RISC Compiler and Simulator, Release V0.5.0: Out-of-Order Parallel Simulatable SystemC Subset

Guantao Liu, Tim Schmidt, Zhongqi Cheng, Daniel Mendoza and Rainer Dömer

Technical Report CECS-18-TBD
September 30, 2018

Center for Embedded and Cyber-physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
+1 (949) 824-8919

{guantaol,schmidtt,zhongqc,dmmendo1,doemer}@uci.edu
http://www.cecs.uci.edu/~doemer/risc.html

RISC Compiler and Simulator, Release V0.5.0: Out-of-Order Parallel Simulatable SystemC Subset

Guantao Liu, Tim Schmidt, Zhongqi Cheng, Daniel Mendoza and Rainer Dömer

Technical Report CECS-18-TBD
September 30, 2018

Center for Embedded and Cyber-physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
+1 (949) 824-8919

{guantaol,schmidtt,zhongqc,dmmendo1,doemer}@uci.edu
http://www.cecs.uci.edu/~doemer/risc.html

Abstract

IEEE SystemC is widely used in industry and academia to specify and simulate Electronic System Level (ESL) models. Despite the wide availability of multi-core processor hosts, however, the reference SystemC simulator is still based on sequential Discrete Event Simulation (DES) and executes only a single thread at any time.

In recent years parallel SystemC simulators were proposed which run multiple threads in parallel based on synchronous Parallel Discrete Event Simulation (PDES) semantics. Synchronous PDES, however, limits parallel execution to threads that run at the same time and delta cycle. Also, most approaches require manual preparation of the SystemC model and rely on the designer to perform difficult conflict analysis.

In this report, we describe the advanced Recoding Infrastructure for SystemC (RISC) approach where a dedicated SystemC compiler and advanced parallel simulator implement Out-of-Order Parallel Discrete Event Simulation (OoO PDES) for SystemC. Using automatic conflict analysis based on Segment Graph (SG) abstraction, OoO PDES can execute threads safely in parallel and out-of-order (ahead of time) and thus achieves fastest simulation speed, but nevertheless maintains the classic SystemC modeling semantics.

This report describes the RISC Compiler and Simulator and details the SystemC subset supported by the RISC Release V0.5.0, as of September 30, 2018. In comparison to the previous V0.4.0 release in 2017, RISC is more efficient and robust, and supports Partial Segment Graphs and new SystemC model visualization.

Contents

List of Figures

List of Tables

RISC Compiler and Simulator, Release V0.5.0: Out-of-Order Parallel Simulatable SystemC Subset

Guantao Liu, Tim Schmidt, Zhongqi Cheng, Daniel Mendoza and Rainer Dömer
Center for Embedded and Cyber-physical Systems
University of California, Irvine
Irvine, CA 92697-2620, USA
{guantaol,schmidtt,zhongqc,dmmendo1,doemer}@uci.edu
http://www.cecs.uci.edu/~doemer/risc.html

Abstract

IEEE SystemC is widely used in industry and academia to specify and simulate Electronic System Level (ESL) models. Despite the wide availability of multi-core processor hosts, however, the reference SystemC simulator is still based on sequential Discrete Event Simulation (DES) and executes only a single thread at any time.

In recent years parallel SystemC simulators were proposed which run multiple threads in parallel based on synchronous Parallel Discrete Event Simulation (PDES) semantics. Synchronous PDES, however, limits parallel execution to threads that run at the same time and delta cycle. Also, most approaches require manual preparation of the SystemC model and rely on the designer to perform difficult conflict analysis.

In this report, we describe the advanced Recoding Infrastructure for SystemC (RISC) approach where a dedicated SystemC compiler and advanced parallel simulator implement Out-of-Order Parallel Discrete Event Simulation (OoO PDES) for SystemC. Using automatic conflict analysis based on Segment Graph (SG) abstraction, OoO PDES can execute threads safely in parallel and out-of-order (ahead of time) and thus achieves fastest simulation speed, but nevertheless maintains the classic SystemC modeling semantics.

This report describes the RISC Compiler and Simulator and details the SystemC subset supported by the RISC Release V0.5.0, as of September 30, 2018. In comparison to the previous V0.4.0 release in 2017, RISC is more efficient and robust, and supports Partial Segment Graphs and new SystemC model visualization.

1 Introduction

As an IEEE standard [1], the SystemC System Level Description Language (SLDL) is widely used for the specification, modeling, validation and evaluation of Electronic System Level (ESL) models. Under the Accellera Systems Initiative [2], the SystemC Language Working Group [3] maintains not only the official SystemC language definition, but also provides an open source proof-of-concept library [4] that can be used to simulate SystemC design models. However, implementing the classic scheme of Discrete Event Simulation (DES), this reference simulator runs sequentially and cannot utilize the parallel computing resources available on multi-core (or many-core) processor hosts. This severely limits the execution speed of SystemC simulation.

In order to provide faster simulation, Parallel Discrete Event Simulation (PDES) [5] has recently gained again significant attraction (examples include [6], [7], [8], [9], [10], and [11]). The PDES approach issues multiple threads (i.e. `SC_METHOD`, `SC_THREAD` and `SC_CTHREAD`) concurrently and runs them on the available processor cores in parallel. In turn, the simulation speed increases significantly.

Regular PDES is synchronous, however. That is, time advances globally and all threads execute in lock-step fashion. Here, the total order of time imposed by synchronous PDES still limits the opportunities for parallel execution. When a thread completes its evaluation phase, it has to wait until all other threads finish their evaluation phases as well. Earlier completed threads must stop at the simulation cycle barrier and available processor cores are left idle until all runable threads reach the cycle barrier.

In order to overcome this problem, we have developed a novel technique called Out-of-Order Parallel Discrete Event Simulation (OoO PDES) [12, 13, 14, 15]. By localizing the simulation time to individual threads and carefully handling events at different times, the simulation kernel can issue threads in parallel and ahead of time, following a partial order of time without loss of accuracy. Thus, Ooo PDES significantly reduces the idle time of available parallel processor cores and results in maximum simulation speed, while maintaining the traditional language and modeling semantics.

The OoO PDES technique was originally implemented based on the SpecC language [16, 17, 18, 19]. In this report, we document our efforts to apply OoO PDES to the IEEE SystemC SLDL [20, 21, 1] which is both the de-facto and official standard for ESL design today. In particular, we describe our Recoding Infrastructure for SystemC (RISC) [22] which consists of a dedicated SystemC compiler and corresponding out-of-order parallel simulator and implements OoO PDES with prediction for SystemC [23].

The remainder of this report is organized as follows: After a brief description of the simulator scheduling algorithms used for DES, PDES and OoO PDES in Section 2, we describe the RISC Compiler and Simulator proof-of-concept prototype in Section 3. In Section 4, we then list in detail the SystemC subset that is supported by the current RISC Release V0.5.0 (2018-09-30)[1] and finally conclude this report in Section 6.

2 Out-of-Order Parallel Simulation

In this section, we briefly outline the scheduling algorithm used in out-of-order parallel simulation. We do this incrementally, starting from the traditional Discrete Event Simulation (DES) scheduler, then describe the synchronous Parallel DES (PDES) extension, and finally define the Out-of-Order PDES (OoO PDES) scheduling algorithm.

2.1 Notations

To formally describe the discrete event scheduling algorithms, the following notations are introduced.

1. Each SystemC thread (`SC_METHOD`, `SC_THREAD` and `SC_CTHREAD`) is assigned a localized time stamp (t_{th}, δ_{th}).
2. Each event (`sc_event`) is assigned a notification time stamp (t_e, δ_e), where $EVENTS = \cup EVENTS_{t,\delta}$.
3. Threads are grouped into different queues. Specifically,
   (a) $QUEUES = \{READY, RUN, WAIT, WAITTIME\}$.
   (b) $READY = \cup th_{t,\delta}$ where Thread th is ready to run at time (t, δ).
   (c) $RUN = \cup th_{t,\delta}$ where Thread th is running at time (t, δ).
   (d) $WAIT = \cup th_{t,\delta}$ where Thread th is waiting since time (t, δ).
   (e) $WAITTIME = \cup th_{t,0}$ where Thread th is waiting for simulation time advance to $(t, 0)$.

[1] Earlier versions of this technical report document the prior alpha release in 2015 [24], the beta release in 2016 [25], and release v0.4.0 in 2017 [26].

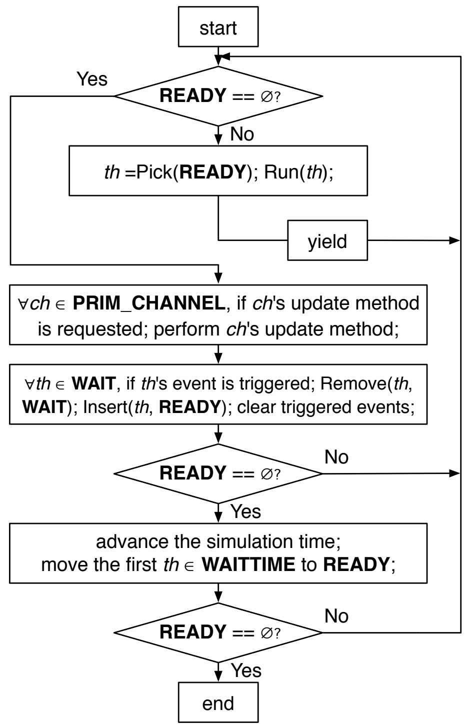

Figure 1: Traditional Discrete Event Simulation (DES) scheduler for SystemC.

2.2 Discrete Event Scheduler

The Accellera reference simulation library of SystemC [4] is based on DES. Figure 1 depicts such a traditional DES scheduling algorithm. In DES, a single thread is running at all times. When all threads in the *READY* and *RUN* queues complete their current delta cycle, the root thread resumes and performs the update and notification phase. Then threads are woken up and moved from the *WAIT* queue back into the *READY* queue. A new delta cycle begins.

If no threads are ready after the update and notification phase, the current time cycle finishes. The simulation kernel advances the simulation time and processes the earliest timed event from the $WAITTIME$ queue. A new cycle begins for the updated simulated time.

Finally, when both the $WAITTIME$ and *READY* queues are empty, the simulation terminates.

2.3 Parallel Discrete Event Scheduler

In comparison to DES, regular synchronous PDES issues multiple threads (`SC_METHOD`, `SC_THREAD` and `SC_CTHREAD`) concurrently in a delta cycle. These threads can then execute truly in parallel on the multiple available processor cores of the host.

Figure 2 shows the regular synchronous PDES scheduling algorithm. In the evaluation phase, as long as the *READY* queue is not empty and an idle core is available, the PDES scheduler will issue a new thread from the *READY* queue. If a thread finishes earlier than other threads in the same cycle, a new ready thread is assigned to the idle processor core, unless there is no thread available in the *READY* queue, in which case the core is kept idle until the next delta cycle.

It should be emphasized that synchronous PDES implies an absolute barrier at the end of each delta and time cycle. All threads need to wait at the barrier until all other runable threads finish their current evaluation phase.

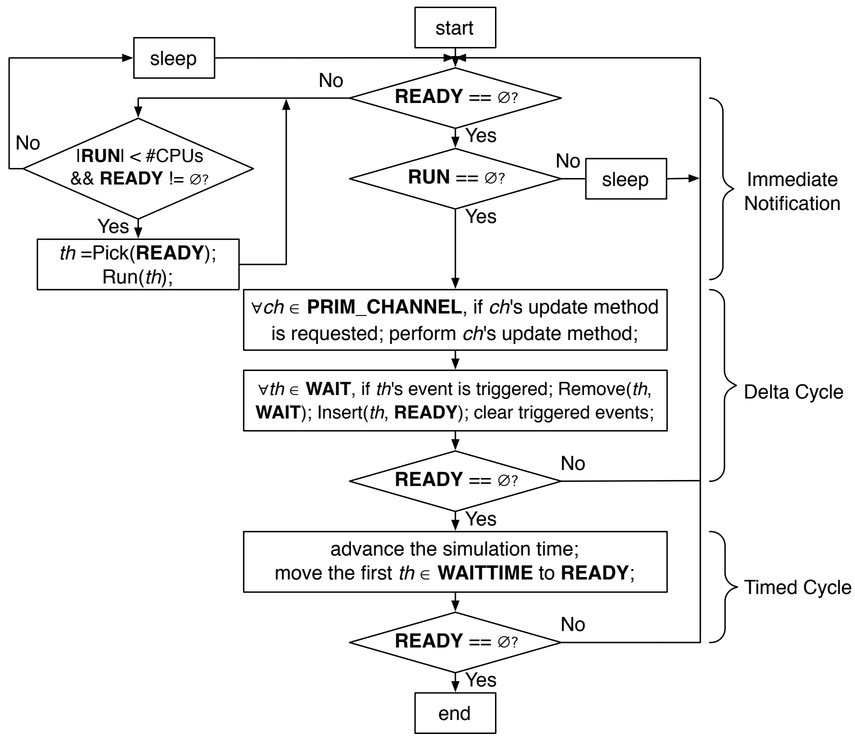

Figure 2: Synchronous Parallel Discrete Event Simulation (PDES) scheduler for SystemC.

Only then the synchronous PDES scheduler resumes and performs the update and notification phases, and finally advances to the next delta or time cycle.

For the SystemC language in particular, there is yet another very important aspect to consider when applying PDES. For semantics-compliant SystemC simulation, complex inter-dependency analysis over all variables in the system model is a prerequisite to parallel simulation [27].

The Standard SystemC Language Reference Manual (LRM) [1] clearly states that *"process instances execute without interruption"*. This requirement is also known as cooperative (or co-routine) multitasking which is assumed by the SystemC execution semantics. As detailed in [27], the particular problem of parallel simulation is specifically addressed in the SystemC LRM [1]:

> *"An implementation running on a machine that provides hardware support for concurrent processes may permit two or more processes to run concurrently, provided that the behavior appears identical to the co-routine semantics defined [...]. In other words, the implementation would be obliged to analyze any dependencies between processes and constrain their execution to match the co-routine semantics."*

We will describe the required dependency analysis in more detail below (in Section 3.3), as it is also needed for out-of-order PDES.

2.4 Out-of-Order Parallel Discrete Event Scheduler

In OoO PDES, we break the strict order of time (the synchronous barrier) by localizing time stamps to each thread. Figure 3 shows the out-of-order parallel DES scheduling algorithm. Since each thread has its own time stamp, the OoO PDES scheduler relaxes the event and simulation time updates, allowing more threads (at different simulation cycles!) to run in parallel and ahead of time. This results in a higher degree of parallelism and thus higher simulation speed.

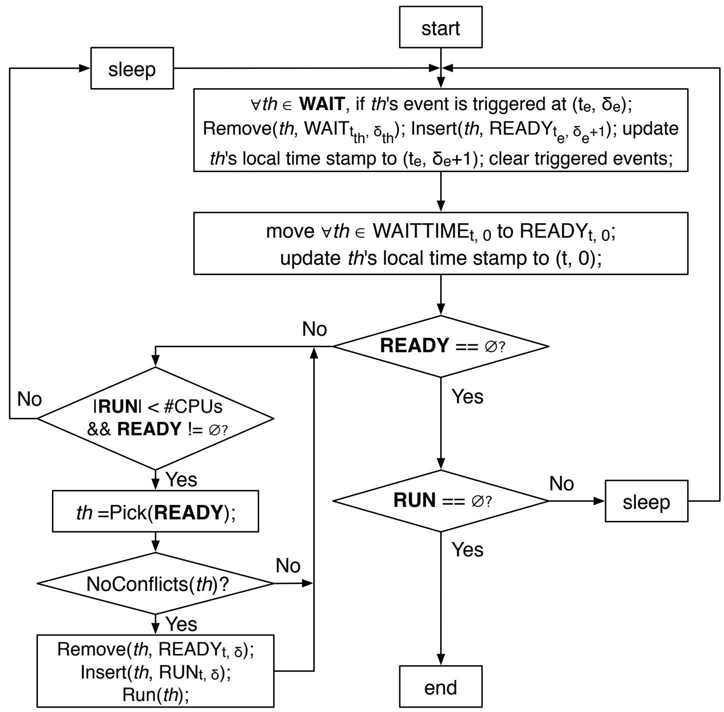

Figure 3: Out-of-Order Parallel Discrete Event Simulation (OoO PDES) scheduler for SystemC.

In comparison to the synchronous PDES in Figure 2, Figure 3 moves threads from the *WAIT* and *WAITTIME* queues into the *READY* queue *as soon as possible*. Also, there is no specific point in the scheduling flow any more for the classic delta and time cycles. Both delta and time updates are performed locally for each thread, provided that there are no possible conflicts in the way (the $NoConflicts(th)$ condition is explained below).

In contrast to Figure 2 which performs requested update methods in primitive channels in each delta cycle, Figure 3 does not contain this step any more. Due to the out-of-order scheduling and the eliminated central scheduling point for delta cycles, it is difficult to determine an efficient and safe point in the OoO PDES scheduler when primitive channel update requests can be served. However, it is always possible to safely fall back to synchronous PDES when primitive channel updates are requested.

Note the $NoConflicts(th)$ condition shown in Figure 3. As already mentioned above for the synchronous PDES, detailed dependency analysis is needed to avoid data or event conflicts for any shared variables among the parallel threads. Only if $NoConflicts(th)$ is true, a new thread is issued for parallel execution (moved from the *READY* to the *RUN* queue).

We will be using advanced static compile-time analysis (and optionally dynamic run-time analysis, see Section 3.3.2) to identify all such potential conflicts. Based on this information (a simple table look-up is sufficient), the OoO PDES scheduler can then at run-time quickly decide whether or not a set of threads has any conflicts with each other.

3 RISC Compiler and Simulator

To realize the OoO PDES approach for the IEEE SystemC language, we present now our Recoding Infrastructure for SystemC (RISC) and describe the overall RISC Compiler and Simulator proof-of-concept prototype (Release V0.5.0 as of 2018-09-30). The RISC software is available as open source and can be downloaded freely from the following web site [22]: http://www.cecs.uci.edu/~doemer/risc.html

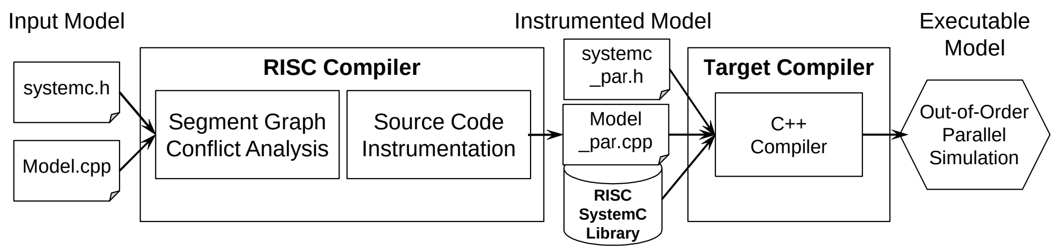

Figure 4: RISC Compiler and Simulator for Out-of-Order PDES of SystemC.

To perform parallel SystemC simulation in maximum compliance with the IEEE standard semantics, we introduce a *dedicated SystemC compiler*. This is in contrast to the traditional SystemC simulation where a regular SystemC-agnostic C++ compiler includes the SystemC headers and links the input model directly against the SystemC library.

As shown in Figure 4, our RISC compiler acts as a frontend that processes the input SystemC model and generates an intermediate model with special instrumentation for OoO PDES. The instrumented parallel model is then linked against the extended RISC SystemC library by the target compiler (a regular C++ compiler) to produce the final executable output model. OoO PDES is then performed simply by running the generated executable model.

From the user perspective, we essentially replace the regular SystemC-agnostic C++ compiler with the SystemC-aware RISC compiler (which in turn calls the underlying C++ compiler). Otherwise, the overall SystemC validation flow remains the same as before. It is just faster due to the parallel simulation.

For reference, the detailed Linux manual page of the RISC compiler `risc` and simulator is included in Appendix A.1 of this report.

Internally, the RISC compiler performs three major tasks, namely Segment Graph (SG) construction, conflict analysis, and source code instrumentation.

3.1 Segment Graph

The first task of the RISC compiler is to parse the SystemC input model into an abstract syntax tree (AST) and then create a SystemC structural representation from the AST which reflects the SystemC module and channel hierarchy, connectivity, and other SystemC-specific relations, similar to the SystemC-clang representation [28, 29]. For details on this part of the RISC application programming interface (API), please refer to the Doxygen-generated documentation [30].

On top of this, the RISC compiler then builds a *Segment Graph (SG)* data structure for the model. A Segment Graph (SG) [12, 15] is a directed graph that represents the code segments executed during the simulation between scheduling steps. That is, every segment is associated with a scheduler entry point, i.e. a `wait` statement in SystemC.

At run time, threads switch back and forth between the states of *running* (threads in *READY* and *RUN* queues) and *waiting* (threads in *WAIT* and *WAITTIME* queues). When *running*, they execute specific segments of their code. These code segments make up the nodes in the Segment Graph, whereas edges in the graph indicate the possible transitions from one segment to another. In other words, the edges in the Segment Graph reflect an abstraction of the model's control flow.

For a formal description of the Segment Graph and its construction algorithm, the interested reader may refer to [15]. For details on the RISC compiler API, please refer to the Doxygen-generated documentation [30].

3.2 Partial Segment Graph

The segment graph is the foundation data structure for the static analysis. However, there are restrictions: the entire source code for the input design must be available in one file, which does not scale. This disables the use of Intellectual Property (IP) and hierarchical file structures.

To solve this problem, we have proposed and implemented a Partial Segment Graph (PSG) as the representation of the behavior model for each separate translation unit or IP. By combining PSGs, our tool is able to reconstruct the complete SG for the input model.

The extended tool flow is shown in Figure 5.

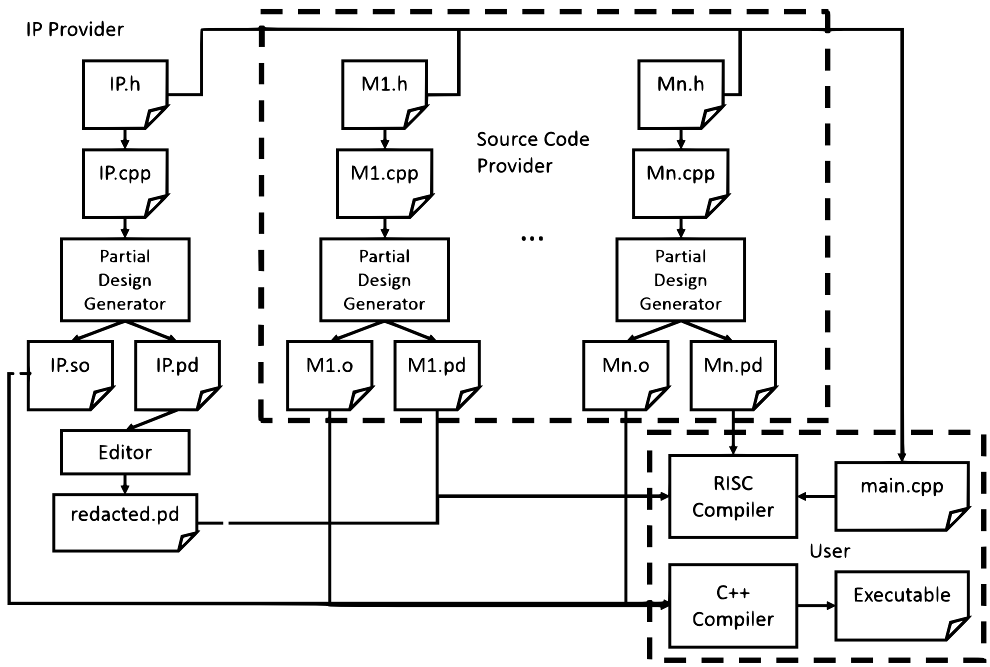

Figure 5: Scaled RISC tool flow with Partial Segment Graph technology.

A PSG is recursively built by traversing the AST of the current translation unit. The main difference between PSG and SG is that PSG is built based on an incomplete AST, where definitions of function calls may be unknown.

To deal with this uncertainty incurred by the non-defining function calls, we introduce three types of PSG nodes, which facilitate the integration of PSGs. They are *Segment Node*, *Partial Segment Node* and *Partial Function Call Node*.

The PSG is constructed by the IP provider. It is stored as a PSG file and is compatible with the Dot format so that the PSG can easily be visualized. The PSG file is shipped together with the IP files to the user. On the user's side, the RISC compiler is able to load and parse the PSG files. Then, the loaded PSGs are integrated to form a complete SG. During integration, *Partial Function Call Node*s are replaced by the corresponding PSGs of the functions. *Partial Segment Node*s are merged into *Segment Node*s. After the integration, the graph becomes a valid and complete SG.

An IP provider can also inspect and redact the automatically generated PSG files so that the implementation details remain hidden. This way the IP users will not be able to obtain the inner implementation and the IP remains protected, while the correctness of behavior model of the design is still maintained.

3.3 Conflict Analysis

The Segment Graph data structure serves as the foundation for segment *conflict analysis*. As outlined earlier, the OoO PDES scheduler must ensure that every parallel thread to be issued has no conflicts with any other threads currently in the *READY* and *RUN* queues. Here, we utilize the RISC compiler to detect any possible conflicts between these threads already at compile time.

Potential conflicts in SystemC include data hazards, event hazards, and timing hazards, all of which may exist among the segments executed by the threads considered for parallel execution. Please refer to [15] for a detailed discussion of these hazards which, if ignored, would become dangerous race conditions at run time.

Both possible hazard detection approaches, namely *static* analysis at compile time and *dynamic* analysis at run time, are supported by the RISC Compiler and Simulator. It should be emphasized that the accuracy of this analysis has significantly improved with the Release V0.5.0. As outlined in detail in [31], the RISC compiler now supports Port Call Path (PCP) sensitive conflict analysis which makes it aware of the actual channel instances used by threads from different modules. This much more precise analysis can avoid false positive conflicts in many cases and thus increases the efficiency of the simulation which, in turn, runs faster.

3.3.1 Static Analysis

Static analysis relies purely on the available information in the SystemC source code of the design model at hand. In this case, the RISC compiler carefully performs conservative identification of the potential hazards in the model.

Identifying all possible hazards is a complex analysis task that requires the full ”understanding” of the module hierarchy. One option is to statically extract the module hierarchy and analyze the individual threads. Here, the RISC compiler follows the approach outlined in [15].

In many cases, however, not all of the needed information can be gathered statically. For instance, design parameters may be passed via the command line, for example, to define the number of modules, certain channel characteristics, or other configuration information. In such SystemC models with a dynamic elaboration phase, the instantiated modules, channels, and ports are typically created by use of loops and `new` operators in a dynamic fashion. Thus, the structural parameters of the model are only available at run time, so they cannot be statically analyzed. In these cases, dynamic analysis is needed.

3.3.2 Dynamic Analysis

Dynamic analysis takes run-time information into account and then augments the classic static analysis. The combination of static and dynamic analysis is here called *hybrid analysis* [32].

Figure 6 shows the extended RISC design flow with support of dynamic analysis. As in the regular compilation flow discussed above in Figure 4, the input SystemC model is processed by the RISC Compiler to generate an executable model for out-of-order parallel simulation, as shown on the top half of Figure 6 from left to right.

The dynamic analysis step, shown on the bottom half of Figure 6, extends the compilation flow by a preprocessing step. The input SystemC model is fed into the RISC Elaborator `elab` which produces an executable model that only performs the SystemC elaboration phase when run. At the end of the elaboration, the executable model automatically traverses the created module hierarchy via the SystemC introspection API and dumps this detailed structural design information, shown as Instance Connectivity Data in Figure 6, into a file

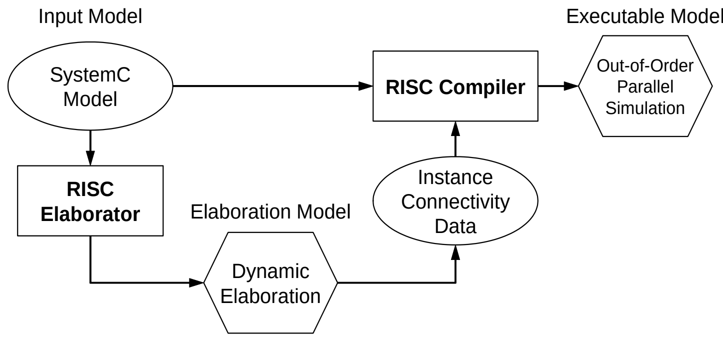

Figure 6: RISC Elaborator feeds dynamic elaboration information to RISC Compiler for precise conflict analysis.

(`model_name.elab`). This file is in turn provided as an input to the RISC compiler, so that the dynamically created design hierarchy and specific instance connectivity can be used for precise conflict analysis. The instance connectivity data file includes the actual module hierarchy, the specific port mapping, and the actual target variable mapping of references.

Note that the use of the RISC Elaborator is optional. Design models, that can be fully analyzed in static fashion, can be fed directly into the RISC Compiler without any pre-processing by the RISC Elaborator.

For reference, the detailed Linux manual pages of the RISC Compiler `risc` and RISC Elaborator `elab` are included in Appendix A.1 and Appendix A.2, respectively.

3.4 Source Code Instrumentation

As a result of the conflict analysis (static, dynamic, or hybrid [32]), the RISC compiler generates several conflict tables that describe all possible conflicts between threads in any two segments. Using this conservative conflict information, the simulator can then at run-time quickly determine by a simple table look-up whether or not it is safe to issue any given thread in parallel or ahead of time.

As shown above in Figure 4, the RISC compiler and simulator work closely together. The compiler performs conservative conflict analysis and passes the analysis results to the simulator which then can make safe scheduling decisions quickly.

To pass information from the compiler to the simulator, we use automatic model instrumentation. That is, the intermediate model generated by the compiler contains instrumented (automatically generated) source code which the simulator can then rely on. At the same time, the RISC compiler also instruments user-defined SystemC channels with automatic protection against race conditions among communicating threads.

In total, the RISC source code instrumentation includes four major components:

1. Segment and instance IDs: Individual threads are uniquely identified by a creator instance ID and their current code location (segment ID). Both IDs are passed into the simulator kernel as additional arguments to scheduler entry functions, including `wait` and thread creation.

2. Data and event conflict tables: Segment concurrency hazards due to potential data conflicts, event conflicts, or timing conflicts are provided to the simulator as two-dimensional tables indexed by a segment ID and instance ID pair. For efficiency, these table entries are filtered for scope, instance path, and reference and port mappings.

3. Current and next time advance tables, and thread state prediction tables: The simulator can make better scheduling decisions by looking ahead in time if it can predict the possible future thread states. This

optimization is discussed in detail in [14] and is available in the RISC Compiler and Simulator in versions 0.4.0 and later. Since thread state prediction for most models requires only little additional compile time but results often in higher simulation speed, it is enabled by default (it can be turned off with the `SYSC_DISABLE_PREDICTION` environment variable, see below).

4. User-defined channel protection: SystemC allows the user to design channels for custom inter-thread communication. To ensure such communication is safe also in the OoO PDES situation where threads execute truly in parallel, the RISC compiler automatically inserts locks (binary semaphores) into these channels so that mutually-exclusive execution of the channel methods is guaranteed. Otherwise, race conditions could exist when communicating threads exchange data.

Note that the source code instrumentation is performed automatically by the RISC Compiler and no user-interaction is necessary. However, the interested user may inspect the instrumented source code. It is stored in a file named `risc_`*`model_name`*`.cpp` which serves as the input file to the compiler backend which in turn then generates the final executable.

3.5 Library Support

In absense of PSG support (Section 3.2), there exists a significant limitation for the described conflict analysis and source code instrumentation. It only works if the compiler has access to the entire source code of the design model. This is typically fine for smaller SystemC benchmark examples, but does not hold true for more complex SystemC models where multiple translation units and/or library files are used. In these cases, the compiler has access only to the function signatures (function declarations in header files), but not to their implementation (function bodies which are pre-compiled in the library or object files). Thus, the compiler cannot analyze the function bodies for potential conflicts, neither can it instrument any segment boundaries (i.e. `wait` calls) in the library code with segment and instance IDs.

In its initial alpha version [24], the RISC Compiler and Simulator operated under the assumption that all library code is thread-safe without any conflicts and does not contain any segment boundaries (no `wait` statements). This is reasonable for the standard C/C++ libraries used in a modern Linux environment, as well as for the specially prepared RISC SystemC simulator library. However, this assumption poses a significant limitation for more complex SystemC models built around custom application libraries.

In order to mitigate this limitation, the beta version [25] and the RISC Compiler and Simulator version 0.4.0 offered basic support for library code by use of *function annotations*. This annotation scheme for library functions provides abstract information for both conflict analysis and segment boundaries [32].

Specifically, the user can annotate function declarations with `pragma` statements which specify whether or not the function poses any potential conflicts. The `pragma` statements can also describe basic situations of `wait` calls that the control flow in the function body contains. For example, the standard math function `sqrt` and the blocking `read` function of the SystemC `sc_fifo` channel can be annotated as follows:

```
// standard math square-root function
#pragma RISC sqrt conflict-free no-wait
double sqrt(double x);

// sc_fifo blocking read function
#pragma RISC read conflict-free looped-wait event
virtual T read();
```

Here, the sqrt function is declared conflict-free because it is thread-safe and has no dangerous side effects. Since this is true for many functions (e.g. most functions in the C standard library), the RISC Compiler assumes this by default. Thus, this pragma statement is not explicitly needed.

The sc_fifo::read function is also declared conflict-free because it operates in a standard SystemC channel that is safely protected by a lock in the RISC simulator library. However, this blocking sc_fifo::read function is annotated as looped-wait because it does contain a wait statement in the body of a loop that is waiting for available data, which is indicated by some event. Thus, the RISC Compiler can take this segment boundary into account when building the Segment Graph for a thread that calls this function.

In general, a function is considered conflict-free if the corresponding function body contains no potential read/write access conflicts to any shared state with the other threads in the simulation model. Otherwise, it must be annotated as not-conflict-free.

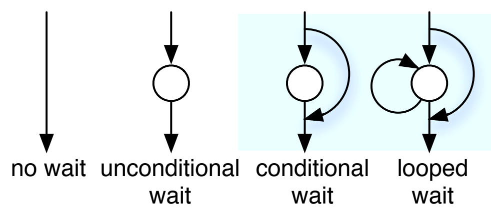

Figure 7: Control-flow abstractions for wait in library functions.

For the annotation of segment boundaries contained in library functions, Figure 7 shows the different control-flow abstractions with regards to wait function calls in the corresponding function body. In the first case, no_wait, the function contains no wait statement and thus is a non-blocking function during the SystemC simulation. The next two cases, conditional_wait and unconditional_wait, apply to functions with a conditional or non-conditional wait statement, respectively. The last case covers the possible encounter of a wait statement in a loop, such as the blocking read call to a sc_fifo channel discussed above.

The last parameter in the RISC pragma annotation specifies the type of the wait statement in the function body, either event for waiting for any notified event, or the minimum time increment that the simulator will incur when executing the corresponding function, such as sc-zero-time or (42,SC_MS).

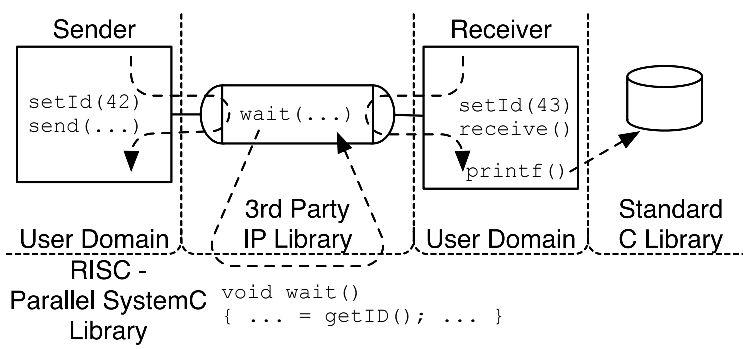

Figure 8: Different source code domains of a design model.

Figure 8 [32] illustrates the different domains of source code in a SystemC model where only the code in the user domain is available for the instrumentation described above in Section 3.4. For library code, any contained wait() calls cannot be instrumented. Here, the RISC Compiler and Simulator (version 0.4.0 and above) instruments the code before such library function calls with setID(SegID) functions that store the upcoming segment IDs for the wait statements in the library in thread-local data. Then, when wait statements without explicit segment ID arguments are executed in the library, the segment IDs are obtained from the thread-local data by use of a getID() function in the RISC simulation library.

Note that with the latest RISC Compiler and Simulator Release V0.5.0 the library support described in this section is still available (for backward compatibility reasons). However, the Partial Segment Graph (PSG) technology described in Section 3.2 offers an alternative solution that is much more general. In particular, the PSG technology resolves two prior limitations. First, the annotations shown in Figure 7 only cover the cases of zero or one wait statement in a library function. Multiple wait statements were not covered. Thus, PSG technology was designed in order to cover general control-flow inside of library functions which are now represented by their own partial segment graphs. Second, PSG technology supports multiple separate translation units by building and storing PSG files together with generated object files that then can be integrated again into a complete SG when the final simulation executable is being built.

3.6 Support for Data-Level Parallelism

As of version 0.4.0, the RISC Compiler and Simulator comes with support for exploiting data-level parallelism, also known as Single-Instruction-Multiple-Data (SIMD) vectorization [33]. Here, an advanced analysis tool, namely the SIMD Advisor simd (see Appendix A.3), can identify possible locations in the SystemC model's source code where data-level parallelism may be exploited for faster simulation (on top of the thread-level parallelism already exploited due to OoO PDES).

The SIMD Advisor adds a pre-analysis step to the RISC Compiler and Simulator tool flow where simd provides the designer with candidates for loop vectorization. Specifically, simd performs advanced thread control-flow and variable access analysis and then reports to the user the source code line numbers where loops qualified for SIMD vectorization are found. The user confirms suitable locations by inserting #pragma simd statements in front of the chosen loops. Finally, the design model is then compiled with the Intel compiler icpc which performs the vectorization and builds the executable for simulation with both thread- and data-level parallelism.

Note that the manual confirmation by the designer is necessary. An example is the following C function:

```
void add(float *a, float *b, float *c, int n)
{
   for(int i=0; i<n; i++)
      { a[i] = a[i] + b[i] + c[i];}
}
```

Here, arrays passed as pointers can only be vectorized if the user asserts that there is no vector dependence in the way. This confirmation step is only possible with application knowledge, not just by static compiler analysis. The RISC SIMD Advisor is aware of SystemC and its concurrent multi-threading semantics, and thus can identify certain loops as potential candidates, but the final data independence assertion must come from the user who knows the application specifics (i.e. that the pointers point to non-overlapping arrays).

Exploiting both thread- and data-level parallelism can be very effective for many design models. Experimental results in [33] show a nearly linear speedup of $N \times M$, where N and M denote the thread and data-level factors, respectively.

The SIMD Advisor is documented in detail in the manual page for simd listed in Appendix A.3.

3.7 Compiler Backend

After the automatic source code instrumentation, the RISC compiler passes the generated intermediate model in file `risc_model_name.cpp` to the underlying regular C++ compiler. That target compiler then produces the final simulation executable by linking the instrumented code against the RISC extended SystemC library.

By default, the RISC Compiler and Simulator rely on the GNU C++ compiler `g++` for the backend code generation. Alternatively, the Intel C++ compiler `icpc` may be used to generate a simulation executable that is optimized for Intel processors with Single-Instruction-Multiple-Data (SIMD) capabilities or the Intel Many-Integrated-Core (MIC) architecture. Please refer to the command-line options `-risc:icpc` and `-risc:mic`, respectively, which are documented in the manual pages for `risc` (see Appendix A.1) and `elab` (see Appendix A.2).

3.8 Simulator

Same as the classic Accellera proof-of-concept implementation [4], the RISC simulator is not an explicit tool, but a run-time library [34] that the generated executable SystemC model is linked against. Thus, simulation is performed by execution of the compiled model, the same way as in the classic tool flow (just faster).

The RISC simulator identifies itself by its log message at the beginning of the simulation run, announcing `RISC 0.5.0` execution after the SystemC language version number (`SystemC 2.3.1`). It also adds the Center for Embedded and Cyber-physical Systems (CECS) as a contributor to the RISC-extended SystemC library.

A simple *HelloWorld* model is shown running in the following example:

```
sh % ./HelloWorld

        SystemC 2.3.1-RISC 0.5.0 --- Sep 30 2018 09:04:24
        Copyright (c) 1996-2018 by CECS and all Contributors,
        ALL RIGHTS RESERVED

Hello World!
```

There are several environment variables which the RISC out-of-order parallel SystemC library recognizes. These are logged at the beginning of the simulation if `SYSC_PRINT_MODE_MESSAGE` is defined.

```
***   RISC simulator mode: out-of-order parallel with prediction  ***
***   SYSC_PRINT_MODE_MESSAGE        is       defined             ***
***   SYSC_SYNC_PAR_SIM              is       not defined         ***
***   SYSC_PRINT_VERBOSE_MESSAGE     is       not defined         ***
***   SYSC_DISABLE_PREDICTION        is       not defined         ***
***   SYSC_PAR_SIM_CPUS              is       64                  ***
```

The environment variable `SYSC_SYNC_PAR_SIM` can be used to force the default out-of-order parallel scheduler to fall-back to synchronous parallel execution. By default (when undefined), `SYSC_SYNC_PAR_SIM` is assumed to be `false`, so out-of-order parallel simulation (OoO PDES) with prediction is performed. On the other hand, if `SYSC_SYNC_PAR_SIM` is defined, the simulator will execute in synchronous PDES fashion.

Also, as indicated above in Section 2.4, the RISC simulator automatically falls back to synchronous execution as soon as primitive SystemC channels are used with requests to update functions. Thus, such models will execute in safe synchronous manner.

The variable SYSC_PRINT_VERBOSE_MESSAGE is used by the RISC simulator at run-time to print debugging information about the simulator queues, event processing, and time advances. Such debugging lines are only printed when SYSC_PRINT_VERBOSE_MESSAGE is defined.

The variable SYSC_DISABLE_PREDICTION is used by the RISC simulator to switch back to non-predictive conflict detection. This avoids scheduling overhead at run time, but usually results in slower simulation due to more false conflicts. If SYSC_DISABLE_PREDICTION is defined, thread state prediction is not used during out-of-order scheduling.

The environment variable SYSC_PAR_SIM_CPUS specifies the maximum number of parallel threads allowed in out-of-order parallel simulation (namely #CPUs in Figure 3). For efficient simulation, this variable should be set to a value suitable for the simulation host, e.g. the number of available CPU cores. If unset, SYSC_PAR_SIM_CPUS defaults to 64.

4 Out-of-Order Parallel Simulatable SystemC Subset

Over more than a decade, the SystemC language [21], which technically is a C++ application programming interface (API) with a corresponding simulation library, has evolved from basic constructs for modeling parallel modules connected by signals and channels to a highly complex set of macros, types, classes, templates, and functions for very advanced modeling (i.e. Transaction Level Modeling (TLM) [35, 36]) and highly optimized simulation of SystemC models. Usually these optimization steps have aimed at higher simulation speed, i.e. by minimizing context switches in the simulator, or at higher levels of abstraction due to purposely relaxed timing. Often, the uninterrupted (sequential) execution semantics on a single processor host have been presumed or are explicitly required.

Along these lines, it has been recognized that there is considerable need to study and adjust or *evolve* the SystemC language towards better support of parallel execution (following some form of suitable PDES semantics). One example of the ongoing discussion within the SystemC community is a presentation at the SystemC Evolution Day 2016 where significant obstacles in the current language standard have been identified [37]. These *seven obstacles* have then been documented also in a letter to the editor of IEEE Embedded System Letters [38].

The RISC Compiler and Simulator aims for advanced parallel execution on multi- and many-core hosts, maximizing the compliance with the current SystemC standard [1]. Changing some assumptions about SystemC simulator execution consequently affects a number of SystemC constructs and APIs which need to be revisited and evaluated anew. The goal of this section is to document this process and status, and enable fruitful discussions.

Below, we describe and list the out-of-order parallel simulatable SystemC subset supported by the current RISC Compiler and Simulator, Release V0.5.0. In particular, Table 1 through Table 8 list for each SystemC construct whether or not it is supported at this time. If applicable, an explanation note is provided that briefly outlines the status and/or the plans for the given feature.

Overall, the current RISC proof-of-concept prototype supports the classic SystemC constructs for hierarchical modeling with modules and interconnected channels by featuring fast multi-threaded execution. However, several specific SystemC features are not supported yet or left undecided at this stage. The status “undecided” in particular indicates that further study is needed to decide whether or not the given construct can be supported in efficient and reasonable manner by RISC and its OoO PDES approach.

4.1 SystemC Hierarchical Structure of Modules and Channels

RISC supports the regular hierarchical and structural composition of the SystemC design model. This includes the SystemC program start (sc_main, sc_start) and the general static or dynamic composition (SC_CTOR)

Table 1: RISC V0.5.0 Out-of-Order Parallel Simulatable SystemC Subset

Name	**Type**	**Supported or not**	**Notes**
sc_abs	function	Undecided	This function may not work with some arithmetic SystemC datatypes.
sc_actions	typedef	Supported	typedef unsigned sc_actions
sc_argc	function	Supported	
sc_argv	function	Supported	
sc_assemble_vector	function	Undecided	Work on this function in the future
sc_assert	macro	Undecided	Work on this macro in the future
sc_attr_base	class	Undecided	Work on this class in the future
sc_attr_cltn	class	Undecided	Work on this class in the future
sc_attribute	class	Undecided	Work on this class in the future
sc_behavior	typedef	Supported	typedef sc_module sc_behavior
sc_bigint	class template	Supported	
sc_biguint	class template	Supported	
sc_bind_proxy	class	Undecided	
sc_bind	macro	Undecided	Work on this macro in the future
sc_bit	type (deprecated)	Undecided	Work on this type in the future
sc_bitref_r	class template	Undecided	Work on this class template in the future
sc_bitref	class template	Undecided	Work on this class template in the future
sc_buffer	class	Undecided	
sc_bv_base	class	Undecided	Work on this class in the future
sc_bv	class template	Undecided	Work on this class template in the future
sc_channel	class	Supported	
sc_clock	class	Not Supported Yet	sc_clock::before_end_of_elaboration() calls sc_spawn().
sc_close_vcd_trace_file	function	Initial support as of v0.5.0	
sc_concatref	class	Undecided	Work on this class in the future
sc_concref_r	class template	Undecided	Work on this class template in the future
sc_context_begin	enumeration	Undecided	
sc_copyright	function	Supported	
sc_cor	class	Supported	
sc_cor_pkg	class	Supported	
sc_cor_pthread	class	Supported	
sc_cor_pkg_pthread	class	Supported	
sc_create_vcd_trace_file	function	Initial support as of v0.5.0	
sc_cref	macro	Undecided	Work on this macro in the future
sc_cthread_process	class	Limited Support	Supported up to Internal Representation
SC_CTHREAD	macro	Limited Support	Supported up to Internal Representation
SC_CTOR	macro	Supported	

Table 2: RISC V0.5.0 Out-of-Order Parallel Simulatable SystemC Subset (continued)

Name	**Type**	**Supported or not**	**Notes**
sc_cycle	function (deprecated)	Not Supported Yet	sc_cycle() calls sc_simcontext::cycle(), which is not supported in the out-of-order simulation in the current release.
sc_delta_count	function	Modified semantics	This function returns the local delta count of the running process.
sc_elab_and_sim	function	Supported	
sc_end_of_simulation_invoked	function	Undecided	Work on this function in the future
sc_event_and_expr	class	Supported	Initial support as of v0.5.0
sc_event_and_list	class	Supported	Initial support as of v0.5.0
sc_event_finder_t	class template	Undecided	Work on this class template in the future
sc_event_finder	class	Undecided	Work on this class in the future
sc_event_or_expr	class	Supported	Initial support as of v0.5.0
sc_event_or_list	class	Supported	Initial support as of v0.5.0
sc_event_queue_if	class	Not Supported Yet	
sc_event_queue	class	Not Supported Yet	The constructor function is not supported by the out-of-order simulation in the current release.
sc_event	class	Limited Support	The immediate notification is not supported by the out-of-order simulation in the current release.
sc_exception	typedef	Undecided	Work on this typedef in the future
sc_export_base	class	Not Supported Yet	No port following in compiler analysis
sc_export	class	Not Supported Yet	No port following in compiler analysis
sc_fifo_blocking_in_if	class	Supported	
sc_fifo_in_if	class	Supported	
sc_fifo_in	class	Supported	
sc_fifo_nonblocking_in_if	class	Supported	
sc_fifo_out_if	class	Supported	
sc_fifo_out	class	Supported	
sc_fifo	class	Limited Support	sc_fifo::operator= is not supported; execution falls back to synchronous PDES
sc_find_event	function	Undecided	Work on this function in the future
sc_find_object	function	Undecided	Work on this function in the future
sc_fix_fast	class	Undecided	Work on this class in the future
sc_fix	class	Undecided	
sc_fixed_fast	class template	Undecided	Work on this class template in the future
sc_fixed	class template	Undecided	

Table 3: RISC V0.5.0 Out-of-Order Parallel Simulatable SystemC Subset (continued)

Name	**Type**	**Supported or not**	**Notes**
SC_FORK	macro	Undecided	Work on this macro in the future
sc_fxcast_context	class	Undecided	Work on this class in the future
sc_fxcast_switch	class	Undecided	Work on this class in the future
sc_fxnum_bitref	class	Undecided	Work on this class in the future
sc_fxnum_fast_bitref	class	Undecided	Work on this class in the future
sc_fxnum_fast_subref	class	Undecided	Work on this class in the future
sc_fxnum_fast	class	Undecided	Work on this class in the future
sc_fxnum_subref	class	Undecided	Work on this class in the future
sc_fxnum	class	Undecided	
sc_fxtype_context	class	Undecided	Work on this class in the future
sc_fxtype_params	class	Undecided	Work on this class in the future
sc_fxval_fast	class	Undecided	Work on this class in the future
sc_fxval	class	Undecided	Work on this class in the future
sc_gen_unique_name	function	Undecided	Work on this function in the future
sc_generic_base	class	Undecided	Work on this class in the future
sc_get_curr_process_handle	function (deprecated)	Supported	
sc_get_current_process_handle	function	Supported	
sc_get_default_time_unit	function (deprecated)	Supported	
sc_get_status	function	Supported	
sc_get_stop_mode	function	Supported	
sc_get_time_resolution	function	Supported	
sc_get_top_level_events	function	Undecided	Work on this function in the future
sc_get_top_level_objects	function	Undecided	Work on this function in the future
SC_HAS_PROCESS	macro	Supported	
sc_hierarchical_name_exists	function	Undecided	Work on this function in the future
sc_in_clk	typedef	Undecided	
sc_in_resolved	class	Undecided	
sc_in_rv	class	Undecided	
sc_in	class	Supported	
sc_in<bool>	class	Supported	
sc_in<sc_dt::sc_logic>	class	Supported	

Table 4: RISC V0.5.0 Out-of-Order Parallel Simulatable SystemC Subset (continued)

Name	Type	Supported or not	Notes
sc_initialize	function (deprecated)	Supported	
sc_inout_clk	type (deprecated)	Undecided	
sc_inout_resolved	class	Undecided	
sc_inout_rv	class	Undecided	
sc_inout	class	Supported	
sc_int_base	class	Supported	
sc_int_bitref_r	class	Undecided	Work on this class in the future
sc_int_bitref	class	Undecided	Work on this class in the future
sc_int	class template	Supported	
sc_interface	class	Supported	
sc_interrupt_here	function	Undecided	Work on this function in the future
sc_is_prerelease	function	Undecided	Work on this function in the future
SC_IS_PRERELEASE	macro	Supported	
sc_is_running	function	Supported	
sc_is_unwinding	function	Supported	
SC_JOIN	macro	Undecided	Work on this macro in the future
sc_length_context	class	Undecided	Work on this class in the future
sc_length_param	class	Undecided	Work on this class in the future
sc_logic	class	Undecided	Work on this class in the future
sc_lv_base	class	Undecided	Work on this class in the future
sc_lv	class template	Undecided	Work on this class template in the future
sc_main	function	Supported	
sc_max_time	function	Not Supported Now	This function is not supported by the out-of-order simulation in the current release.
sc_max	function	Supported	
sc_method_process	class	Limited Support	Supported up to Internal Representation
SC_METHOD	macro	Limited Support	Supported up to Internal Representation
sc_min	function	Supported	
sc_module_name	class	Supported	
sc_module	class	Supported	
SC_MODULE	macro	Supported	
sc_mutex_if	class	Not Supported Now	This class is not supported by the risc compiler in the current release.
sc_mutex	class	Not Supported Now	This class is not supported by the risc compiler in the current release.
sc_object	class	Supported	
sc_out_clk	type (deprecated)	Undecided	

Table 5: RISC V0.5.0 Out-of-Order Parallel Simulatable SystemC Subset (continued)

Name	**Type**	**Supported or not**	**Notes**
sc_out_resolved	class	Undecided	
sc_out_rv	class	Undecided	
sc_out	class	Supported	
sc_pause	function	Undecided	Work on this function in the future
sc_pending_activity_at_current_time	function	Limited Support	Supported when called inside sc_main()
sc_pending_activity_at_future_time	function	Limited Support	Supported when called inside sc_main()
sc_pending_activity	function	Limited Support	Supported when called inside sc_main()
sc_phash	class (deprecated)	Undecided	Work on this class in the future
sc_plist	class (deprecated)	Undecided	Work on this class in the future
sc_port	class	Supported	
sc_port_base	class	Supported	
sc_ppq	class (deprecated)	Undecided	Work on this class in the future
sc_prim_channel	class	Supported	sc_prim_channel::update() is performed in synchronous manner; execution falls back to synchronous PDES
sc_process_b	type (deprecated)	Supported	
sc_process_handle	class	Supported	
sc_pvector	class (deprecated)	Undecided	Work on this class in the future
sc_ref	macro	Undecided	Work on this macro in the future
sc_release	function	Supported	
sc_report_handler_proc	typedef	Undecided	Work on this typedef in the future
sc_report_handler	class	Undecided	Work on this class in the future
sc_report	class	Undecided	Work on this class in the future
sc_semaphore_if	class	Not Supported Yet	This class is not supported by the risc compiler in the current release.
sc_semaphore	class	Not Supported Yet	This class is not supported by the risc compiler in the current release.
sc_sensitive_neg	class (deprecated)	Not Supported Yet	This class is not supported by the risc compiler in the current release.
sc_sensitive_pos	class (deprecated)	Not Supported Yet	This class is not supported by the risc compiler in the current release.
sc_sensitive	class	Supported	Initial support as of v0.5.0
sc_set_default_time_unit	function (deprecated)	Supported	
sc_set_stop_mode	function	Undecided	Work on this function in the future

Table 6: RISC V0.5.0 Out-of-Order Parallel Simulatable SystemC Subset (continued)

Name	**Type**	**Supported or not**	**Notes**
sc_set_time_resolution	function	Supported	
sc_set_vcd_time_unit	member function (deprecated)	Supported	Initial support as of v0.5.0
sc_signal_in_if	class	Limited Support	Supported up to Internal Representation
sc_signal_in_if<bool>	class	Limited Support	Supported up to Internal Representation
sc_signal_in_if<sc_logic>	class	Limited Support	Supported up to Internal Representation
sc_signal_inout_if	class	Limited Support	Supported up to Internal Representation
sc_signal_out_if	type (deprecated)	Limited Support	Supported up to Internal Representation
sc_signal_resolved	class	Limited Support	Supported up to Internal Representation
sc_signal_rv	class	Limited Support	Supported up to Internal Representation
sc_signal_write_if	class	Limited Support	Supported up to Internal Representation
sc_signal	class	Limited Support	Supported up to Internal Representation
sc_signal<bool>	class	Limited Support	Supported up to Internal Representation
sc_signal<sc_logic>	class	Limited Support	Supported up to Internal Representation
sc_signed_bitref_r	class	Undecided	Work on this class in the future
sc_signed_bitref	class	Undecided	Work on this class in the future
sc_signed_subref_r	class	Undecided	Work on this class in the future
sc_signed_subref	class	Undecided	Work on this class in the future
sc_signed	class	Supported	
sc_simcontext	class (deprecated)	Limited Support	sc_simcontext::initial_crunch(), cycle() and other functions are partially supported by the out-of-order simulation in the current release.
sc_simulation_time	function (deprecated)	Supported	
sc_spawn_options	class	Undecided	
sc_spawn	function	Not Supported Now	sc_spawn() is not supported by the out-of-order simulation in the current release.
sc_start_of_simulation_invoked	function	Undecided	Work on this function in the future
sc_start	function	Supported	
sc_start(double)	function	Supported	Initial support as of v0.5.0
sc_status	enumeration	Supported	

Table 7: RISC V0.5.0 Out-of-Order Parallel Simulatable SystemC Subset (continued)

Name	Type	Supported or not	Notes
sc_stop_here	function	Undecided	Work on this function in the future
sc_stop	function	Supported	supported as of v0.3.0
sc_string	class (deprecated)	Undecided	Work on this class in the future
sc_subref_r	class template	Undecided	Work on this class template in the future
sc_subref	class	Undecided	Work on this class in the future
sc_switch	enumeration	Supported	
sc_thread_process	class	Supported	
SC_THREAD	macro	Supported	
sc_time	class	Supported	
sc_time_stamp	function	Supported	
sc_time_to_pending_activity	function	Limited Support	Supported when called inside sc_main()
sc_trace_delta_cycles	function (deprecated)	Undecided	Work on this function in the future
sc_trace_file	class	Supported	Initial support as of v0.5.0; execution falls back to synchronous PDES
sc_trace	function	Supported	Initial support as of v0.5.0; execution falls back to synchronous PDES
sc_ufix_fast	class	Undecided	Work on this class in the future
sc_ufix	class	Supported	
sc_ufixed_fast	class template	Undecided	Work on this class template in the future
sc_ufixed	class template	Supported	
sc_uint_base	class	Supported	
sc_uint_bitref_r	class	Undecided	Work on this class in the future
sc_uint_bitref	class	Undecided	Work on this class in the future
sc_uint_subref_r	class	Undecided	Work on this class in the future
sc_uint_subref	class	Undecided	Work on this class in the future
sc_uint	class template	Supported	
sc_unsigned_bitref_r	class	Undecided	Work on this class in the future
sc_unsigned_bitref	class	Undecided	Work on this class in the future
sc_unsigned_subref_r	class	Undecided	Work on this class in the future
sc_unsigned_subref	class	Undecided	Work on this class in the future
sc_unsigned	class	Supported	
sc_unwind_exception	class	Undecided	Work on this class in the future
sc_value_base	class	Undecided	Work on this class in the future
sc_vector_assembly	class	Undecided	Work on this class in the future
sc_vector_base	class	Undecided	Work on this class in the future
sc_vector	class	Undecided	Work on this class in the future

Table 8: RISC V0.5.0 Out-of-Order Parallel Simulatable SystemC Subset (continued)

Name	Type	Supported or not	Notes
sc_version_major	function	Supported	
sc_version_minor	function	Supported	
sc_version_originator	function	Supported	
sc_version_patch	function	Supported	
sc_version_prerelease	function	Supported	
sc_version_release_date	function	Supported	
sc_version_string	function	Supported	
sc_version	function	Supported	
wait	function	Supported	Full support as of v0.5.0
next_trigger	function	Not Supported Now	This function is not supported by the risc compiler in the current release.
halt	function	Not Supported Now	This function is not supported by the risc compiler in the current release.

of modules (`sc_module`, `SC_MODULE`, `sc_behavior`) and channels (`sc_channel`, `sc_prim_channel`).

Connectivity and communication of the instantiated components is supported through ports (`sc_port`, `sc_in`, `sc_inout`, `sc_out`) and interfaces (`sc_interface`).

In contrast to the traditional Accellera library, which only provides a type alias (`typedef`) `sc_channel` for `sc_module`, the RISC header files explicitly distinguish channel and module classes. Here, a separate `sc_channel` class is inherited from `sc_module`, providing the same functionality, but making the two class types explicit.

Most of the SystemC predefined primitive channels[2] (such as `sc_fifo`) are supported for OoO PDES, except `sc_fifo::operator=` which is not supported yet. For more details, please refer to Tables 1 through 8 and the Doxygen-generated documentation of the RISC simulation library [34].

4.2 SystemC Threads

The explicit and statically or dynamically [32] analyzable multi-threading of a SystemC design model is naturally supported in RISC OoO PDES. This includes SystemC processes (`SC_HAS_PROCESS`, `sc_process_handle`, `sc_thread_process`) and the corresponding threads (`SC_THREAD`). For basic inter-thread synchronization, SystemC event notifications (`sc_event.notify`) and waiting for events or simulation time advance (`wait`) are supported.

However, dynamic SystemC thread creation and deletion (`sc_spawn`, `SC_FORK`, `SC_JOIN`) are not supported at this time.

While the application programming interface (API) for these constructs remains unmodified from the SystemC user perspective, the RISC SystemC kernel internally supports extra parameters or arguments for several of these constructs which are utilized after the automatic source code instrumentation by the RISC compiler (see Section 3.4 above). In particular, segment and instance identifiers are supplied with each of these function calls so

[2] As described in Section 2.4 and Section 3.8, the RISC Compiler and Simulator Release V0.5.0 falls back to synchronous PDES execution when primitive channels with update requests are used in the design model.

that the simulator kernel is aware of the exact thread state upon every scheduler entry. This includes in particular the thread creation constructs (`SC_THREAD`) and wait statements (`wait`), as well as standard communication interface methods (e.g. `sc_fifo_in_if::read`).

4.3 SystemC Transaction Level Modeling (TLM)

While transaction level modeling in general is a natural feature supported by OoO PDES [15], the modeling and implementation choices made by SystemC TLM 2.0 [36] create significant problems for supporting it efficiently in RISC. The root problem here lies in the elimination of explicit channels, which were a key contribution in the early days of research on system-level design [16, 17, 18]. As most researchers agreed, the concept of separation of concerns was of highest importance, and for system-level design in particular, this meant the clear separation of computation (in behaviors or modules) and communication (in channels).

Regrettably, SystemC TLM 2.0 chose to implement communication interfaces directly as sockets in modules [39] and this indifference between channels and modules thus breaks the assumption of communication being safely encapsulated in channels. Without such encapsulating channels, there is little opportunity for safe parallel execution.

With TLM-2.0 modeling guidelines, threads intentionally execute code directly in other modules' boundaries (i.e. in "foreign territory") without any protection. Channel boundaries are omitted and trespassing across module boundaries (via sockets) is encouraged (for the sake of saving context switches in sequential simulation). Such violation of a thread's "home territory" cannot be analyzed by the RISC Compiler and Simulator this time.

A possible solution to this problem is the introduction and analysis of so-called *Socket-Call-Paths* in the RISC thread control-flow analysis which, however, is only at an idea stage at this time and thus requires further study and research.

While a discussion of this obstacle has started at the SystemC Language Working Group [3, 37] and in the overall ESL community [38], it remains unclear at this point how the agressive TLM-2.0 modeling situation can be supported, revised, or worked around. Thus, the RISC Compiler and Simulator V0.5.0 only supports traditional SystemC TLM, not yet SystemC TLM-2.0.

4.4 SystemC Data Types

A large part of the SystemC language covers special data types designed for bit-accurate hardware modeling, simulation time representation, and other ESL specifics. These SystemC data types include `sc_bigint`, `sc_biguint`, `sc_bit`, `sc_bv`, `sc_fix`, `sc_ufix`, `sc_fixed`, `sc_ufixed`, `sc_int`, `sc_uint`, `sc_logic`, and `sc_lv`.

While all these SystemC data types are available in RISC, only a few of them have been validated and tested for being safe in a truly parallel multi-threading context. At this point, RISC supports `sc_int`, `sc_uint`, `sc_fixed`, and `sc_ufixed` (which appear as MT-safe). All other data types are so far untested and may or may not be safely used in OoO PDES.

4.5 SystemC Utilities and Other Constructs

As listed in Table 1 through Table 8, there is a plethora of other SystemC APIs available. Some of these are easily supported in RISC (such as `sc_copyright`, `sc_version_major`, `sc_version_minor`, `sc_version_patch`, `sc_version`), others are not supported yet at this time.

At this point, there is also a large number of special SystemC constructs for which it is unclear whether or not these can be supported in an OoO PDES context with reasonable effort and efficiency. An example of such constructs are those functions which involve or allow to inspect the simulator state at run-time, such as sc_find_event, sc_find_object, sc_get_current_process_handle, sc_get_status, sc_get_time_resolution, sc_get_top_level_events, sc_get_top_level_objects, sc_hierarchical_name_exists, sc_is_running, sc_is_unwinding, sc_simcontext, and sc_status.

On the other hand, access to the current simulated time (sc_time, sc_simulation_time, an essential part of every SystemC model evaluation, is fully supported by RISC OoO PDES.

5 RISC Analysis and Transformation Tools

Utilizing the RISC Internal Representation, the RISC framework also includes tools for the analysis and transformation of SystemC models. As of Release V0.5.0, the RISC visual tool [40] is available which enables the user to visualize the SystemC module hierarchy. As an example, Figure 9 shows the module visualization of a Canny edge detector application.

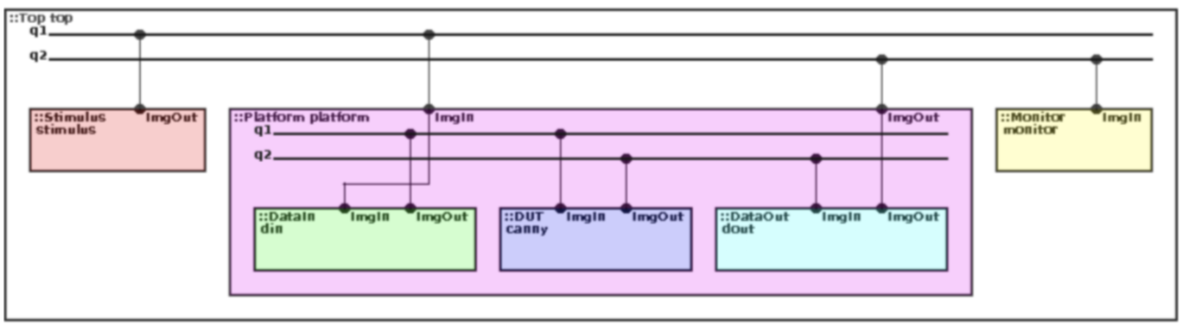

Figure 9: Module hierarchy visualization of a SystemC model of a Canny edge detector.

The visual tool supports a graphical user interface implemented with the Gtk API and renders a specified SystemC source file's module hierarchy, which is drawn using the Cairo API. The tool obtains module data from the SystemC IR in the RISC software stack which contains information about nested modules and thus can recursively iterate through nested lists of child modules in order to obtain enough information to visualize the hierarchy of the entire SystemC source file. The input SystemC source file may contain thousands of lines of code which can make manually drawing a representation of the modules, ports, and channels described by the code a difficult and time-consuming task. Thus the visual tool was created to address this issue. It can automatically generate a visual representation of a SystemC model in a very short period of time.

The RISC visual tool is documented in detail in its manual page which is provided in the Appendix A.4. For a pure textual representation, a similar command-line tool tree is available as well, which is documented in Appendix A.5).

6 Conclusion

While SystemC is the de-facto and official standard language for ESL design, SystemC simulation largely is still performed sequentially following classic DES semantics. Thus, SystemC simulation cannot utilize the parallel processing capabilities available on today's multi- and many-core host computers.

In this report, we have described the Recoding Infrastructure for SystemC (RISC), an aggressive simulation approach beyond traditional parallel DES, where a dedicated SystemC compiler and advanced parallel simulator implement Out-of-Order Parallel Discrete Event Simulation (OoO PDES) with prediction for SystemC. This approach can exploit parallel computing resources at the thread- and data-level to the maximum extend and thus reaches fastest simulation speed. At the same time, RISC OoO PDES largely maintains the traditional SystemC modeling semantics.

This technical report documents the RISC Compiler and Simulator and supporting tools, and details the SystemC subset supported by the RISC Release V0.5.0. In contrast to the previous alpha [24], beta [25], and version 0.4.0 [26] releases, the RISC Compiler and Simulator Release V0.5.0 is more robust and easier to install, and features Partial Segment Graph (PSG) technology (see Section 3.2) for multiple translation units and 3rd-party IP libraries without source code, more precise conflict analysis based on port-call-path (PCP) analysis [31], and provides new tools for graphical SystemC model visualization (see Section 5).

Future work includes several areas of technical extensions and further research. Technical improvements include addressing the limitations in the currently supported SystemC subset and other maintenance tasks including improved documentation and, of course, bug fixes.

In terms of future research, two main limitations need to be addressed. First, TLM-2.0 modeling should be supported. Here, communication is not properly encapsulated in channels as it is in traditional TLM and classic SystemC modeling. Instead, TLM-2.0 modeling lets threads execute directly in “foreign context” without any protection and thus trespasses channel boundaries which cannot be analyzed by RISC at this time. A possible solution to this problem is the introduction of so-called *Socket-Call-Paths* into the RISC analysis which, however, remains at an early idea stage at this point and thus requires further study.

Second, the SystemC constructs for modeling at the Register Transfer Level (RTL) of abstraction are largely not supported yet. Prior focus was on abstract modeling at the Embedded System Level (ESL), but the large amount of legacy RTL models demands support for efficient parallel simulation as well.

As we move on in these future endeavors, we will update and extend the Recoding Infrastructure for SystemC (RISC) and this corresponding technical report accordingly.

Acknowledgements

The RISC project has been supported in part by substantial funding from Intel Corporation under an initial seed grant and a following three year grant for the project titled *“Out-of-Order Parallel Simulation of SystemC Virtual Platforms on Many-Core Architectures”*. The recent improvements documented in this report have been supported by funding for the project titled *“Scaling the Recoding Infrastructure for Parallel SystemC Simulation”*.

The authors thank Intel Corporation for the valuable support and express special gratitude to Ajit Dingankar, Desmond Kirkpatrick and Abhijit Davare for fruitful discussions, productive feedback and invaluable insights.

References

[1] IEEE Computer Society. *IEEE Standard 1666-2011 for Standard SystemC Language Reference Manual*. IEEE, New York, USA, 2011.

[2] Accellera Systems Initiative. http://www.accellera.org.

[3] SystemC Language Working Group (LWG). http://accellera.org/activities/working-groups/systemc-language.

[4] SystemC Language Working Group. SystemC 2.3.1, Core SystemC Language and Examples. http://accellera.org/downloads/standards/systemc.

[5] Richard Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, Oct 1990.

[6] Christoph Schumacher, Rainer Leupers, Dietmar Petras, and Andreas Hoffmann. parSC: Synchronous Parallel SystemC Simulation on Multi-Core Host Architectures. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*, pages 241–246, 2010.

[7] Dukyoung Yun, Jinwoo Kim, Sungchan Kim, and Soonhoi Ha. Simulation Environment Configuration for Parallel Simulation of Multicore Embedded Systems. In *Proceedings of the Design Automation Conference (DAC)*, pages 345–350, 2011.

[8] Ezudheen P, Priya Chandran, Joy Chandra, Biju Puthur Simon, and Deepak Ravi. Parallelizing SystemC Kernel for Fast Hardware Simulation on SMP Machines. In *PADS '09: Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, pages 80–87, 2009.

[9] Rohit Sinha, Aayush Prakash, and Hiren D. Patel. Parallel simulation of mixed-abstraction SystemC models on GPUs and multicore CPUs. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, 2012.

[10] Weiwei Chen, Xu Han, and Rainer Dömer. Multi-Core Simulation of Transaction Level Models using the System-on-Chip Environment. *IEEE Design and Test of Computers*, 28(3):20–31, May/June 2011.

[11] 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*, Dresden, Germany, March 2014.

[12] Weiwei Chen, Xu Han, and Rainer Dömer. Out-of-Order Parallel Simulation for ESL Design. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, March 2012.

[13] Weiwei Chen and Rainer Dömer. An Optimizing Compiler for Out-of-Order Parallel ESL Simulation Exploiting Instance Isolation. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 461–466, February 2012.

[14] Weiwei Chen and Rainer Dömer. Optimized Out-of-Order Parallel Discrete Event Simulation using Predictions. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, March 2013.

[15] Weiwei Chen, Xu Han, Che-Wei Chang, Guantao Liu, and Rainer Dömer. Out-of-Order Parallel Discrete Event Simulation for Transaction Level Models. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 33(12):1859–1872, December 2014.

[16] Jianwen Zhu, Rainer Dömer, and Daniel D. Gajski. Syntax and semantics of the SpecC language. In *Proceedings of the International Symposium on System Synthesis*, Osaka, Japan, December 1997.

[17] Daniel D. Gajski, Jianwen Zhu, Rainer Dömer, Andreas Gerstlauer, and Shuqing Zhao. *SpecC: Specification Language and Design Methodology*. Kluwer Academic Publishers, 2000.

[18] Andreas Gerstlauer, Rainer Dömer, Junyu Peng, and Daniel D. Gajski. *System Design: A Practical Guide with SpecC*. Kluwer Academic Publishers, 2001.

[19] Rainer Dömer, Andreas Gerstlauer, and Daniel Gajski. *SpecC Language Reference Manual, Version 2.0*. SpecC Technology Open Consortium, http://www.specc.org, December 2002.

[20] Open SystemC Initiative, http://www.systemc.org. *Functional Specification for SystemC 2.0*, 2000.

[21] Thorsten Grötker, Stan Liao, Grant Martin, and Stuart Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[22] Guantao Liu, Tim Schmidt, Zhongqi Cheng, Daniel Mendoza, and Rainer Doemer. Recoding Infrastructure for SystemC (RISC). http://www.cecs.uci.edu/~doemer/risc.html.

[23] Rainer Dömer, Guantao Liu, and Tim Schmidt. Parallel simulation. In Soonhoi Ha and Jürgen Teich, editors, *Handbook of Hardware/Software Codesign*, pages 1–32. Springer Netherlands, Dordrecht, 2017.

[24] Guantao Liu, Tim Schmidt, and Rainer Dömer. RISC Compiler and Simulator, Alpha Release V0.2.1: Out-of-Order Parallel Simulatable SystemC Subset. Technical Report CECS-TR-15-02, Center for Embedded and Cyber-physical Systems, University of California, Irvine, October 2015.

[25] Guantao Liu, Tim Schmidt, and Rainer Dömer. RISC Compiler and Simulator, Beta Release V0.3.0: Out-of-Order Parallel Simulatable SystemC Subset. Technical Report CECS-TR-16-06, Center for Embedded and Cyber-physical Systems, University of California, Irvine, September 2016.

[26] Guantao Liu, Tim Schmidt, Zhongqi Cheng, and Rainer Dömer. RISC Compiler and Simulator, Release V0.4.0: Out-of-Order Parallel Simulatable SystemC Subset. Technical Report CECS-TR-17-05, Center for Embedded and Cyber-physical Systems, University of California, Irvine, July 2017.

[27] Rainer Dömer, Weiwei Chen, Xu Han, and Andreas Gerstlauer. Multi-Core Parallel Simulation of System-Level Description Languages. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, pages 311–316, January 2011.

[28] Anirudh Kaushik and Hiren D. Patel. SystemC-clang: An Open-source Framework for Analyzing Mixed-abstraction SystemC Models. In *Proceedings of the Forum on Specification and Design Languages (FDL)*, Paris, France, September 2013.

[29] Hiren Patel. "SystemC-clang: SystemC parser using the clang front-end". https://github.com/hdpatel/systemcclang.

[30] Tim Schmidt. Recoding Infrastructure for SystemC (RISC) API, Version 0.5.0. www.cecs.uci.edu/~doemer/risc/v050/html_risc/index.html.

[31] Tim Schmidt, Zhongqi Cheng, and Rainer Dömer. Port Call Path Sensitive Conflict Analysis for Instance-Aware Parallel SystemC Simulation. In *Proceedings of the Design, Automation and Test in Europe (DATE) Conference*, Dresden, Germany, March 2018.

[32] Tim Schmidt, Guantao Liu, and Rainer Dömer. Hybrid Analysis of SystemC Models for Fast and Accurate Parallel Simulation. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC)*, January 2017.

[33] Tim Schmidt, Guantao Liu, and Rainer Dömer. Exploiting Thread and Data Level Parallelism for Ultimate Parallel SystemC Simulation. In *Proceedings of the Design Automation Conference (DAC)*, June 2017.

[34] Guantao Liu. Out-of-Order Parallel SystemC (OOPSC) API, Version 0.5.0. http://www.cecs.uci.edu/~doemer/risc/v050/html_oopsc/index.html.

[35] Frank Ghenassia. *Transaction-Level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems*. Springer, 2005.

[36] Open SystemC Initiative (OSCI). *OSCI TLM-2.0 Language Reference Manual*. OSCI, July 2009.

[37] Rainer Dömer. *Seven Obstacles in the Way of Parallel SystemC Simulation*. Presentation at SystemC Evolution Day 2016, Munich, Germany, May 2016.

[38] Rainer Dömer. Seven obstacles in the way of standard-compliant parallel SystemC simulation. *IEEE Embedded Systems Letters*, 8(4):81–84, December 2016.

[39] David C. Black. The Definitive Guide to SystemC: TLM-2.0 and the IEEE 1666-2011 Standard. Tutorial at Design Automation Conference, San Francisco, California, June 2015.

[40] Daniel Mendoza and Rainer Dömer. A Tool for Visualization of SystemC Models. Technical Report CECS-TR-17-06, Center for Embedded and Cyber-physical Systems, University of California, Irvine, November 2017.

[41] Tim Schmidt, Guantao Liu, and Rainer Dömer. Automatic Generation of Thread Communication Graphs from SystemC Source Code. In *Proceedings of the International Workshop on Software and Compilers for Embedded Systems*, May 2016.

[42] Guantao Liu, Tim Schmidt, and Rainer Dömer. A Segment-Aware Multi-Core Scheduler for SystemC PDES. In *Proceedings of the IEEE International High Level Design Validation and Test Workshop*, October 2016.

[43] Kasra Moazzemi, Rainer Dömer, and Aparna Chandramowlishwaran. A SystemC Model for N-body Problems and its Parallel Design Space Exploration. Technical Report CECS-TR-16-09, Center for Embedded and Cyber-physical Systems, University of California, Irvine, November 2016.

[44] Zhongqi Cheng and Rainer Dömer. A SystemC Model of a Bitcoin Miner. Technical Report CECS-TR-16-04, Center for Embedded and Cyber-physical Systems, University of California, Irvine, September 2016.

[45] Farah Arabi and Rainer Dömer. A Light Weight SystemC Library for Faster Compilation. Technical Report CECS-TR-16-07, Center for Embedded and Cyber-physical Systems, University of California, Irvine, October 2016.

A Appendix

A.1 Manual Page of the RISC Compiler and Simulator

NAME

risc – Recoding Infrastructure for SystemC (RISC) Compiler and Simulator

SYNOPSIS

risc [*options*] *design* [*options*]

DESCRIPTION

risc is a dedicated compiler for the SystemC language. The purpose of **risc** is to parse, analyze, instrument, and compile a SystemC source program into an executable program for out-of-order parallel simulation. **risc** is a frontend source-to-source compiler for SystemC built on top of the ROSE compiler infrastructure with GNU or Intel C++ as backend target compiler. As such, **risc** relies on and supports also most of the ROSE and GNU compiler options.

Using the command syntax shown in the synopsis above, the specified *design* is compiled. By default, **risc** reads the SystemC source file, performs preprocessing and builds an internal representation (abstract syntax tree) and a Segment Graph (SG) of the model. Next, segment conflict analysis is performed and the design model is instrumented for Out-of-Order Parallel Discrete Event Simulation (OoO PDES). Finally, instrumented C++ code is generated, compiled, and linked into an executable file that can be run for fast parallel simulation.

On successful completion, the exit value 0 is returned. In case of errors during processing, an error code with a brief diagnostic message is written to the standard error stream and the compilation is aborted with an exit value greater than zero.

For preprocessing and C++ compilation into an executable file, **risc** relies on the availability of an external C++ compiler which is used automatically in the background. By default, the GNU C++ compiler **g++** is used. Alternatively (see options *–risc:icpc* and *–risc:mic* below), the Intel C++ compiler **icpc** may be used to generate an executable optimized for Intel processors with SIMD capabilities or the Intel Many-Integrated-Core (MIC) architecture.

ARGUMENTS

design specifies the file name of the input SystemC design model; by default, the base name of *design* is used as base name for the intermediate and output files;

OPTIONS

–h | *—-help* print the **risc** compiler version and a brief usage information message to standard output and quit;

–v | *—-verbose* increment the verbosity level so that all tasks performed are logged to standard error (default: be silent); at level 1, high-level messages about the tasks performed are displayed; at level 2, additional details such as input and output file names are listed; at level 3, very detailed information about each executed task is printed;

–vv increment the verbosity level by two counts (same as *–v –v*);

–vvv increment the verbosity level by three counts (same as *–v –v –v*);

–w | *—-warnings* increment the warning level so that compiler warning messages are enabled (default: warnings are disabled); four levels are supported ranging from only important warnings (level 1) to pedantic warnings (level 4); for most cases, warning level 2 is recommended (*–w –w*);

–ww increment the warning level by two counts (same as *–w –w*);

–www increment the warning level by three counts (same as *–w –w –w*);

–g add a symbol table suitable for debugging (e.g. using **gdb**) to the generated object files and simulation executable (default: no debugging symbols);

–O | *–O level* optimize the generated simulation executable for higher execution speed and/or less memory usage (default: no optimization);

–Idir add the specified *dir* to the include path (extend the list of directories to be searched for including source files); include directories are searched in the order of their specification; the standard include path ($SYSTEMC_LW_HOME/include or $SYSTEMC_OOP_HOME/include) is automatically appended to this list; by default, only the standard include directories are searched;

–Ldir add the specified *dir* to the library path (extend the list of directories to be searched for linker libraries); the library path is searched in the specified order; the standard library path ($SYSTEMC_OOP_HOME/lib) is automatically appended to this list; by default, only the standard library path is searched;

–llib add the specified *lib* to the list of libraries for the linker so that the executable is linked against *lib;* libraries are linked in the specified order; the standard libraries (i.e. -lsystemc) are automatically appended to this list; by default, only standard libraries are used;

–c perform only the preprocessing, analysis, instrumentation, and compilation tasks; skip the final linking stage so that only an object file is created (default: perform all tasks including linking);

–o output file specify the name of the final output file explicitly (default: a.out);

–psg switch to partial segment graph (PSG) generation mode (and do not link); this generates a file with suffix .psg for the current translation unit; PSG files follow the DOT graph description language and can be processed with DOT file tools (e.g. displayed with the xdot.py tool); for 3rd-party IP components, PSG files may be edited with a text editor for further fine-tuning and IP protection;

–psg_input PSG file specifies the name of a PSG input file; the specified file will be loaded and its PSG will be integrated with the current translation unit to form a complete segment graph;

–psg_output output file in PSG generation mode (see above), this specifies the name of the PSG output file explicitly; by default, the output PSG file has the same basename as the input SystemC file;

–risc:dump output the computed segment graph (SG) and conflict tables as HTML files (default: no HTML files are generated); these files may be viewed by a user in a browser in order to inspect the out-of-order execution conditions of the model and improve it accordingly;

–risc:icpc use the Intel C++ compiler **icpc** in the backend for generating the executable (default: GNU C++ compiler **g++**);

–risc:mic use the Intel C++ compiler **icpc** with option *–mic* in the backend for cross-compiling an executable for the Intel Many Integrated Core (MIC) architecture (default: generate an executable for the same processor the compiler is running on);

–risc:elab filename import the elaboration result produced by the RISC elaborator **elab** from file *filename* and use it for segment conflict analysis based on a dynamic elaboration phase (default: pure static analysis);

–<*rose:option*> pass this option through to the underlying ROSE compiler (default: none);

–<*GNU option*> pass this option through to the underlying GNU compiler (default: none);

ENVIRONMENT

SYSTEMC_LW_HOME is used at compile-time to find the RISC light-weight SystemC header files which are expected in directory $SYSTEMC_LW_HOME/include (default: none);

SYSTEMC_OOP_HOME is used at compile-time to find the RISC out-of-order SystemC header files which are expected in directory $SYSTEMC_OOP_HOME/include, and the RISC out-of-order SystemC library which is expected in directory $SYSTEMC_OOP_HOME/lib (default: none);

SYSTEMC_MIC_HOME is used at compile-time to find the RISC SystemC header files and library files for the Intel many-integrated-core (MIC) architecture which are expected in directory $SYSTEMC_MIC_HOME/include and and $SYSTEMC_MIC_HOME/lib, respectively (default: none); this is used only when the option *–mic* is used (see above);

SYSC_PRINT_MODE_MESSAGE is used by the RISC simulator at run-time to print the mode of simulation and also the actual values of the environment variables listed below; these log lines start with "***" and are only printed when $SYSC_PRINT_MODE_MESSAGE is defined (default: no messages are printed);

SYSC_SYNC_PAR_SIM is used by the RISC simulator at run-time to force the RISC out-of-order SystemC simulation to fall back to synchronous (in-order) PDES execution; note that this mode is also automatically selected when SystemC primitive channels with update requests are used (default: out-of-order execution);

SYSC_PRINT_VERBOSE_MESSAGE is used by the RISC simulator at run-time to print debugging information about the simulator queues, event processing, and time advances; such debugging lines are only printed when $SYSC_PRINT_VERBOSE_MESSAGE is defined (default: no debugging infos are printed);

SYSC_DISABLE_PREDICTION is used by the RISC simulator at run-time to switch back to non-predictive conflict detection; this avoids scheduling overhead at run time, but usually results in slower simulation due to more conflicts; if $SYSC_DISABLE_PREDICTION is defined, thread state prediction is not used during out-of-order scheduling (default: out-of-order execution with prediction);

SYSC_PAR_SIM_CPUS is used by the RISC simulator at run-time to set the maximum number of concurrent threads allowed in the RISC out-of-order SystemC simulation (default: 64);

VERSION

The RISC compiler and simulator are release version 0.5.0.

AUTHORS

Zhongqi Cheng <zhongqc@uci.edu>, Rainer Doemer <doemer@uci.edu>, Guantao Liu <guantaol@uci.edu>, and Tim Schmidt <schmidtt@uci.edu>.

COPYRIGHT

LICENSE

BUGS, LIMITATIONS

This is an academic proof-of-concept prototype implementation, not commercial-quality software. See the file BUGS in the software packages for known limitations.

A.2 Manual Page of the RISC Elaborator

NAME

elab – Recoding Infrastructure for SystemC (RISC) Dynamic Elaborator

SYNOPSIS

elab *design* [*options*]

DESCRIPTION

elab is a special compiler for the SystemC language. The purpose of **elab** is to parse, analyze, instrument, and compile a SystemC source program into an executable program for dynamic elaboration. **elab** is a frontend source-to-source compiler for SystemC built on top of the ROSE compiler infrastructure with GNU or Intel C++ as backend target compiler. As such, **elab** relies on and supports also most of the ROSE and GNU compiler options.

Using the command syntax shown in the synopsis above, the specified *design* is compiled. By default, **elab** reads the SystemC source file, performs preprocessing and builds an internal representation (abstract syntax tree) of the SystemC structural hierarchy. **elab** then instruments the design model so that its execution stops after the end of the elaboration phase (no actual simulation will take place); the dynamically built hierarchy and instance connectivity data is then dumped into a file *design.elab* which can be passed to the RISC compiler **risc** for more precise conflict analysis.

On successful completion, the exit value 0 is returned. In case of errors during processing, an error code with a brief diagnostic message is written to the standard error stream and the compilation is aborted with an exit value greater than zero.

For preprocessing and C++ compilation into an executable file, **elab** relies on the availability of an external C++ compiler which is used automatically in the background. By default, the GNU C++ compiler **g++** is used.

ARGUMENTS

design specifies the file name of the input SystemC design model; by default, the base name of *design* is used as base name for the intermediate and output files;

OPTIONS

–h | *—-help* print the **elab** elaborator version and a brief usage information message to standard output and quit;

–v | *—-verbose* increment the verbosity level so that all tasks performed are logged to standard error (default: be silent); at level 1, high-level messages about the tasks performed are displayed; at level 2, additional details such as input and output file names are listed; at level 3, very detailed information about each executed task is printed;

–vv increment the verbosity level by two counts (same as *–v –v*);

–vvv increment the verbosity level by three counts (same as *–v –v –v*);

–w | *—-warnings* increment the warning level so that compiler warning messages are enabled (default: warnings are disabled); four levels are supported ranging from only important warnings (level 1) to pedantic warnings (level 4); for most cases, warning level 2 is recommended (*–w –w*);

–ww increment the warning level by two counts (same as *–w –w*);

–www increment the warning level by three counts (same as *–w –w –w*);

–g add a symbol table suitable for debugging (e.g. using **gdb**) to the generated object files and simulation executable (default: no debugging symbols);

–O | *–O level* optimize the generated simulation executable for higher execution speed and/or less memory usage (default: no optimization);

–Idir add the specified *dir* to the include path (extend the list of directories to be searched for including source files); include directories are searched in the order of their specification; the standard include path ($SYSTEMC_LW_HOME/include or $SYSTEMC_OOP_HOME/include) is automatically appended to this list; by default, only the standard include directories are searched;

–Ldir add the specified *dir* to the library path (extend the list of directories to be searched for linker libraries); the library path is searched in the specified order; the standard library path ($SYSTEMC_OOP_HOME/lib) is automatically appended to this list; by default, only the standard library path is searched;

–llib add the specified *lib* to the list of libraries for the linker so that the executable is linked against *lib;* libraries are linked in the specified order; the standard libraries (i.e. -lsystemc) are automatically appended to this list; by default, only standard libraries are used;

–c perform only the preprocessing, analysis, instrumentation, and compilation tasks; skip the final linking stage so that only an object file is created (default: perform all tasks including linking);

–o output file specify the name of the final output file explicitly (default: a.out);

–elab:o specify the name of the elaboration result file with instance connectivity data explicitly (default: *design.elab*); this file will be produced when the executable generated by **elab** is run (after its elaboration phase);

–<rose:option> pass this option through to the underlying ROSE compiler (default: none);

–<GNU option> pass this option through to the underlying GNU compiler (default: none);

ENVIRONMENT

SYSTEMC_LW_HOME is used at compile-time to find the RISC light-weight SystemC header files which are expected in directory $SYSTEMC_LW_HOME/include (default: none);

SYSTEMC_OOP_HOME is used at compile-time to find the RISC out-of-order SystemC header files which are expected in directory $SYSTEMC_OOP_HOME/include, and the RISC out-of-order SystemC library which is expected in directory $SYSTEMC_OOP_HOME/lib (default: none);

VERSION

The RISC Dynamic Elaborator is release version 0.5.0.

AUTHORS

Zhongqi Cheng <zhongqc@uci.edu>, Rainer Doemer <doemer@uci.edu>, Guantao Liu <guantaol@uci.edu>, and Tim Schmidt <schmidtt@uci.edu>.

COPYRIGHT

LICENSE

BUGS, LIMITATIONS

This is an academic proof-of-concept prototype implementation, not commercial-quality software. See the file BUGS in the software packages for known limitations.

A.3 Manual Page of the RISC SIMD Advisor

NAME

simd – Recoding Infrastructure for SystemC (RISC) SIMD Advisor

SYNOPSIS

simd [*options*] *design* [*options*]

DESCRIPTION

simd is an analysis tool for exploiting data-level parallelism based on the RISC compiler for the SystemC language. The purpose of **simd** is to parse and analyze a SystemC source program, and then provide advise to the user regarding possible optimizations of the model to exploit SIMD parallelism for faster out-of-order parallel simulation.

Using the command syntax shown in the synopsis above, the specified *design* is compiled and statically analyzed. By default, **simd** reads the SystemC source file, performs preprocessing and builds an internal representation (abstract syntax tree) of the SystemC constructs in the model. Next, thread control flow analysis is performed and encountered loops are analyzed for potential single-instruction-multiple-data (SIMD) execution which exploits data-level parallelism and can lead to significantly improved simulation performance in Out-of-Order Parallel Discrete Event Simulation (OoO PDES).

Specifically, **simd** presents to the user a list of loops that might be suitable for SIMD vectorization. The user is expected to review the options and, based on his application knowledge, select those loops that do not contain SIMD conflicts, such as parallel accesses to overlapping memory locations. For confirmed loops, the user then inserts into the source code **#pragma omp simd** annotations immediately before the selected loops. The annotated model can then be compiled with **risc** and option *–risc:icpc* using the Intel C++ compiler **icpc** to generate an executable for execution on a SIMD-capable target architecture with improved performance.

The output of **simd** lists the loops found in the control flow of the SystemC threads of the model. For each loop, its line number in the source code is listed together with its analyzed SIMD qualification. If the loop is not qualified, a reason for its disqualification may or may not be shown in form of an error code.

A qualification error code of 1 indicates the use of an invalid array index in the loop. The code 2 indicates that a non-loop local variable is written. Finally, code 3 indicates that an unsupported construct (e.g. goto statement) is found in the loop.

On successful completion, the **simd** advisor returns the value 0. In case of errors during processing, an error code with a brief diagnostic message is written to the standard error stream and the compilation is aborted with an exit value greater than zero.

ARGUMENTS

design specifies the file name of the input SystemC design model; by default, the base name of *design* is used as base name for the intermediate and output files;

OPTIONS

–h | *—-help* print the **simd** advisor version and a brief usage information message to standard output and quit;

–v | —-*verbose* increment the verbosity level so that the tasks performed are logged to standard error (default: be silent); at level 1, high-level messages about the tasks performed are displayed; at level 2, additional details such as input and output file names are listed; at level 3, very detailed information about each executed task is printed;

–vv increment the verbosity level by two counts (same as *–v –v*);

–vvv increment the verbosity level by three counts (same as *–v –v –v*);

–w | —-*warnings* increment the warning level so that warning messages are enabled (default: warnings are disabled); four levels are supported ranging from only important warnings (level 1) to pedantic warnings (level 4); for most cases, warning level 2 is recommended (*–w –w*);

–ww increment the warning level by two counts (same as *–w –w*);

–www increment the warning level by three counts (same as *–w –w –w*);

–Idir add the specified *dir* to the include path (extend the list of directories to be searched for including source files); include directories are searched in the order of their specification; the standard include path ($SYSTEMC_LW_HOME/include) is automatically appended to this list; by default, only the standard include directories are searched;

–o output file specify the name of the text output file explicitly (default: none);

–<*rose:option*> pass this option through to the underlying ROSE compiler (default: none);

–<*GNU option*> pass this option through to the underlying GNU compiler (default: none);

ENVIRONMENT

SYSTEMC_LW_HOME is used at compile-time to find the RISC light-weight SystemC header files which are expected in directory $SYSTEMC_LW_HOME/include (default: none);

VERSION

The SIMD Advisor is release version 0.5.0.

AUTHORS

Zhongqi Cheng <zhongqc@uci.edu>, Rainer Doemer <doemer@uci.edu>, Guantao Liu <guantaol@uci.edu>, and Tim Schmidt <schmidtt@uci.edu>.

COPYRIGHT

LICENSE

BUGS, LIMITATIONS

This is an academic proof-of-concept prototype implementation, not commercial-quality software. See the file BUGS in the software packages for known limitations.

A.4 Manual Page of the RISC Visual Tool

NAME

visual – Graphical SystemC Module Visualizer using RISC

SYNOPSIS

visual [*options*] *design* [*options*]

DESCRIPTION

visual is an analysis tool for graphical visualizing of ports and modules of SystemC code. It uses the RISC compiler to parse and analyze the SystemC source code into a data structure. The tool iterates through this data structure and displays a visual representation of the hierarchy of modules and ports. **visual** provides a GUI to provide a graphical representation of the SystemC model as well as provide user modifiable options during run-time to change the graphical properties of the visualization.

ARGUMENTS

design specifies the file name of the input SystemC model.

OPTIONS

–*h* | —-*help* prints a brief message on the usage of the tool to standard output and quits;

–*bw* Modules are drawing without color;

–*tm module* Only draw ”module”;

–*Il integer* Draw only a certain depth in the hierarchy given by ”integer”;

–*s float* Scale the drawing size by ”float”. If ”float” = 0.5, then the size of the drawing is scaled by 50 percent.

–*np* The module hierachy will be drawn without ports or channels;

ENVIRONMENT

SYSTEMC_LW_HOME is used at run-time to find the RISC light-weight SystemC header files which are expected in directory $SYSTEMC_LW_HOME/include

VERSION

Visual is release version 0.5.0.

AUTHORS

Daniel Mendoza <dmmendo1@uci.edu>

COPYRIGHT

LICENSE

BUGS, LIMITATIONS

This is an academic proof-of-concept prototype implementation, not commercial-quality software. GTK is used at compile-time for the GUI. CAIRO is used at compile-time for drawings displayed on the GUI.

A.5 Manual Page of the RISC Tree Tool

NAME

tree – Textual SystemC Module Visualizer using RISC

SYNOPSIS

tree [*options*] *design* [*options*]

DESCRIPTION

tree is an analysis tool for textual visualizing of ports and modules of SystemC code. It uses the RISC compiler to parse and analyze the SystemC source code into a data structure. The tool iterates through this data structure and displays a visual representation of the hierarchy of modules and ports.

ARGUMENTS

design specifies the file name of the input SystemC model.

OPTIONS

–h | *—-help* prints a brief message on the usage of the tool to standard output and quits;

ENVIRONMENT

SYSTEMC_LW_HOME is used at tun-time to find the RISC light-weight SystemC header files which are expected in directory $SYSTEMC_LW_HOME/include

VERSION

Tree is release version 0.5.0.

AUTHORS

Daniel Mendoza <dmmendo1@uci.edu>

COPYRIGHT

(c) 2018 CECS, University of California, Irvine

LICENSE

Open source BSD license terms apply.

BUGS, LIMITATIONS

This is an academic proof-of-concept prototype implementation, not commercial-quality software.