Static Analysis Method for Deadlock Detection in SystemC Designs

Alexey Zakharov, Mikhail Moiseev
Department of Computer Systems
and Software Engineering
Saint-Petersburg State Polytechnical University
Saint-Petersburg

Ilya Klotchkov, Sergey Salishev
Intel Labs
Saint-Petersburg

Abstract

One of the goals of SystemC is high level system design verification at the early stage. Currently, simulation is widely used for this purpose. As the level of design parallelism grows, efficiency of simulation-based verification methods decreases. Thus different formal verification methods for SystemC are actively researched.

In this paper we present an approach to deadlock detection in SystemC designs based on static code analysis. Our approach to static analysis considers SystemC scheduler semantics. The developed approach has been implemented in Deadlock Analyzer tool. We demonstrate efficiency of our tool by applying it to dining philosophers, crossroads, producer-consumer cases and to a real-life model of video accelerator.

1 Introduction

System level modeling has become an essential part of SoC (System On Chip) design process. SystemC [1] is an IEEE standard for system-level software/hardware co-design. SystemC is very popular in the industry as system level modeling tool.

A typical SoC design consists of a hardware part and a software part running on a CPU. The system model will thus include models for software, hardware and their interfaces and is highly parallel. Due to high complexity of real system designs they might contain errors. Debug and verification of such designs is often done by simulations. However synchronization error detection in highly parallel programs is known to be difficult due to the exponential state space growth. Another approach to error detection is use of static methods. Problems of exact detection of race condition and deadlock are known to be algorithmically unsolvable in general case and NP-complete for bound execution graphs [16, 17]. Some approximation methods with reasonable complexity are used in practice. There are several static methods used in software development which are aimed at detecting synchronization errors. These methods allow to increase the system state space coverage comparing to simulation and thus to detect more errors in the system design at early stage.

In this paper we will focus on finding deadlocks in SystemC designs. Deadlocks are selected due to their high debug complexity and dramatic consequences for the system - in SOC deadlock usually means partial or full lockout and reset. Furthermore, limited on-chip observability makes deadlock detection in real silicon hard and unpredictable. We have selected static analysis because it allows sound deadlock detection at an expense of some false positives.

The contribution of this paper is a novel method for deadlock detection in SystemC designs based on the static code analysis. This method considers SystemC scheduler semantics. It supports immediate notifications, delta notifications and channel updates.

The rest of this paper is organized as follows: Section 2 briefly presents SystemC library followed by program model description in 3 section. Section 4 describes analysis algorithms used for the SystemC code and section 5 presents deadlock detection rules. Efficiency of the proposed approach is discussed in section 6 and the last section concludes the paper.

1.1 Related Work

The SystemC program is a parallel C++ program. So it’s natural to apply formal methods used in C++ software verification for the synchronization error detection in SystemC. There are several main classes of formal methods that can be applied to SystemC [22].

Error detection boils down to checking some constraints on a program state space. These constraints
can be predefined for certain types of errors or derived from user defined assertions. These assertions can be written using program model formalism or in property specification language [8, 19]. The latter approach has been successfully used for interface protocol verification.

The first step of error detection is extracting the program model. This can be done dynamically or statically. The run-time approach captures the program execution traces including events and constraint sensitive components of program state. In order to apply the run-time method to SystemC, the simulation kernel must be modified [3, 20]. This method can only capture errors that could potentially happen on the execution trace, so its accuracy heavily depends on the program test coverage [9].

Static approach extracts the program model directly from a program source code. Static methods can usually handle only a limited subset of SystemC language [15]. The SystemC kernel is usually abstracted at this stage by replacing it with annotations [10, 6] or with a simpler C/C++ stubs [7] to reduce the complexity of subsequent analysis. In the latter case it’s possible to apply tools for C/C++ to the modified program directly.

A mixed method of program model extraction is implemented in PinaVM [14]. System design is extracted using runtime information at elaboration phase. This information allows to improve static analysis precision for virtual methods and pointers. The tool is based on LLVM for C/C++ language.

The program model formalism heavily depends on the error detection methods to be applied. Possible formalisms include Control Flow Graph, Dynamic Dependency Graph [3] Petri nets [13, 4], ISO Lotos [10, 6], PROMELA [21], mCRL2 [11], etc. After the program model is extracted, a formal analysis method is applied to it. Model checking and static code analysis are of the most practical interest among different analysis methods.

Model checking methods are based on proving some exact model properties. These are popular for SystemC programs analysis [4, 3, 13, 9, 2, 10]. The approach has limited applicability for large parallel systems due to the state space explosion [5] that leads to unfeasible memory and computational requirements.

Methods of static code analysis operate on program model approximations exhibiting modest resource requirements at the cost of false-positives. This approach has been used in [12] to extract certain characteristics of SystemC program for measuring its parallelism and testing complexity. The static code analysis is known to provide good results for C++ error detection. We conclude that the potential of static code analysis for SystemC is yet to be fully explored.

2 SystemC Fundamentals

SystemC program consists of a number of modules connected via interfaces. Each module is represented by sc_module class and can contain processes of SC_METHOD, SC_THREAD or SC_CTHREAD type. SC_METHOD is executed once upon reception of a trigger event and can not have waits inside. SC_THREAD and SC_CTHREAD can contain wait’s. SC_CTHREAD is clocked version of SC_THREAD.

Thread execution is controlled by the SystemC simulation kernel and follows cooperative multitasking model. Distinct feature of SystemC kernel is an explicit notion of time – the kernel keeps track of the model time. Time is associated with every event in the model thus allowing to simulate very fine grain details of models’ behaviour. In order to distinguish events with the same time stamp, notion of delta time (or delta cycle) is introduced. Delta cycle number is assigned to each active event in the order of their appearance at any given model time and cleared after the time is advanced. All events within the same delta cycle are assumed independent and their execution order is not specified. The SystemC kernel doesn’t advance delta cycles or time until all ready threads that can resume in the current delta cycles have been resumed. It’s technically possible to use only delta time for modeling of systems for which event ordering is sufficient.

Since SystemC execution model is cooperative multitasking, a thread can only be suspended if it yields control explicitly either by executing return statement or by calling wait. If thread encounters a wait call, it blocks until the wait condition is satisfied. Wait conditions come in two forms: time wait and event wait. Time wait forces thread to sleep for given time, event wait does the same until designated event(s) is received. Upon receiving control, the kernel then decides which of blocked threads to resume based on their wait conditions and current delta cycle.

Interprocess synchronization in the SystemC is limited to wait-notify handshake and doesn’t have any mutable objects or critical sections. Whenever a thread needs to send an event to another thread, it calls notify function with an appropriate event to send. Global nature of delta cycles in SystemC ensures events’ ordering. Wait-notify synchronization is the only one provided by SystemC kernel and doesn’t include any data transfers or code critical sections and
thus is substantially different from the generic software execution. There are however several data-oriented channels which encapsulate data transfer behaviour – all of them are based on a data storage of some form which access function transparently performs a wait-notify synchronization with it’s peers.

3 SystemC Program Model

The deadlock detection method is applied to the formal program model, which is convenient for analysis algorithms. The program model is extracted from SystemC program sources. It is built in the form of control flow graph with the following properties:

- All assignments are in 3-operand form.
- There is exactly one declare and one undeclare statement for each variable in the program.
- Control statements are limited to if and goto statements.
- A condition of if statement is a single variable.
- Special $\phi$-function statements are used when different control flows are joined together.
- Some function calls are replaced with annotations that may include domain-specific statements.

The program model contains SystemC-specific statements which are listed in the table 1.

<table>
<thead>
<tr>
<th>Table 2: Representation of SystemC Statements</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Source Code</strong></td>
</tr>
<tr>
<td>run(this, t, f)</td>
</tr>
<tr>
<td>sensitive &lt;&lt; e1 &lt;&lt; e2;</td>
</tr>
<tr>
<td>request_update();</td>
</tr>
<tr>
<td>sc_start();</td>
</tr>
<tr>
<td>wait();</td>
</tr>
<tr>
<td>wait(e);</td>
</tr>
<tr>
<td>wait(e, -1);</td>
</tr>
<tr>
<td>wait(e, 0);</td>
</tr>
<tr>
<td>sc_time t(0, SC_NS);</td>
</tr>
<tr>
<td>wait(t);</td>
</tr>
<tr>
<td>notify(e);</td>
</tr>
<tr>
<td>notify(e, 0);</td>
</tr>
<tr>
<td>sc_time t(0, SC_NS);</td>
</tr>
</tbody>
</table>

Process creation statements, sensitivity list management statements and interprocess synchronization statements are all expressed with statements from table 1. The representation of main SystemC statements is shown in the table 2.

The program model and analysis algorithms have some limitations:

- Only immediate and delta notifications are supported (time parameter in wait and notify statements can be -1 or 0 only, see table 2).
- Spawning processes are not supported. Statement sc_spawn, macros SC_FORK and SC_JOIN are not presented in program model.
- Dynamic processes are not supported. Call-back methods of before_end_of_elaboration, end_of_elaboration and other are not supported.
- Method processes are not supported. Function next_trigger calls are not presented in program model.
- Clocked thread processes are not supported. Function reset_signal_is calls are not presented in program model.
- Special scheduler control functions such as sc_stop, sc_time_stamp and other are not supported.

4 Analysis

SystemC design is a parallel C++ program from analysis point of view. Control-flow of an arbitrary SystemC program may depend on values of scalar variables as well as on values of pointers, class members and predicates on them. Since synchronization statements may occur in the branch of if, switch or loop statement it is necessary to evaluate possible values of branch condition variables. In order to obtain possible values of scalar variables interval analysis is used.
SystemC events, ports and other essential entities are usually class members of sc_module. These objects are implicitly accessed via this pointer in the source code, so points-to analysis is required to deal with synchronization. Widespread use of virtual methods in SystemC library is another reason for points-to analysis.

Both interval and points-to analyses provide auxiliary information to reason about program synchronization and possible parallel executions.

Proposed static analysis approach follows the SystemC scheduler semantics defined in SystemC language standard and includes two stages:

- elaboration phase analysis,
- delta-cycle analysis:
  - evaluation phase analysis of current delta-cycle,
  - update phase analysis of current delta-cycle.

4.1 Interval and Points-to Analyses

Both interval and points-to algorithms are specific implementations of well-known dataflow static analysis approaches [18]. These approaches presume creation of dataflow equations for each statement of a program and solving the equations until fixed point is reached. The solution includes possible values of variables.

The points-to algorithm uses representation of program state in the form of points-to relation. Each program object is a tuple \((o, k)\), where \(o\) is an area of memory, \(k\) is an offset within this object. Points-to relation includes tuples \(((o_1, k_1), (o_2, k_2))\), where \((o_1, k_1)\) points to \((o_2, k_2)\). The algorithm considers dynamically allocated objects, stack-allocated objects and special objects that represent uninitialized value of a pointer and NULL pointer. This relation is able to express both direct and indirect memory addressing and to represent objects of complex types.

Interval analysis algorithm operates on tuples in the form \(((o, k), i)\), where \(i\) is an interval value which is an ordinary interval \([low, high]\) or a special interval that represents uninitialized value.

Static analysis algorithms discussed in this section are sound, context- and flow-sensitive. These features allow high precision and soundness of deadlock detection.

4.2 Elaboration Phase Analysis

Elaboration phase includes the following activities:

- module building and initialization,
- construction of unspawned processes,
- setting static sensitivity lists of processes.

Analysis of the elaboration phase involves interval and points-to analyses to extract structure of SystemC program, e.g. processes, their start methods, events, process sensitivity, possible values of global variables.

The elaboration phase covers analysis of a global initializer and analysis of entry function sc_main. The global initializer is a block of statements representing initialization of global variables. Analysis of SystemC program entry function ends when start statement is met. The result of elaboration phase is a program state that contains:

- values of global variables and sc_main function local variables,
- set of constructed processes,
- start method for each process,
- static sensitivity lists of processes.

4.3 Evaluation phase analysis

Evaluation phase analysis begins with a non-empty set of start points. A start point is a program point which is active in this evaluation phase. A program point is \((s, p, C, I)\) tuple, where \(s\) – statement, \(p\) – process, \(C\) – call stack, \(I\) – cycle iteration counters. Start points in the first delta cycle are first statements of methods that are run in processes with empty call stack and cycle iteration counters. Start points of subsequent evaluation phases are finish points of previous evaluation or update phase which has delayed notification. A finish point of evaluation phase analysis is a program point with statement \(s\) which is wait or return statement in process. There is one program state for each start point.

The evaluation phase analysis contains interval analysis, points-to analysis and lockset analysis. A lockset analysis counts delayed notifications, immediate notifications and request update calls in execution paths that are analyzed in this phase. Analyzed execution paths are statement chains from a start point to a finish point.

After all execution paths are analyzed evaluation phase analysis checks program points blocked on preceding evaluation phases. If there is an immediate notification for such a blocked point, this point is moved to a start point set and analysis of this evaluation phase continues (Fig. 1). Analysis of these start points begins with a program state that is union of
states from all finish points where immediate notification has happened. This rule allows us to consider possible variable changes that are made before and after immediate notification. Evaluation phase analysis continues until start point set is empty.

4.4 Update phase analysis

Update phase analysis starts after the evaluation phase analysis if there are request_update calls in the evaluation phase of current delta cycle. For each finish point of execution, if paths have request_update call then corresponding update method is analyzed. The input state of an update() method analysis is a state in this finish point. It is not required to consider states of other finish points because update method may operate with current object data only. An example of update phase analysis in shown in the Fig. 2 (ru denotes request_update call and upd – a first statement of update).

Analysis of update methods can be performed in any order because there is no interaction between methods on update phase. Results of the update phase analysis are linked to corresponding finish points. For such finish point there are:

- modified program state,
- an additional set of delayed notifications.

4.5 Interprocess Data Analysis

For each delta cycle static analysis algorithms work with individual program state on execution paths. At the end of delta cycle, all program states changes are merged. A state merging determines and uses actual values of shared variables in processes. An actual value is the last value assigned to this variable.

To determine actual values, a set of changed objects (Def) is used. This set is filled during evaluation and update phase analysis – if a value of variable is changed, it is added to Def, if a variable leaves its scope, it is removed from Def.

The state merging is done for states of all finish points. The result state for a finish point is computed using the following formula:

$$\hat{S}_f = S_f \otimes \bigcup_{j \neq k} \bigcup_{(o_i, v) \in S_m, (s, p_j, C, I) \in Def_j} \langle o_i, v \rangle$$

where $S_f$ and $\hat{S}_f$ are program states before and after state merging in $F_f = (s, p_k, C, I)$, $\otimes$ is an operation that replaces tuples with equal left elements. An example of interprocess data analysis in shown in the Fig. 3.

4.6 Variant analysis

A delta notification may occur in an arbitrary number of execution paths of a process. For an event $e$ all finish points at the end of delta cycle can be divided into following groups:
Figure 3: Interprocess data analysis

- Finish points, all paths to which contain `notify(e, 0)` statement;
- Finish points, all paths to which don’t contain `notify(e, 0)` statement;
- Finish points, some paths to which contain `notify(e, 0)` statement.

Both the first and the second groups unambiguously define whether process blocked on corresponding event has to be resumed or not. Finish points from the last group define notification ambiguously. If we simultaneously analyze presence and absence of an event notification at each finish point, deadlock detection will get a lot of false positives. To cope with this problem we use variant analysis.

Algorithm for variant creation has the following steps:

- Determine a set of events `{e_i}`, `i = 1...n` that have been notified in the delta cycle.
- For each finish point of a process calculate a notification vector `V = (v_1, v_2, ..., v_n)`, where `v_i = 1` if there is notification for `e_i`, and `v_i = 0` otherwise. If a finish point contains ambiguous notification, then it has several notification vectors.
- Create combinations of finish points. A combination `C` contains one finish point for each process.
- Calculate notification vector `V(C)` for each combination `C` as `V(C) = V_1 \oplus V_2 \oplus ... \oplus V_K`, where `V_i` is a notification vector for `F_i` and `\oplus` is a bit OR operation.
- Combinations with the same notification vectors are joined into one variant.

Obviously, a variant has only unambiguous notification for all events. All variants are analyzed separately.

4.7 Program Dependencies

Developed algorithms are sound but not precise. Main causes of algorithm imprecision are the following:

- Program state merging in \( \phi \)-functions,
- Program state merging in the end of delta cycles.

One of methods to improve precision is to use program dependencies. Program dependencies are relations between values of several program objects. A dependency is defined for some part of a program, which is a program point or a statement. Our approach utilizes linear dependencies of variable pairs inside a program processes. It can be extended with interprocess and non-linear dependencies.

Another method to increase precision is to use dependencies between program variables and presence of notifications for events. Such dependencies let to avoid analysis of inconsistent notification for certain `wait` statement.

Our approach uses variant analysis, which leads to decrease of analysis scalability, especially for programs with a large number of ambiguous notifications. One of solutions is to simultaneously analyze interprocess and non-linear dependencies.

5 Deadlock Detection

The basis of synchronization primitives in SystemC includes blocking statement `wait` and notification statement `notify`. All other synchronization primitives are based on `wait` and `notify`. So the only
possible deadlock situation is infinite wait on an event that will never be notified.

There are two main types of SystemC programs: programs that finish execution in a finite number of steps and infinitely executed programs. Deadlock in the program of the first type leads to presence of blocked processes at the end of simulation. Programs of the second type may continue their execution infinitely even in the presence of deadlock in some processes. These two possibilities lead to different deadlock detection rules. These rules are based on an analysis of blocked points and are applied at the start of each delta-cycle.

**Rule 1**

A deadlock of the first type is possible if for each process there is at least one blocked point. Corresponding formula is presented below.

∀p_j ∈ P  ∃B_l : (B_l = (s, p_j, C, I)),

where P is a set of processes, B_l is a blocked point.

This rule can lead to lots of false positives. For improving the precision of the rule, program dependencies can be used as discussed above.

**Rule 2**

To deal with deadlocks of the second type, we introduce delta-cycle counters that are linked to blocked statements. If no notification exists for a wait statement in a given amount of delta-cycles, there is a deadlock in the program.

∃B_l :  count(B_l) > T,

where B_l is a blocked point, count(B_l) is a number of delta-cycles the program is blocked in B_l and T is a threshold value.

Threshold values can be set globally (for all wait statements), or locally (for any subset of wait statements independently).

6 Efficiency Evaluation

The approach presented in this paper has been implemented in Deadlock Analyzer tool. Source codes of the tool include more than 50 KLOC of Java code in total.

In this section we describe experiments being conducted in order to estimate an efficiency of Deadlock Analyzer. Since there is no available SystemC deadlock testbenches, we created our own testbench including artificial tests and one real-life model.

Test cases include well-known multi-process synchronization problems: dining philosophers, producer-consumer, crossroads. We created program modifications of the code with deadlock as well as without deadlock. There are 65 test programs in total (each program is about 100 LOC and contains 2-5 processes). All deadlocks were successfully detected, no false errors were found.

A real-life example is a model of a video accelerator provided by Intel Labs. Figure 4 shows a top level of the accelerator and its modelled surrounding. The accelerator itself has deeply embedded microcontroller and high performance data crunching engine consisting of custom datapath, local SRAM memory and Direct Memory Access unit (DMA). An accelerator environment is represented in simplified form as a CPU and it’s memory. The system memory can be accessed by DMA independent of CPU and thus is represented as a separate unit.

![Figure 4](attachment:image.png)

Figure 4: Example system under test

Source codes of the video accelerator model contain more than 7800 LOC. This model contains one real deadlock, which was successfully found by our tool.

To estimate completeness of the deadlock detection we utilized mutation testing. We used two types of mutation for artificial deadlock injection:

- moving notify into the branch of if,
- insertion of unsatisfied wait.

Using simple bash script we generated 12 mutants. In 11 of 12 cases our tool has successfully found deadlock. In these experiments 3 false deadlocks were found.

We use Intel Xeon E5410 2.33 GHz processor with 12 GB of RAM for the described experiments. Analysis time for each mutant varies from 10 to 30 minutes.
7 Conclusions

In this paper we have described an approach to deadlock detection in SystemC programs using static analysis. The presented approach has been implemented in Deadlock Analyzer tool, which we used to estimate precision and completeness of our algorithms. Developed algorithms yield correct results on the set of artificial tests: all the expected deadlocks were successfully detected, no false errors were found. Moreover, our analysis of a real-life video accelerator model developed at Intel Labs corroborates the ability of deadlock detection in industrial-level SystemC designs. In order to estimate completeness of error detection we create mutants of this model by injecting deadlocks. The result of these experiments is 91 percent completeness.

The approach discussed in the paper has some limitations. It doesn’t support spawned and dynamic processes, time notifications and time waits. Currently, we are working on implementation of these features in our tool.

References


