapping, and the mapped variables of references. We represent this information as an abstract tree. Specifically, we store the declaration name and the address for all declarations. In addition, we store the address of the bounded channel for ports and the typename of the channel for the channels. Listing 1 illustrates such a DIR file. Undoubtedly, we can identify that the reference `ref` is mapped to the variable `var`. Also, we can identify that the port `port` is bound to the channel `chnl1` of the type `MyChannelType`. We can use the address as a lookup because references and ports are only bound once. During the simulation, the specific address might be different. However, the lookup will lead to the same variable.

Listing 1 Example of a DIR file

```
top:0x111(
  chnl1:0x222:MyChannelType, var:0x333
  mod1:0x444(var:0x555,ref:0x333,port:0x222))
```

We can partially analyze a design in three different ways, but each has its own limitations: A) The C++ language has limited support of introspection. It is possible to identify the type name of a variable at run time. However, it is not possible to identify the declaration name of a variable. B) The SystemC library has an interface for the introspection of a design. It is possible to identify all top modules at run time, and then traverse all the elements which are derived from `sc_object` in a hierarchical fashion. However, it is not possible to identify plain old data types (e.g. `float` and `int`) or their respective declaration name. C) A static analysis of the source code allows to identify modules and analyze all of their members. However, the module hierarchy can only be explored with a severely limiting modeling guide lines.

In our approach, we perform a combined solution of A, B, and, C to generate a DIR file where the designer has no modeling limitations. First, the RISC instrumentor reads the file *design.cpp* and analyzes all the design elements statically. In other words, we have access to the declarations and their names. For each module, we instrument functions to print variables, ports, and other information. For instance, in the function `void print_vars() {fprintf("var:%p",&var); ...}`, we use the `typeid` support of C++ to obtain the type name at run time. Through that process, we are able to generate the source code for the instrumented design and build an executable.

Second, we modify the simulation kernel to allow it to undergo the *Dynamic Design Analysis*. The simulation of a SystemC model is split into two major phases. In the first phase, the *elaboration* starts where the module hierarchy and the port binding are established. In the second phase, the *simulation* of the design is performed. The SystemC API provides a hook between these two steps. We simulate the design with all command line parameters until the *elaboration* is completed. Finally, we traverse the module hierarchy via the SystemC introspection API and call the functions for the variable printing.

B. Static Conflict Analysis

The *Static Conflict Analysis* identifies potential race conditions between the individual threads. In detail, we partition each thread into so-called segments. A segment considers all potentially executed statements between two scheduling steps. A new scheduling step is triggered with a `wait()` function call which gives control back to the simulation kernel. Figure 4 shows a segment graph of the source code in Figure 3.

```
0  void foo() {
1    r++;
2    wait();
3    a=b+c;
4    if(condition){
5      i++;
6      wait();
7      j++;
8    } else {
9      b=x+y;
10   }
11   z=z*z;
12   wait()
13   y=z+4; }
```

Fig. 3. Example Source Code [11]

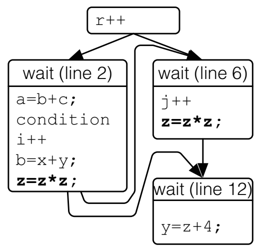

Fig. 4. Segment Graph [11]

Algorithm 1 formally defines the central function BuildSG [11]. Function BuildSG is recursive and builds the graph of segments by traversing the AST of the design model. Here, the first parameter *CurrStmt* is the current statement which is processed next. The set *CurrSegs* contains the current set of segments that leads to *CurrStmt* and thus will incorporate the current statement. For instance, while processing the assignment `z=z*z` in Figure 4, *CurrSegs* is the set {wait(line2), wait(line6)}, so the expression will be added to both segments.

If *CurrStmt* is a boundary statement (e.g. `wait`), a new segment is added to *CurrSegs* with the corresponding transition edges (lines 2 to 7 in Algorithm 1). Compound statements are processed by recursively iterating over the enclosed statements (lines 8 to 12) while conditional statements are processed recursively for each possible flow of control (from line 13). For example, the `break` and `continue` statements represent an unconditional jump in the program. For handling these keywords, the segments in *CurrSegs* move into the associated set *BreakSegs* or *ContSegs*, respectively. After completing the corresponding loop or `switch` statement, the segments in *BreakSegs* or *ContSegs* move back to the *CurrSegs* set.

For brevity, we illustrate the processing of function calls and loops in Figure 5 and Figure 6. The analysis of function calls is shown in Figure 5. In step 1, the dashed circle represents the segment set *CurrSegs*. The RISC algorithm detects the function call expression and checks if the function is already analyzed. If the function is encountered for the first time, the function is analyzed separately, as shown in step 2. Otherwise, the algorithm reuses the cached SG for the particular function. Then,

Algorithm 1 Segment Graph Generation

```
function BuildSG(CurrStmt, CurrSegs, BreakSegs, ContSegs)
    if isBoundary(CurrStmt) then
        NewSeg ← new segment
        for Seg ∈ CurrSegs do
            AddEdge(Seg, NewSeg)
        end for
        return CurrSegs ∪ { NewSeg }
    else if isCompoundStmt(CurrStmt) then
        for Stmt ∈ CurrStmt do
            CurrSegs ← BuildSG(Stmt, CurrSegs, BreakSegs, ContSegs)
        end for
        return CurrSegs
    else if isIfStmt(CurrStmt) then
        AddExpression(IfCondition, CurrSegs);
        NewSegSet1 ← BuildSG(IfBody, CurrSegs, BreakSegs, ContSegs)
        NewSegSet2 ← BuildSg(ElseBody, CurrSegs, BreakSegs, ContSegs)
        return NewSegSet1 ∪ NewSegSet2
    else if isBreakStmt(CurrStmt) then
        BreakSegs ← BreakSegs ∪ CurrSegs
        CurrSegs ← ∅
        return CurrSegs
    else if isContinueStmt(CurrStmt) then
        ContSegs ← ContSegs ∪ CurrSegs
        CurrSegs ← ∅
        return CurrSegs
    else if isExpression(CurrStmt) then
        if isFunctionCall(CurrStmt) then
            return AddFunctionCall(CurrStmt, CurrSegs)    ▷ See Figure 5
        else
            AddExpression(CurrStmt, CurrSegs)
            return CurrSegs
        end if
    else if isLoop(CurrStmt) then
        return AddLoop(CurrStmt, CurrSegs)    ▷ See Figure 6
    end if
end function
```

in step 3, each expression in segment 1 of the function is joined with each individual segment in *CurrSegs* (set 0). Finally, segments 4 and 5 represent the new set *CurrSegs*.

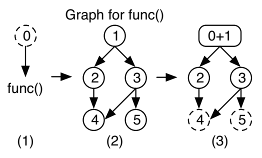

Fig. 5. Function call processing [11]

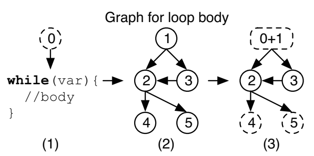

Fig. 6. Loop processing [11]

Correspondingly, Figure 6 illustrates the SG analysis for a while loop. Again, the dashed circle in step 1 represents the incoming set *CurrSegs*. The algorithm detects the `while` statement and analyzes the loop body separately. The graph for the body of the loop is shown in step 2. Then each expression in segment 1 is joined into the segment set 0 and the new set *CurrSegs* becomes the joined set of 0+1, 4, and 5. Note that we have to consider set 0+1 for the case that the loop is not taken.

Next, we build a conflict table which identifies the potential race conditions between all individual segments in the design. For each segment, we identify which shared variables are read and written. If we access a reference, we perform a lookup through the *Dynamic Design Analysis*. If a channel call happens, we use the *Dynamic Design Analysis* lookup to get the channel and the corresponding called function.

III. Library Handling

The simulator requires the following two bits of information in order to run in a parallel manner. First, it needs to know the race conditions of each individual thread before the simulation starts. Second, the simulator needs to know the current segment ID of each thread. Before the simulator triggers the next segment of a thread, it is essential to check for race conditions with all other active segments. These are two main obstacles for real world SystemC designs which include standard and 3rd party libraries.

The first issue is that the *Static Conflict Analysis* needs access to the function bodies to identify the race conditions for the individual threads. However, 3rd party intellectual property (IP) is often shipped through libraries. The designer has access to the function signatures, but the implementation is hidden in the library file. In other words, our static analysis cannot identify potential `wait()` calls and race conditions. Consequently, a segment which calls a library function is set in conflict with all other segments. For example, inherent function calls like `printf()` and `sqrt()` would sequentialize the parallel simulation. Therefore, we provide a function annotation scheme to include information about these library functions for the static analysis.

The second issue is that the parallel simulator needs to know which segments are ready to execute. One previously employed strategy was to statically instrument each individual `wait()` call with the associated segment ID, e.g., `wait(event,42);`. The simulator obtains the upcoming segment ID through the `wait()` call. However, this strategy cannot be used for designs with 3rd party libraries. The RISC Parallelizer cannot instrument library files. Instead, we provide a modified RISC simulation kernel to pass the segment ID through the 3rd party library to the simulation kernel.

A. Function Annotation

We present an annotation scheme for function declarations to provide information for the conflict analysis. Thus, the user can annotate via `pragma` statements two different pieces of information, namely, the conflict status and the type of `wait()` function calls in a function body. We consider a function as conflict-free if the corresponding function body has no read/write access conflicts on any shared state with the other threads in the simulation model. A segment which calls a function with the annotation of *non-conflict-free* results in a conflict with all other segments. The simulator executes this segment sequentially and safeguards a fully accurate simulation.

Figure 7 shows four different options of annotating a function declaration with wait information. We designed the scheme to have full support of the SystemC built-in library channels. In the first case, the function has no wait statement. This is the option for non-blocking function calls. The next two cases cover the situation that the function has a conditional or a non-conditional wait, for instance an `sc_buffer`. The last case portrays a more complicated channel type, an `sc_fifo`, where the `wait()` call is in a loop.

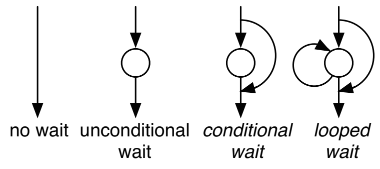

Fig. 7. Wait annotation for function declarations

The user also has the option not to provide any annotation for library functions. In this case, we assume that the function is conflict-free and there are no wait statements in the function. This behavior is expected for the Standard C Library and Standard Template Library. We decided to make this the default mechanism to avoid annotating all standard library functions.

B. Segment ID Passing

The parallel simulator needs the active segment ID of each individual thread during the simulation. The approach of instrumenting `wait()` calls with an additional segment ID parameter is only possible if the source code is available for all parts of the design. In other words, we cannot instrument `wait()` calls which occur in the library.

Figure 8 illustrates our generalized solution for support of libraries. The thread carries the upcoming segment ID from the user domain to the parallel RISC SystemC library. We instrument the function call `setID(42)` before the function call in the library domain happens. In the RISC SystemC kernel, we get the segment id via `getID()`. This solution provides the benefit that any 3rd party channel can be used without any modification.

IV. Experiments

A. Producer Consumer Example

We implemented the consumer producer example which is illustrated in Figure 8. Both the sender and receiver crunch numbers and exchange them. The number of sender and receiver pairs is scalable via the command line.

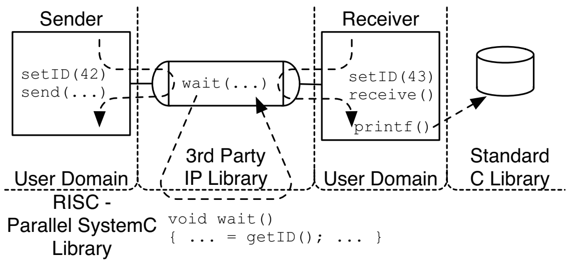

Fig. 8. Different domains of a design

The regular RISC Parallelizer — without the *Dynamic Design Analysis* capabilities — cannot process this design and returns an error message. The *Dynamic Design Analysis* creates the correct module hierarchy and identifies the number of modules which is double the number of communication pairs. The obtained speedup depends on the number of modules and the ratio of communication and number crunching.

B. Network-on-Chip Particle Simulator

We selected as a comprehensive example a Network-on-Chip (NoC) particle simulator to demonstrate the parallel simulation capabilities of our hybrid analysis. The abstract architecture of the particle simulator is illustrated in Figure 1. The grid is assembled of tiles where each tile communicates bidirectionally with a tile to its north, south, east, or west. For a 8x8 example, we have 64 tiles and one grid module. Each tile has one thread which computes the motion of the individual particles in a certain area of the model. These particles move continuously in 2D space. The moment when a particle crosses the boundary, the responsibility of computing and updating the position of the particles shifts from one tile to its neighbor tile. The entire design can be scaled up to any quadratic size. The user can define via the command line the number of tiles as well as the number of particles, the gravity, and other options. The individual tiles including ports and channels are dynamically created.

At the beginning of the simulation, the grid sends the initial particles to each individual tile. This happens in a purely sequential fashion. Next, all tiles simulate the particles and synchronize with their neighbors. A tile can be blocked due to its communication with one of its neighbors. At the end of the simulation, all the tiles send the particles back to the grid.

B.1 Static Design Analysis

The static analysis of the particle simulator causes an error message: ”Error: Array of modules in line 231”. The RISC Parallelizer detects an array of pointers for tile modules. However, it cannot identify how many instances are created. The same applies to the array of ports and channels. The module hierarchy cannot correctly be extracted and the RISC Parallelizer does not create an executable.

B.2 Dynamic Design Analysis

The dynamic analysis is performed for different grid sizes. For the 8x8 particle simulator, 65 modules and 176 channels are correctly identified. The parallel simulation creates the same results as the traditional sequential discrete event simulation. In other words, the parallel simulation has the same accuracy as the sequential simulation.

TABLE I
Simulation Speedup of the Particle Simulator

	Speedup				Time (in sec.)	
Particles	10k	20k	40k	60k	seq. 60k	par. 60k
5x5	2.56x	3.58x	3.25x	2.97x	160.2	53.77
6x6	2.80x	4.88x	5.04x	4.34x	126.53	29.09
7x7	2.32x	3.91x	5.01x	4.87x	117.46	24.11
8x8	2.07x	4.12x	6.05x	6.39x	108.4	16.96

Table I shows the simulation speed of the individual particle simulators. We performed all experiments on an Intel Xeon E3-1240 with 4 cores with 2 threads per core. Our test infrastructure provides a theoretical speedup of maximum 8x. The speedup is dependent on the number of particles and the grid size. For a 8x8 and 6x6 grid size a speedup of 6.39x respectively 5.04x is measured. The increasing speedup is dependent on the number of particles in the simulation. More particles increase the number crunching and consequently the execution time of the parallel threads. The sequential communication becomes a minor part of the simulation.

The design has several sequential parts, which cannot be parallelized. One reason is due to the initialization and final synchronization of the individual tiles. A second reason is due to the communication which is performed in a double handshake fashion. This means a tile is blocked until the receiving tile completes the communication.

TABLE II
Exchanged Particles of the Particle Simulator

	7x7		8x8	
Particles	20k	60k	20k	60k
seq.	467,728	1,321,247	497,111	1,356,083
par.	467,728	1,321,247	497,111	1,356,083

Table II shows the simulation characteristics of the sequential and parallel simulation to demonstrate the accuracy. All simulations are performed with 20,000 and 60,000 particles for a particle simulator of 6x6 and 8x8 tiles. Both, the sequential and the parallel simulation have identical numbers of communicated particles.

V. Conclusion and Future Work

In this paper, we propose an efficient solution for accurate and parallel simulation of SystemC models with 3rd party libraries. Our approach does not trade off simulation speed for simulation accuracy as do time decoupled modeling techniques. In contrast to previous compiler related work, our RISC infrastructure allows to simulate models which are not statically analyzable. Also, we are now able to simulate models in parallel which include 3rd party libraries for the first time. We demonstrated a simulation speedup of 6.39x while maintaining 100% accuracy.

In future work, we plan to integrate our infrastructure with virtual platforms. Furthermore, we plan to extend advanced support of the pointer analysis to avoid unnecessary conflicts. Also, we plan to introduce techniques to fold the complexity of conflict tables.

ACKNOWLEDGMENT

This work has been supported in part by substantial funding from Intel Corporation for the project titled "Out-of-Order Parallel Simulation of SystemC Virtual Platforms on Many-Core Architectures". The authors thank Intel Corporation for the valuable support.

References

[1] "IEEE Standard SystemC Language Reference Manual, IEEE Std 1666-2011," 2011.

[2] G. Glaser, G. Nitschey, and E. Hennig, "Temporal Decoupling with Error-Bounded Predictive Quantum Control," in *Forum on Specification and Design Languages (FDL)*, 2015.

[3] W. Chen, X. Han, and R. Dömer, "Out-of-Order Parallel Simulation for ESL Design," in *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, 2012.

[4] R. M. Fujimoto, "Parallel Discrete Event Simulation," *Commun. ACM*, vol. 33, no. 10, 1990.

[5] W. Chen, X. Han, and R. Dömer, "May-Happen-in-Parallel Analysis based on Segment Graphs for Safe ESL Models," in *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, 2014.

[6] J. H. Weinstock, R. Leupers, G. Ascheid, D. Petras, and A. Hoffmann, "SystemC-Link: Parallel SystemC Simulation using Time-Decoupled Segments," in *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, 2016.

[7] J. H. Weinstock, C. Schumacher, R. Leupers, G. Ascheid, and L. Tosoratto, "Time-Decoupled Parallel SystemC Simulation," in *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, 2014.

[8] C. Roth, S. Reder, H. Bucher, O. Sander, and J. Becker, "Adaptive algorithm and tool flow for accelerating systemc on many-core architectures," in *Digital System Design (DSD), 17th Euromicro Conference*, 2014.

[9] K. M. Chandy and J. Misra, "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, no. 5, pp. 440–452, Sept 1979.

[10] R. Sinha, A. Prakash, and H. D. Patel, "Parallel Simulation of Mixed-abstraction SystemC Models on GPUs and Multicore CPUs," in *17th Asia and South Pacific Design Automation Conference (ASPDAC)*, 2012.

[11] T. Schmidt, G. Liu, and R. Dömer, "Automatic generation of thread communication graphs from SystemC source code," in *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, 2016.