Enabling IP Reuse and Protection in Out-of-Order Parallel SystemC Simulation

Zhongqi Cheng, Tim Schmidt and Rainer Doemer

University of California at Irvine, USA

Abstract. Parallel discrete event simulation has presented itself to be a tempting approach for high speed SystemC simulation. To preserve the simulation semantics, a compiler based approach statically analyzes race conditions in the design model. However, there are severe restrictions: the 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. In this paper, we extend the static analysis design flow to support separate files and IP reuse by introducing Partial Segment Graph (PSG) abstraction and prevent IP security leakage. Experiments demonstrate the effective design flow and sustained speedup with parallel IPs.

Keywords: Out-of-order PDES · Intellectual Property · SystemC.

1 Introduction

The complexity of system design has been growing with the increasing functionality of modern embedded systems. As a system level design language, the IEEE SystemC standard [1] is widely used for testing, validation and verification of system level models. The proof-of-concept Accellera SystemC simulator [2] is based on Discrete Event Simulation (DES) and runs sequentially. In contrast, Out-of-order Parallel Discrete Event Simulation (OoO PDES) [3] can exploit the parallel computation of modern multi- and many-core platforms. In OoO PDES, threads comply with a partial order such that different simulation threads may run in different time cycles to increase the parallelism of execution.

The Recoding Infrastructure for SystemC (RISC) [4] has been developed to implement OoO PDES for SystemC. RISC includes a dedicated compiler and an OoO PDES library. The RISC compiler is the frontend that processes the input SystemC file. It first builds the Abstract Syntax Tree (AST) of the input file and then derives from the AST the *behavior model (BM)* of the input SystemC design. With BM available, the RISC compiler then performs static analysis regarding potential thread race conditions of the design.

The BM is an abstraction of the execution of the SystemC processes in the design. The RISC compiler represents BM with a statically built Segment Graph (SG) data structure. Based on the SG, the RISC compiler is able to analyze the data conflicts, timing conflicts and event hazards in the design.

1.1 Problem Definition

To completely build the BM of the input SystemC design, the RISC compiler needs the entire AST for the input model. Thus the user has to provide all the source code in one single translation unit. In other words, the RISC compiler cannot build BM for SystemC designs whose source code is separately structured in multiple source files or 3rd party Intellectual Properties (IP). With the wide use of IP, this requirement severely restricts the RISC compiler to meet industrial system level design needs.
In this paper, we propose a solution that scales the RISC compiler to support multiple file inputs, especially for the integration of IPs, as shown in Figure 1. In the new design flow, the construction of BM no longer relies on the complete AST. Besides the usual object and header files, component providers supply a *partial design (PD)* file that abstracts the BM of integrated design components. Specifically, in the PD file, the BM is abstracted by a *Partial Segment Graph (PSG)*. IP providers can inspect and redact the PD file, in order to further minimize the PSG, which protects the security of their IP. On the user's side, by combining all the received PSGs, the RISC compiler is able to reconstruct the BM of the whole design.

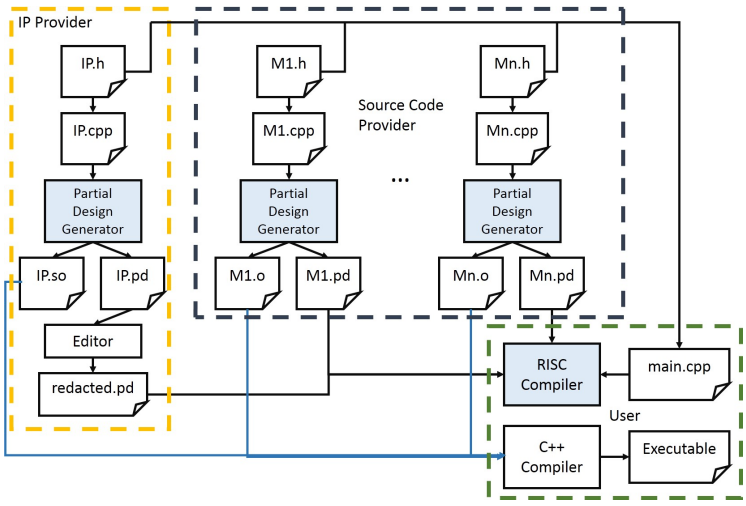


Fig. 1: Scaled RISC tool flow with IP components

1.2 Related Work

IP reuse and protection have not received a lot of attentions in parallel simulation. In [5], the authors described an effective methodology for IP reuse in SOC design. They studied the IP enhancement and also proposed a framework for the

reuse of customer IP.
Parallel SystemC simulation is well-studied. In [6], the SystemC-clang is proposed. It analyzes SystemC models with a mixture of transaction-level and register-transfer level components. In [7], the authors studied the distributed parallel simulation, where SystemC models are organized into small executable units and distributed onto different host machines to run in parallel.
In [8], the authors proposed a way to use pre-defined graphs to represent the BM of IP components. However, this simple approach requires the users to manually analyze the design and insert pragmas where needed. Furthermore, there are only three kinds of predefined graphs, which is insufficient. In contrast, we propose PSG as the data structure to represent the BM of IP components, which is accurate and is automatically built by a compiler.

2 Partial Segment Graph

We now describe the PSG technique that represents the BM in each seperate translation unit.

2.1 Behavior Model and Segment Graph

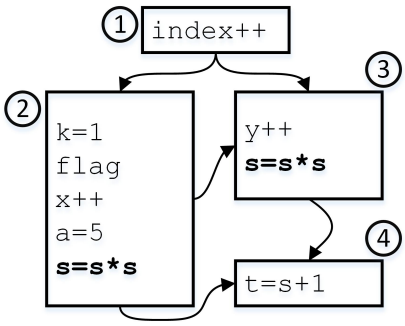
The behavior model of a SystemC design can be described by the Segment Graph, which provides a way to analyze threads and their position during execution. The SG is a directed graph where each node is a sequence of code statements executed between two scheduling steps, i.e., `wait` statements [3]. During the execution of the model, the scheduling step is the entry to the simulator kernel. The edges in the SG indicate the transition between segments. An example of SystemC source code and corresponding SG is shown in Figure 2a and Figure 2b.
In this example, line 8 `y++` and line 12 `s=s*s` could possibly be executed in the same simulation cycle, so they are put both into *segment_3*. One statement may also belong to multiple segments as it may occur in different cycles. Both *segment_2* and *segment_3* contain `s=s*s`. Note that a new segment starts only on `wait` statements except for the first one, which is the entry point of a thread.

2.2 Concept of PSG

In our proposed design flow, we store the BM specified in each translation unit as a PSG in a PD file and when the PSGs are loaded and integrated together, they reconstruct the complete SG.
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. An example is shown in Figure 3a. It contains only the definition and implementation of module `M`. Function `p->func()` is called in `M::th()`, but it is not defined in this translation unit. We refer to a function call that lacks the definition as

```
1  void foo(){
2    index++;
3    wait(2,SC_NS);
4    k=1;
5    if(flag){
6      x++;
7      wait(10,
             SC_NS);
8      y+;
9    }else{
10     a=5;
11   }
12   s=s*s;
13   wait(1,SC_NS);
14   t=s+1;
15 }
```

(a) Example Source Code



(b) SG of Fig. 2a

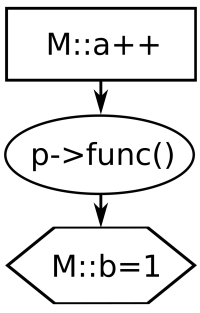
Fig. 2: SystemC Code and corresponding SG

a *non-defining function call*. Because the compiler cannot determine from the current AST if a non-defining function call contains scheduling steps or not, the simulation cycle of the code statements following the non-defining function call cannot be statically determined. In the example, we cannot know if line 5 and line 7 execute in the same cycle.
To deal with this uncertainty incurred by the non-defining function calls, we introduce three types of PSG nodes:

– *Segment node* contains a sequence of code statements executed in the same determined simulation cycle. In Figure 3a, `M::a++` belongs to a segment node because its simulation cycle is determined, which is the first cycle of

```
1  SC_MODULE(M){
2    ...
3    sc_port<C> p;
4    void th(){
5      M::a++;
6      p->func();
7      M::b=1;
8    }
9    ...
10 }
```

(a) Example Source Code



(b) PSG of Fig. 3a

Fig. 3: SystemC Code and PSG

the `sc_thread M::th()`. A segment node becomes a segment after the integration of PSGs.
- *Partial segment node* contains a sequence of code statements executed in the same non-determined simulation cycle. In Figure 3a, `M::b=1` belongs to a partial segment node because it is executed after the non-defining function call `p->func()`. Later during the PSG integration phase, the partial segment node will be merged with other segment nodes.
- *Partial function call node* is created as a place holder for the non-defining function call in the PSG such that during the PSG integration phase, the partial function call node can be replaced by the sub-PSG corresponding to the function's definition. In Figure 3a, node 3 is a partial function call node for the non-defining function call `p->func()`.

2.3 Create PSG

A PSG is recursively built by traversing the AST of the current translation unit, as shown in Algorithm 1. If the current statement `CurrStmt` is a scheduling entry point (`wait` statement), then an empty segment node is created and connected to the nodes in the `CurrNodes`. On the other hand, if `CurrStmt` is not a scheduling point, then it is added to all the nodes in `CurrNodes`. This is similar as in the *BuildSG* in [8]. The main difference is that to build PSG, the compiler also needs to deal with non-defining function calls. If `CurrStmt` contains a non-defining function call, for example `f()`, the compiler first builds a partial function call node `NewNode` and stores the qualified name `M::f()`. Next, the compiler connects `NewNode` to all the nodes in `CurrNodes`. Then, a partial segment node `NextNode` is created and connected to `NewNode`, and the compiler sets `NextNode` as the only node in the `CurrNodes`.

2.4 Store and Load PSG

The PD file stores an abstraction of the PSG. For each node, we omit the detailed code statements and store only the access types (R,W,RW) to non-local variables. This is sufficient for the RISC compiler to analyze the data and event conflicts. In addition, some meta-data is stored for each node, which is needed for the integration of PSGs, as listed in Table 1. Note that the PD file is compatible with dot format and the PSG therefore can easily be visualized. An example of PSG is later shown in Figure 8.
A PSG is loaded from the PD file with a dot file parser. The parser reads the attributes of each PSG node, and reconstructs the data in memory. For example, a node has a variable access attribute `(W)M::a`, which indicates that `M::a` is been written in the node. To load the node into memory, the PSG parser locates the symbol of `M::a` in the AST and puts it into the *variable_write_list* of the node. PSG edges are constructed according to the connections specified in the PD file. After the loading of individual PSGs, the compiler integrates them together to construct the complete SG.

Algorithm 1 Partial Segment Graph Generation

```
 1: function BuildPSG(CurrStmt, CurrNodes)
 2:     if isBoundary(CurrStmt) then
 3:         NewNode ← new segmentNode
 4:         for Node ∈ CurrNodes do
 5:             AddEdge(Node, NewNode)
 6:         end for
 7:         return CurrNodes ∪ { NewNode }
 8:     else if isNonDefiningFunctionCallStmt(CurrStmt) then
 9:         NewNode ← new partialFuntionCallNode
10:         Mark(NewNode, getFuncName(CurrStmt))
11:         for Node ∈ CurrNodes do
12:             AddEdge(Node, NewNode)
13:         end for
14:         NextNode ← new partialSegmentNode
15:         AddEdge(NewNode, NextNode)
16:     else if isControlFlow(CurrStmt) then
17:         BuildSG(CurrStmt, CurrNodes)
18:         ...
19:     end if
20: end function
```

2.5 Integration phase

A complete segment graph is the basis for accurate static analysis. After loading all the PSGs, first the partial function call nodes are recursively replaced with the corresponding sub-PSG. Second, all the partial segment nodes are merged with segment nodes they follow. All remaining nodes in the graph are segment nodes (with underlying `wait` boundaries) and belong to determined simulation cycles, such that the integrated graph by definition becomes a proper segment graph. With the reconstructed SG, the RISC compiler has the complete the BM and can perform the needed static analysis of the design.

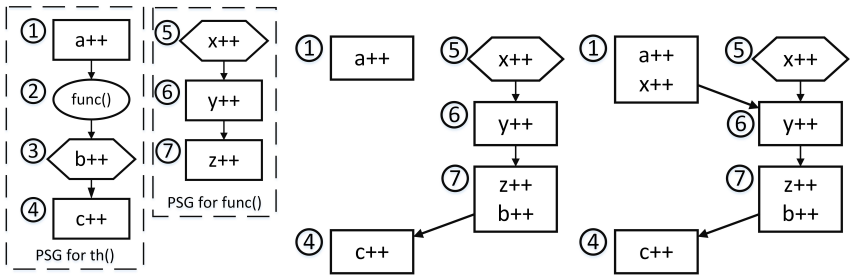
We illustrate the merging process of two PSGs in Figure 4a, 4b and 4c. In this example, *node_2* is a partial function call node that holds the non-defining function call `func()`, and *node_5*, *node_6* and *node_7* are loaded from the psg in func.pd and forms the sub-PSG of `func()`. *node_5* and *node_7* are respectively the entry and exit node of `func()`. First, *node_3* is merged into *node_7* because they belong to the same simulation cycle. After merging, *node_4* is connected to *node_7* since it was connected to *node_3*. Then, *node_5* is merged into *node_1* because it is the starting node of `func()`. *node_6* is connected to *node_1* since it was connected to *node_5*.

3 IP Protection and Security

IP reuse is an important feature in semiconductor industry. Basically, an IP consists of two parts: a header file that describes the interfaces and protocols,

Table 1: PSG meta-data node attributes

Attribute	Description
Node type	Segment node, Non-segment node, Function call node
Written variables	Qualified name of variables written
Read variables	Qualified name of variables read
Notified events	Qualified name of events notified
Dependent events	Qualified name of events waiting for
Hosting function name	Qualified name of function belonging to
Hosting module name	Qualified name of module belonging to
Is entry node	Marker for function entry point
Is exit node	Marker for function exit point
Is simulation process	Marker for simulation process
Non-defining function name	Qualified name of non-defining function



(a) Two PSGs (b) Phase 1 (c) Phase 2

Fig. 4: Integration of PSGs

and a binary file that implements the IP component. Since no implementation source code is provided, the IP is protected. The internal BM of the IP is hidden from the users.

However, static analysis cannot be performed without the BM. We need the IP provider to supply an abstract PSG of the IP to the user via PD files. For the IP provider not to reveal too much implementation detail and to solve this IP security leakage problem, we allow the IP provider to redact the PSG in the PD file, so that the implementation details remain hidden. If desired, misleading information can even be added. This way the users will not be able to obtain the inner implementation, while still maintaining the correctness of BM.

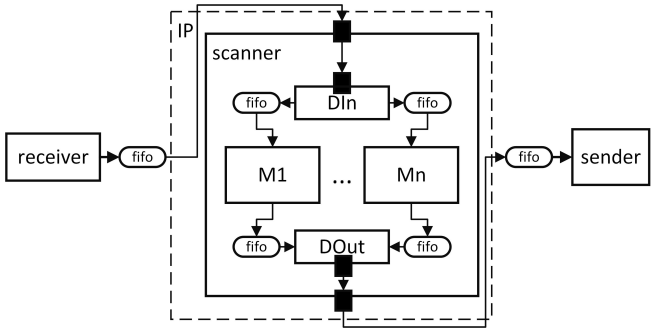
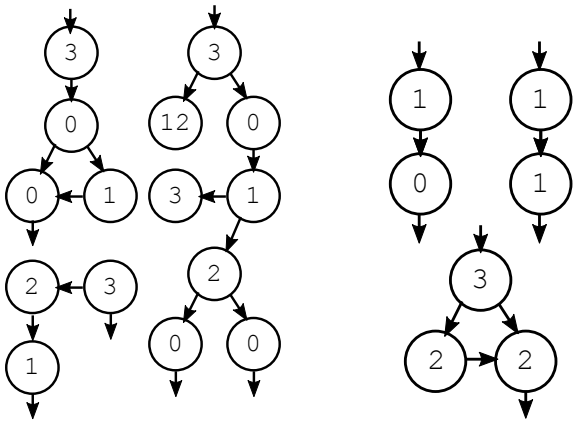


Fig. 5: SystemC model of Bitcoin miner



(a) Original (b) Redacted

Fig. 6: Original and redacted scanner.pd

Figure 5 shows the SystemC model of a Bitcoim miner [9]. It has several user defined modules (receiver and sender) for data input and output, and uses an IP module (scanner) for number crunching. Three PD files (*receiver.pd*, *sender.pd*, *scanner.pd*) contain the corresponding BM of each module.

By default, in each PD file the RISC compiler stores (1) the qualified name of variables accessed and access types, (2) qualified name of events and dependencies, (3) PSG structure and timing advance. For *receiver.pd* and *sender.pd*, it is fine to have such information transparent because the two modules are user-defined. However, for *scanner.pd*, exposing the internals is risky from the perspective of IP protection.

The original and redacted versions of scanner.pd are shown in Figure 6. Here the numbers in the PSG nodes indicate the number of variable accesses stored. Compared to the original, the revised file has fewer nodes, and each node has fewer variable accesses stored. These modifications are carefully performed such

that during the simulation, the model still executes correctly.

3.1 Redaction of PD files for IP Protection

There are several possible changes that can be performed to redact the PSG:

1. Reduce the amount of variable accesses: If two nodes share more than one internal variable accesses, only one of them needs to be kept and others can be removed from the two nodes. This does not change the data conflict of the two segments. Only externally visible variables need to be retained. Furthermore, if a variable is read only in any node, it can be removed because it cannot lead to data conflicts.
2. Add fake variable access: By adding extra variables to a node, the IP provider can further obscure the IP and inject misleading details.
3. Hide nodes: An entire node that contains no variable accesses or event notifications can be hidden from PSG because it does not affect the static analysis. To maintain the timing correctness, the incoming and outgoing edges need to merge.
4. Merge segment nodes: Segment nodes can be merged to form an aggregation. This effectively hides the detailed PSG structure. The downside is that merging may pollute conflict-free segment pairs. Two code statements that actually can run in parallel after merging run only sequentially because they now belong to two conflicting aggregated nodes.

In general, there is a trade-off between the amount of IP protection versus the analysis accuracy, which may affect simulation performance.

4 Experimentats and Results

We first show the correctness of the proposed design flow using a simple producer-consumer example and a more complex Canny edge detector model. Then we demonstrate our IP protection using a SystemC model of Bitcoin miner, where we designed an IP for the parallel data crunching module and redacted the PD file to hide details in the PSG. Our experiments were executed on an Intel Xeon E3-1240 multicore processor with 8 CPU cores. The CPU frequency scaling was turned off so as to provide accurate and stable results.

4.1 Producer-Consumer Example

In the producer-consumer model, we have defined and implemented each module/channel in individual files. According to the tool flow in Figure 1, we first generate the PD file for each translation unit. At top level, RISC integrated the PSGs. The regular RISC tool flow without PD support cannot handle this multi-file design and generates an error message. Now with the proposed approach, RISC is able to correctly perform static analysis and generates a functioning parallel executable.

4.2 Canny Edge filter

The Canny edge detector algorithm is a multi-stage operator that detects edges in an image. Our SystemC model has a pipeline structure with five stages, and each stage communicates with the next via user-defined channels. We have defined and implemented all stage modules and channels in different translation units. In this experiment, we have a total of 15 implementation files and corresponding PD files. The sequential simulation runs for 280.90 seconds, and the parallel one runs for 116.48 seconds, achieving a speedup of 2.41x.
Without the proposed technique, the RISC compiler generates an error and cannot compile. With the proposed design flow, the RISC compiler is able to construct the BM of the complex design and correctly perform static analysis, and gains speedup for the simulation.

4.3 Bitcoin miner

We have implemented a Bitcoin miner model in SystemC to demonstrate the IP protection capability. Our model consists of 3 stages: data preparation, scanning and result output, as shown in Figure 7. The scanning stage involves the use of multiple parallel scanners, which runs *SHA256* algorithm are provided as IP. In order to protect the IP, we have inspected *scanner.pd* and carefully redacted the PSG. The graphical view of the default and the redacted PD files of scanner are shown in Figure 8. We have removed several variable accesses because they do not actually result in conflicts. Furthermore, We have redacted the structure of PSG by removing an empty node. The new PSG is smaller and hides information about the detailed implementation of the scanner.

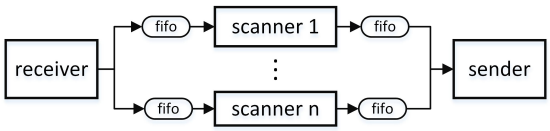


Fig. 7: SystemC model of Bitcoin miner

Table 2 shows the simulation speed of Bitcoin miner model with different number of scanners under both sequential and parallel simulations, using the default and redacted PD files of the IP. We have performed this experiment on a Xeon E1240 8-core processor. The speedup of the parallel simulation grows about linearly with the number of scanners. When there are 8 scanners, we get the maximum speedup of 6.76x. A full speedup of 8 cannot be achieved because of the sequential part in the model and the scheduling overhead of OoO PDES. Another important observation is that the simulation result and speed does not change with the IP scanner. This demonstrates that our information hiding approach works well for IP protection.

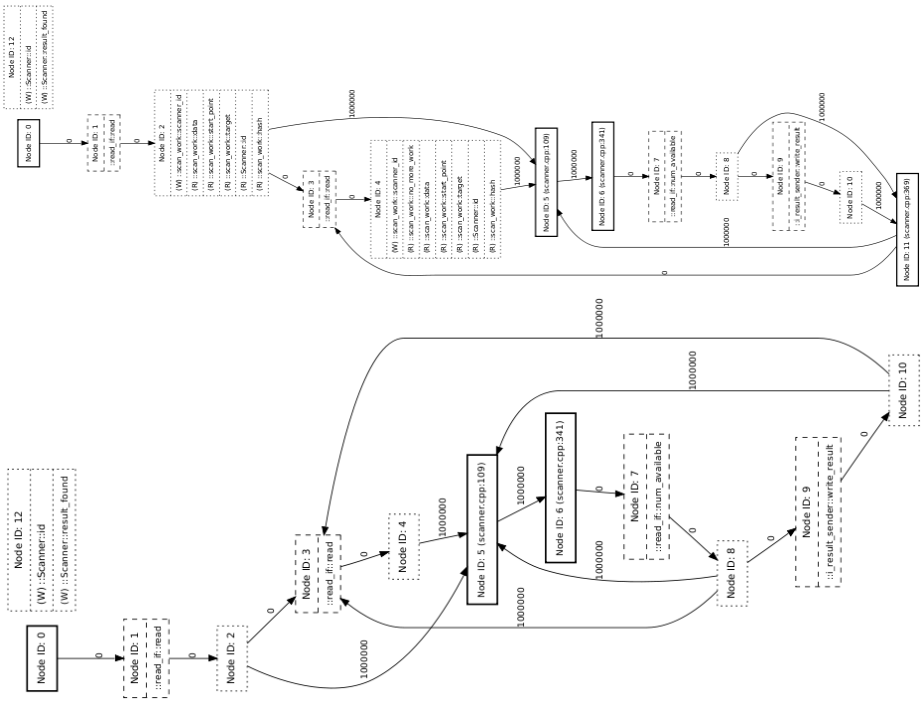


Fig. 8: Original and modified PSG for scanner

Table 2: Simulation of Bitcoin Miner SystemC Model: runtime(secs)/speedup

#scanner	SEQ	Original	Modified
1	117.68 / 1	117.69 /1.00	117.51/1.00
2	114.96 / 1	86.90 / 1.32	87.2/1.32
4	158.00 / 1	49.08 / 3.22	50.11/3.15
8	164.50 / 1	24.91 / 6.60	24.32/ 6.76

5 Conclusion

This work removes two scaling limitations of static-analysis based parallel SystemC simulation. The new Partial Segment Graph (PSG) techniques enable the use of hierarchical input models with multiple translation units and the reuse of SystemC IP components. 3rd party IP is protected from security leakage by high abstraction from the source code, using automatically generated behavior model that the IP provider can further redact to meet trust expectations. The IP-enabled design flow is effective and sustains the speedup of advanced parallel

simulation.

Acknowledgement

This work has been supported in part by substantial funding from Intel Corporation for the project titled "Scaling the Recoding Infrastructure for Parallel SystemC Simulation". The authors thank Intel Corporation for the valuable support.

References

1. IEEE Standard 1666-2011 for Standard SystemC® Language Reference Manual, IEEE Computer Society, January 2012.
2. SystemC Language Working Group. (2014). SystemC 2.3.1, Core SystemC Language and Examples, Accellera Systems Initiative.
3. W. Chen, X. Han, C. W. Chang, G. Liu, and R. Dömer. Out-of-Order Parallel Discrete Event Simulation for Transaction Level Models. IEEE TCAD, 33(12):1859-1872, 2014.
4. Lab for Embedded Computer Systems (LECS). Recoding Infrastructure for SystemC [Online]. Available: `http://www.cecs.uci.edu/~doemer/risc.html#RISC050`
5. S. Sarkar, S. Chanclar G and S. Shinde, Effective IP reuse for high quality SOC design, IEEE SOCC, 2005, pp. 217-224.
6. Kaushik A, Patel HD, SystemC-clang: an open-source framework for analyzing mixed-abstraction SystemC models. FDL, Paris 2013
7. J. Viitanen, P. Sjvöall, M. Viitanen, T. D. Hämäläinen, and J. Vanne. Distributed SystemC simulation on manycore servers. In IEEE NORCAS, pages 1-6, 2016.
8. T. Schmidt, G. Liu, R. Dömer. Hybrid Analysis of SystemC Models for Fast and Accurate Parallel Simulation, ASPDAC, Tokyo, Japan, January 2017.
9. Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. 2008.