Static Data Race Detection in SystemC Parallel Programs

Alexey Zakharov and Mikhail Moiseev

Saint-Petersburg State Polytechnical University
zakharov@kspt.ftk.spbstu.ru mikhail.moiseev@gmail.com
http://www.spbstu.ru

Abstract. Hardware/software systems simulated using SystemC language are usually parallel and, therefore, may contain synchronization errors. One widespread type of synchronization errors is data races.
In this paper we propose an approach to data race detection in SystemC programs which is based on source code static analysis. We‘ve developed static analysis algorithms that can extract information for data race detection in SystemC program without quantitative time. These algorithms can detect all errors that exist in the program. Efficiency of our approach is shown by the evaluation results of the developed tool on a set of test SystemC programs.

Keywords: SystemC static analysis data race error detection

1 Introduction

Last years number of elements and as the result complexity of system-on-chip is increasing. This is the reason for use of high-level system description and modeling languages in process of system-on-chip design. SystemC [1] is one of these languages.

The main goal of SystemC is modelling of software/hardware systems for performance parameters estimation, variants of system design comparison and system behavior verification. A target software/hardware system is presented as a SystemC program. It is modelled by SystemC simulator.

Let us define *program block* as a sequence of SystemC statements which are executed in a process without context switching. *Parallel program blocks* – program blocks which are executed in different processes in the same delta cycle (at the one moment of modelling time). Execution order of parallel blocks is non-deterministic – SystemC scheduler chooses one of these blocks for execution in arbitrary order. There is *data race* error in SystemC program if two or more operations with a program object are performed in parallel blocks, and at least one of these operations modifies this object.

Presence of data races in a SystemC program leads to non-deterministic program behavior and incorrect modelling results. In the cases when data races remain in a target system, the system may become inoperative.

In this paper we propose an approach to data race error detection in SystemC programs. Our approach is based on both well-known and new static analysis algorithms. These algorithms extract information about operations with program objects in parallel program blocks taking into account interprocess synchronization and SystemC scheduler semantics. The approach is *sound* – it gives all data race errors, but it is not *precise* – it can give false positives.

2 Related Work

Data race errors may be found using dynamic or static methods. Dynamic methods require instrumentation of either source or object code. Such an instrumentation may be used for logging information about variable usage by processes as well as for assertion checking. For example, object code instrumentation is used in [2] for lockset-checking which allows data race detection. Dynamic methods don't ensure analysis of all possible paths for all input data, so they don't ensure sound error detection.

In [3] combination of static analysis and model checking is used for verification of SystemC programs. Static analysis extracts information about system components and interconnections, whereas model checking is used to infer processes commutativity conditions. This approach is aimed to optimize execution time of SystemC programs. The approach is implemented in Scoot tool. This tool doesn't detect data races explicitly but infers process commutativity invariants.

Model checking is adopted in many papers related to SystemC error detection. In some of these papers a program model is converted into an input language of existing model checking tool [4],[5], whereas other papers introduce novel formalisms and model checking algorithms [6],[7].

Static analysis is widely used for error detection in parallel programs written in such languages as C, C++, Java. An approach to data races detection in multithreaded C programs is described in [8]. This approach is based on type and effect system that determines synchronization objects which are used for protection operations with shared objects.

Synchronization error detection in multithreaded C programs is discussed in [9]. The suggested approach uses a transaction graph that is being refined by partial order reduction and lockset analysis. A modification of this approach for programs with asynchronous tasks and pool of threads is proposed in [10].

A method for data race detection in Java programs is suggested in [11]. This method is based on iterative use of static analysis algorithms that extract information about interval variables, pointers, shared objects use, program statements which can be executed in parallel.

As the result of related work review we conclude that static analysis methods are not widely used for data race detection in SystemC programs. At the same time these methods are successfully applied for error detection in parallel programs in other programming languages. In this paper we extend static analysis methods to SystemC programs using our experience in C/C++ programs analysis.

3 Main Idea

Most systems simulated using SystemC are parallel systems. A parallel SystemC program is a set of interacting processes managed by a scheduler. The SystemC scheduler binds execution of a statement to a moment of time and orders execution of statements from different processes for the same time using delta cycles.

Program execution includes several phases. In elaboration phase modules, processes, channels and other entities of a program are created and connected to each other. Elaboration phase ends with `sc_start` function call. Execution of `sc_start` results in running of created processes and initiating of the first delta cycle. Each delta cycle includes two phases: evaluation phase and update phase.

Processes are executed by turns. Execution of a process in evaluation phase ends when either `wait` function is called or process finished. If some process is waiting for an event by calling `wait`, and some another process invokes immediate notification for the same event, the first process resumes execution in the current delta cycle. Whenever delta notification occurs, corresponding waiting process becomes runnable in the next delta cycle. Evaluation phase continues while there are runnable processes. In update phase for all the processes that have executed `request_update` method in evaluation phase of the delta cycle, update methods are called. Whenever all the delta cycles are analyzed for current modelling time, the simulation kernel increases modelling time until runnable processes become available. If there are no runnable processes, program execution stops.

In order to simplify analysis algorithms, program model in the form of control flow graph is used. Methods and functions for parallel execution are represented in program model using special types of statements listed in table 1.

Table 1. SystemC-specific Statements

Statement	Description
`run(this, t, f)`	Run function `f` in process `t`. Replaces `create_thread_process`
`run_update`(this, update)	Run `update(this)` method on the update phase of current delta cycle. Replaces `request_update`
`start()`	Start simulation. Replaces `sc_start`
`sense(t, e)`	Add to sensitivity list of process `t` event `e`. Corresponds to `«`
`wait(e, time)`	Wait for event list `e` at most `time` time. Represents `wait(t), wait(n, time_unit), wait(e)`
`notify(e, time)`	Notify event `e` after `time` time. Represents `e.notify(), e.notify(t), e.notify(n, time_unit)`

In this paper we consider a set of SystemC programs that does not use quantitative time, i.e. nonzero time parameters in synchronization functions. Simulation of such programs includes elaboration phase followed by several delta cycles.

Data race detection requires analysis of read and write access to shared variables in parallel blocks. Since SystemC is an extension of C++, program objects may be accessed indirectly through pointers. So analysis of accesses to program objects requires pointer analysis. In order to ensure high precision of error detection, it is necessary to perform interval analysis – an analysis of numeric variable values. Numeric variables may be used as branch conditions, as iterators of loops, as indices in array access expressions etc. Pointer and interval analysis algorithms are based on algorithms described in [12].

Data race analysis requires detection of parallel blocks of program, so it is necessary to analyze synchronization statements. We use lockset analysis to cope with synchronization statements. Analysis of a parallel program is done by a special algorithm, that considers SystemC scheduler semantics.

Thus data race error detection requires the following static analysis algorithms:

- an interval analysis algorithm
- a pointer analysis algorithm
- an algorithm for detection of accesses to program objects
- a lockset algorithm
- a parallel execution analysis algorithm.

4 Static Analysis Algorithms

The lockset algorithm gives auxiliary information for parallel block detection. This algorithm calculates a number of `notify` statements on all possible execution paths in program processes.

Let B_i^j be a program block that is executed in j-th process in the current delta cycle. Let designate $l(s_k, e_m)$ as numbers of `notify(`e_m`)` statements on all possible execution paths from the first statement of B_i^j to $s_k : s_k \in B_i^j$ (e.g. $l(s_k, e_m)$ is a set of numbers of e_m notification). The lockset algorithm consists of rules for program statements (see Fig. 1):

$$[notify(e_m)]_k : \quad \begin{cases} l(s_k, e_r) = l(s_n, e_r) + 1, & r = m \\ l(s_k, e_r) = l(s_n, e_r), & r \neq m \end{cases}, \quad s_n \in Pred(s_k),$$

$$[phi()]_k : \quad l(s_k, e_r) = \bigcup_{\forall s_n \in Pred(s_k)} l(s_n, e_r), \quad \forall r,$$

$$[if(...)]_k : \quad \begin{cases} l(s_k, e_r)_{true} = l(s_n, e_r) \\ l(s_k, e_r)_{false} = l(s_n, e_r) \end{cases}, s_n \in Pred(s_k), \quad \forall r,$$

$phi()$ – a program model statement, where program branches are joined, $Pred(s_k)$ – a set of statements which are direct predecessors of s_k.

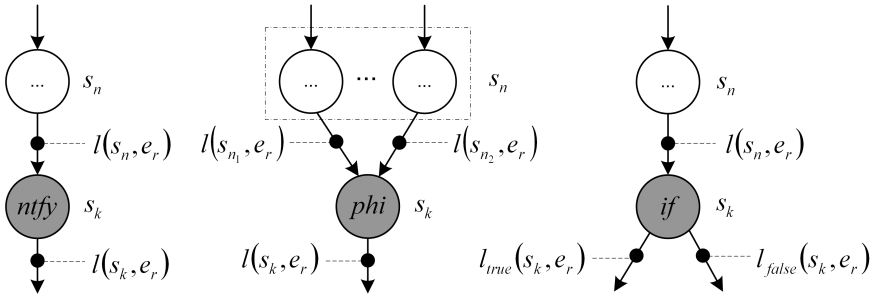

Fig. 1. Lockset algorithm rules

The parallel execution analysis algorithm determines bounds of a delta cycle in each program process – parallel program blocks which are executed in this delta cycle. Analysis of evaluation phase ends with a `wait` statement or with the last statement of a process. Afterwards the algorithm looks for immediate notifications of events that are waited in processes blocked on `wait` statements. If there are such notifications, then corresponding `wait` statements and the following statement chains are analyzed in evaluation phase of the current delta cycle. In the result of these statements analysis a set of immediate notifications may be extended. Analysis of immediate notification is run until there are no new unblocked `wait` statements are determined.

After evaluation phase analysis is finished, update phase analysis begins. In update phase, analysis of `update()` methods executed by `request_update()` calls in evaluation phase of the current delta cycle are processed. After update phase analysis the algorithm proceeds to the next delta cycle. It determines `wait` statements in all processes which are unblocked by delayed notifications of this delta cycle. These `wait` statements and following statement chains are analyzed in the next delta cycle.

Interval analysis, pointer analysis and lockset algorithms are performed for each program process separately. Interaction between process analysis algorithms are considered by merging results of these algorithms in the end of each delta cycle. Merging rules for values of an object o_k are the following:

$$V(o_k) = \bigcup_{\forall B_i^j} V_{B_i^j}(o_k), \qquad o_k \notin Def(B_i^j), \forall B_i^j$$

$$V(o_k) = \bigcup_{\forall B_i^j : o_k \in Def(B_i^j)} V_{B_i^j}(o_k), \qquad \exists B_i^j : o_k \in Def(B_i^j),$$

$V(o_k)$ – a set of o_k values in the end of the current delta cycle, $V_{B_i^j}(o_k)$ – a set of o_k values in the block B_i^j, $Def(B_i^j)$ – a set of program objects that are modified in the block B_i^j in the current delta cycle.

One problem of parallel block detection is `notify` statements in process branches – *variant notification*. Such `notify` statements appear on some process

execution paths only. Variant notifications lead to analysis of some statements in several delta cycles.

The error detection algorithm uses the following rule:

$$\exists B_{i_1}^{j_1}, B_{i_2}^{j_2} : (B_{i_1}^{j_1} \parallel B_{i_2}^{j_2}) \wedge (Def(B_{i_1}^{j_1}) \cap (Def(B_{i_2}^{j_2}) \cup Use(B_{i_2}^{j_2}))) \neq \emptyset),$$

$B_{i_1}^{j_1}, B_{i_2}^{j_2}$ – program parallel blocks, $Use(B_i^j)$ – set of program objects which are used in the block B_i^j in the current delta cycle. This rule detects cases where a program object is modified in parallel blocks or it is modified in one of parallel blocks and used in other blocks.

An example of a program which contains a data race error is shown in the Fig. 2. Blocks $B_{i_2}^2$ and $B_{i_3}^3$ in this example can be executed in an arbitrary order –

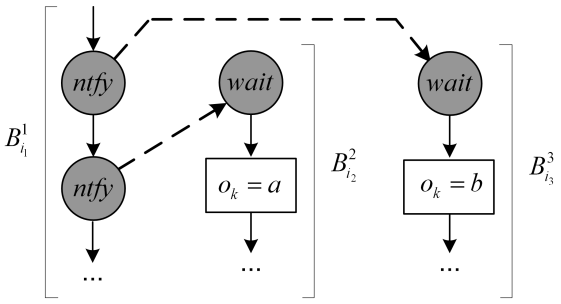

Fig. 2. Data race error sample

the result value of o_k is a or b.

5 Efficiency Evaluation

Efficiency indicators of error detection are *precision* – a fraction of a number of true errors found to a total number of errors found, and *soundness* – a fraction of number of true errors found to a number of errors in the program.

In order to estimate efficiency of the proposed approach we‘ve, implemented it in SC Race Detector tool. We‘ve developed a set of SystemC test programs, some of which include injected data race errors. Developed test programs use various SystemC synchronization models: waiting for next delta cycle, immediate notification, delta-notification. Some of test programs have masked errors – errors that may cause to fault only when a specific condition meets or for specific iterations of a loop. The test suit includes 18 tests in total. Size of each test program is 100-150 lines of code. A description of test groups is shown in table 2.

We use Intel Xeon 2.33 GHz processor with 12 GB of RAM for these experiments. Analysis time of test programs varies from 5 to 10 seconds. For each test program we estimate total number of errors, a number of errors found and

a number of true errors found. The results of the experiments show that all injected errors were detected and no false positives were found.

Table 2. Tests Description

Test Group Description	Number of Programs	Number of Processes	Number of Errors
Use `wait(0, SC_NS)`	4	2	2
Use immediate notifications and loops	6	2-3	4
Use delta-notification	2	2	1
Use request_update	2	3	1
Dining philosophers implementations	4	2-5	3

6 Conclusion

In this paper we present an approach that ensures automatic data race error detection in SystemC programs based on static analysis. Errors of this type can cause to non-deterministic behavior of a program. The proposed approach ensures sound data race detection.

The developed analysis algorithms consider SystemC scheduler semantics. These algorithms support immediate notification, delta notification, waiting for the next delta cycle. The proposed approach and corresponding algorithms are implemented in SC Race Detector tool based on AEGIS static analysis platform [13].

In order to estimate an efficiency of the proposed approach, we have developed SystemC test programs containing various interprocess communication cases. The experiments show good results of SC Race Detector on test programs, so we believe that it is possible to apply our tool to industrial-level SystemC models.

Currently we are extending described algorithms to provide full support of quantitative time. Implementation of these algorithms will allow to analyze a wide set of SystemC programs.

References

1. 1666-2005 IEEE Standard SystemC Language Reference Manual
2. S. Savage, M. Burrows, G. Nelson.: Eraser: a Dynamic Data Race Detector for Multithreaded Programs. J. ACM Transaction of Computer System. 15, 391–411 (1997)
3. N. Blanc, D. Kroening.: Race Analysis for SystemC using Model Checking. In: ICCAD, pp. 356–363 (2008)
4. C.Traulsen, J.Cornet, M.Moy, F.Maraninchi.: A SystemC/TLM Semantics in Promela and Its Possible Applications. In: 14th Workshop on Model Checking Software SPIN, pp. 204-222 (2007)

5. H. Garavel, C. Helmstetter, O. Ponsini, W. Serwe.: Verification of an Industrial SystemC/TLM Model Using LOTOS and CADP. In: 7th IEEE/ACM International Conference on Formal Methods and Models for Co-Design, pp. 46-55 (2009)
6. H.Hojjat, M.R.Mousavi, M.Sirjani.: Process Algebraic Verification of SystemC Codes. In: 8th International Conference on Application of Concurrency to System Design, pp. 62-67 (2008)
7. Yu Zhang, F.Vedrine, B.Monsuez.: SystemC Waiting-State Automata. In: 1st International Workshop on Verification and Evaluation of Computer and Communication Systems (2007)
8. P.Pratikakis, J.S.Foster, M.Hicks.: LOCKSMITH: Practical Static Race Detection for C. J. ACM Transactions on Programming Languages and Systems. (2011)
9. V.Kahlon, S.Sankaranarayanan, A.Gupta.: Semantic Reduction of Thread Interleavings in Concurrent Programs. In: 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (2009)
10. V.Kahlon, N.Sinha, E.Kruus, Y.Zhang.: Static Data Race Detection for Concurrent Programs with Asynchronous Calls. In: 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering (2009)
11. M.Naik, A.Aiken.: Conditional Must Not Aliasing for Static Race Detection. In: 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (2007)
12. V.M.Itsykson, M.Ju.Moiseev, A.V.Zakharov et al.: Automatic Defects Detection in Industrial C/C++ Software. In: 5th Central and Eastern European Software Engineering Conference in Russia (2009)
13. Aegis Error Detection Tool, `http://www.digiteklabs.ru/aegis/`