Simulation Verification of multiple levels of constraints for system level designs in systemC

Piyush Ranjan Satapathy, Xi Chen and Harry C. Hsieh
Department of Computer Science
University of California, Riverside, CA 92521
piyush, xichen, harry@cs.ucr.edu

Abstract— **A problem of increasing importance in the design of highly complex, heterogeneous, concurrent and large multiprogramming systems is the, so-called, deadlock or deadly-embrace problem. So deadlock detection and resolution has been an important issue for decades. In this paper we have surveyed the different possibilities of deadlocks in system design in SystemC language environment and then we verify the work that has been done on the treatment of deadlocks from both the theoretical and practical points of view in concurrent systems modeled in Metropolis design environment. We have applied the proposed data structure called the dynamic synchronization dependency graph (which captures the run time dependencies in MMM environment) in our deadlock detection algorithms and found that it suits well to the designs in the systemC environment too. We have also figured out some other constraints of system level design like live lock and starvation in systemC environment. Some changes have been applied to the loop detection algorithm to handle the detection of such scenarios in the specified design environment. We demonstrate our surveys, analysis and applications through different real world design examples.**

I. Introduction

System level design has been an essential part of EDA industry for last few years. The earlier in the design process a designer can locate a problem, the less time and resources it costs to fix the problem. Every time a design is synthesized from one level to a more detailed level, such as behavioral to RTL or RTL to gate, it takes longer to do a design simulation. Furthermore, problems that occur at higher levels of design abstraction are often hidden by the details of a lower-level abstraction, making design debugging more difficult and time consuming. Lost time and added expense are pushing design, analysis, and verification above the RTL level. So verification with multiple levels of constraints at system level design is a must.

Different Constraints specification at higher level of abstraction in system design has already been done by the authors in [3]. The most significant constraint in system level design is deadlock detection and prevention which has been simulated and verified in a generalized model in [5]. The authors in [2] have discussed the deadlock analysis of system level design in a specific environment called Metropolis simulation. They have also proposed a data structure called the dynamic synchronization dependency graph (DSDG) that reflects online deadlock analysis. The authors in [1] have presented a live lock analysis for a specific purpose oriented system level design. Also the starvation analysis [4] which is a similar concept of deadlock analysis has been discussed heavily in the literature. Here we do the verification of all the above mentioned constraints named, deadlock analysis, live lock analysis and starvation analysis at system level design in SystemC simulation environment.

SystemC [**?**] enables system level modeling that is, modeling of systems above the RTL level of abstraction, including systems which might be implemented in software or hardware or some combination of the two. One of the challenges in providing a system level design language is that there is a wide range of design models of computation, design abstraction levels, and design methodologies used in system design. To address this challenge in SystemC 2.0, a small but very general purpose modeling foundation has been added to the language. On top of this language foundation we can then add the more specific models of computation, design libraries, modeling guidelines, and design methodologies which are required for system design. Using some of the existing features of SystemC (for example; wait() method, Event type, SystemC scheduler, and systemC execution model) we can build a simulation model for the analysis of our desired constraints.

II. Related Work

III. Synchronizations in systemC

SystemC has a set of features like channels, interfaces, and events for generalized modeling of communication and synchronization. A channel is an object that serves as a container for communication and synchronization. Channels implement one or more interfaces. An interface specifies a set of access methods to be implemented within a channel, but the interface itself doesn't provide the implementation. An event is a flexible, low-level synchronization primitive that is used to construct other forms of synchronization. In this section we review different synchronization constructs in systemC level description and discuss how deadlock situations are caused by the synchronization mechanism in a concurrent system model.

A. Static and Dynamic Sensitivity in SystemC

In order to facilitate the modeling at higher level of abstraction and as well as the creation of refined communication channels, SystemC language description forms a layered approach for system level design. SystemC simulation kernel forms the base layer. Dynamic sensitivity and notion of events are introduced as components of the next layer. Channels, interfaces and ports form the third layer as Interface Method Call scheme. Then signals are implemented on top of the

TABLE I
LAYERED APPROACH OF SYSTEMC SIMULATION KERNEL

Remote Procedure Calls (RPC)
Signals
Channels, Interfaces and Ports
Events & Dynamic Sensitivity
SystemC Simulation Kernel

channels, interfaces and ports layer. And then the Remote Procedure Calls scheme is implemented on top of the other four layers. The layered approach has been shown in Table **??** In systemC a process plays an important role. It describes the functionality of the system, and allows expressing concurrency in the system. Processes are contained in modules and they access external channel interfaces through the ports of a module. Events are treated as primitive behavior triggers. A process can suspend on or be sensitive to one or more events. Events allow for resuming and activating processes. The sensitivity of a process defines when this process will be resumed or activated. A process can be sensitive to a set of events. Whenever one of the corresponding events is triggered, the process is resumed or activated. If the sensitivity of the process is declared statically; i.e., if it is declared during elaboration and can not be changed once simulation has started, then it is called Static Sensitivity. The sensitivity of a process can also be altered during simulation which is called Dynamic Sensitivity. In some cases of the system level design we want a process to be sensitive to a specific event or a specific collection of events which may change during simulation. This dynamic sensitivity is possible by using the wait( ) method and notify( ) method. This method has been extended to allow specifying one or more events or a collection of events to wait for.

B. Synchronization by wait statement

In systemC, the wait( ) method is called anywhere in the thread of execution of a thread process. When it is called, the specified events temporarily overrule the sensitivity list, and the calling thread process suspends. When one or all of the specified events is notified, the waiting thread process is resumed. The calling process is again sensitive to the sensitivity list. When the wait( ) method is called without arguments, the calling thread process will suspend. When one of the events in the sensitivity list is notified, the waiting thread process is resumed. The static sensitivity of the calling thread process doesn't change. In addition to events, it is also possible to wait for time which is used as a timeout when waiting for one or more events. Following is a list of different forms of wait( ) method which are supported by systemC.

1) wait( ): Upon this method, the calling process waits until the events in the sensitivity list are notified.
2) wait(e_1): This wait method resumes execution of the current process until the event e_1 is notified.
3) wait($e_1 \mid e_2 \mid e_3$): This wait method holds the current thread process until either of events e_1, e_2, or e_3 is done.

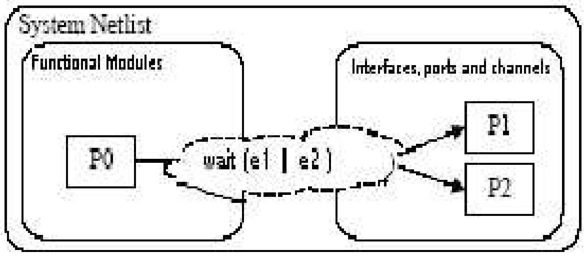

Fig. 1. An example of wait() statements

4) wait (e_1 & e_2 & e_3): This wait method holds the execution of current thread process until all of the events e_1, e_2, e_3 are done.

The semantics of the wait( ) method with one or more event arguments is that the method returns (i.e. the thread of execution is resumed) either when at least one of the events is notified or when all the events are notified. A mixture of $\mid$ operator and & operator is not supported by systemC.

In systemC, the system function and the architecture are modeled as separate networks of process communicating through channels. In a functional network, functional processes run concurrently and communicate with each other through ports, channels and interfaces. In an architectural network, computing and storage resources are modeled with the media. Services that the architecture can provide are modeled with the processes that are called mapping processes. A function model is mapped to an architecture model as the events of functional processes and mapping processes are synchronized with wait( ) and notify( ) constraint. A designer is allowed to implement particular schedulers as systemC schedulers to manage architectural resources and services in architecture model.

A wait constraint is an alternative of a tryst used in the concurrent programming. It can specify that two events in two different processes must occur at the same time. If only one of the two events can be scheduled to occur, the process containing the event has to be blocked until the other event can occur also. A wait can also require that an event can't occur until any of the other events occur. The execution of a process has to be blocked at a certain event until all the wait constraints containing the events are satisfied. For example lets assume functional process P_0 and mapping process P_1 and P_2 have events e_0, e_1 and e_2 respectively and P_0 is synchronized by a wait constraint wait($e_1 \mid e_2$) which requires that e_0 cannot occur until e_1 or e_2 occurs. This scenario may denote that a functional process can not run until there are free computation resources and communication channels in the architecture. The execution of P_0 may be blocked by either P_1 or P_2 as illustrated in Fig. 1.

A wait constraint amongst functional processes can also lead to blocking until one or all the events of the called processes are notified. For example lets assume functional process P_0, P_1, P_2 and P_3 have events e_0, e_1, e_2 and e_3 respectively and P_0 is synchronized by a wait constraint wait(e_1 & e_2 & e_3) which requires that e_0 cannot occur until e_1 and e_2 and e_3

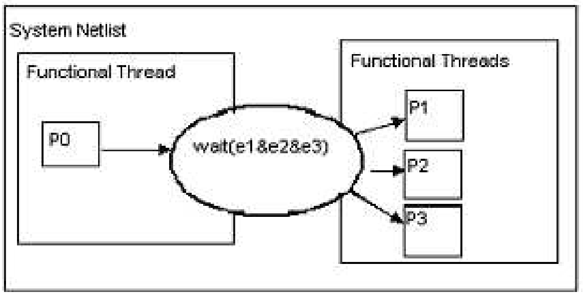

Fig. 2. An example of wait() constraint amongst the thread processes

occurs. This scenario may denote that a functional process can not run until there are other functional processes are done. The execution of P_0 may be blocked by any one processes of P_1, P_2 and P_3 as illustrated in the Fig. 2.

C. Synchronization by Events and Notifications

SystemC provides a fixed set of channels and corresponding events. The event type SC_EVENT supports user defined channel types. The different functionality that the event type SC_EVENT provides are as follows;

1) Constructor: SC_EVENT is used to create an event object by calling the constructor without any arguments.
2) Notify: An event can be notified by calling the notify( ) method of the event object.
3) Cancel: An event notification can be cancelled by calling the cancel() method of the event object.

notify( ) method can operate in 3 different ways. Those are as follows; (1)notify() with no arguments: immediate notification. Processes sensitive to the event will run during the current evaluation phase. (2) notify() with a zero time argument: delta notification. Processes sensitive to the event will run during the evaluation phase of the next delta cycle. (3) notify() with a non-zero time argument: timed notification. Processes sensitive to the event will run during the evaluation phase at some future simulation time.

The SystemC simulation kernel supports the concept of delta cycles. A delta cycle consists of an evaluation phase and an update phase. This is typically used for modeling primitive channels that cannot change instantaneously, such as sc_signal. By separating the two phases of evaluation and update, it is possible to guarantee deterministic behavior (because a primitive channel will not change value until the update phase occurs - it cannot change immediately during the evaluation phase).However SystemC can model software, and in that case it is useful to be able to cause a process to run without a delta cycle (i.e. without executing the update phase). This requires events to be notified immediately (immediate notification). Immediate notification may cause non-deterministic behavior. The ordering of the execution of events will be random and there would be possibilities like deadlock and live lock. For Example; If a process P_1 notifies immediately to an event e_2 of process P_2 and P_2 notifies immediately event e_3 of process P_3 and e_3 waits up to event e_1 of process P_1, then there will be a deadlock if the execution of the P_1 is over by the

TABLE II
DESCRIPTION OF THE BEHAVIOUR OF THE SIMULATION KERNEL

1	Initialization: execute all processes (except SC_CTHREADs) in an unspecifi ed order.
2	Evaluation: select a process that is ready to run and resume its execution. This may cause immediate event notifi cations to occur, which may result in additional processes being made ready to run in this same phase.
3	Repeat step 2 until there are no processes ready to run.
4	Update: execute all pending calls to update() resulting from calls to request_update() made in step 2.
5	If there were any delta event notifi cations made during steps 2 or 4, determine which processes are ready to run due to all those events and go back to step 2.
6	If there are no timed events, simulation is fi nished.
7	Advance the simulation time to the time of the earliest pending timed event notifi cation.
8	Determine which processes are ready to run due to all the timed events at what is now the current time, and go back to step 2.

time the notification of e_3 comes to the scheduler. In this case the e_3 will keep waiting but P_1 will never be finished as its already over. Also series of notify() or different combinations of wait() and notify() would cause such situations which we will explore detail later.

D. SystemC simulator kernel

The systemC simulation kernel plays an important role to do the scheduling. The scheduler's task is to determine the order of execution of processes within the design based on the event sensitivity of the processes and the event notifications which occur. The systemC scheduler has support for both software and hardware oriented modeling. Due to software modeling the systemC exhibits some non deterministic behavior which leads to system level blockings. The over all steps for the execution of the systemC scheduler are outlined below and the flowchart is represented in Fig. 3.

E. Deadlock in SystemC

Deadlock in systemC can be defined as situations where two or more processes are blocked in execution while each is waiting for some conditions to be changed by others. Given the events and sensitivity lists, the following situations may block the execution of a running process.

1) SystemC uses concepts like mutual exclusion (mutex) or semaphores to handle the software modeling where a shared variable is used. It guarantees that two processes can not simultaneously access the variable. So a process has to wait for the availability of resources from other functional or architectural processes. For example: In this example, which SC_THREAD will run first is undefined and there is no way in systemC to tell which will run first. If Thread2 runs first then there is no way to execute the thread1 which will cause a deadlock for the thread1.
2) A process cannot execute if it has wait statement and the events in the wait list are not finished. Suppose we have a process P_1 which has a wait statement as

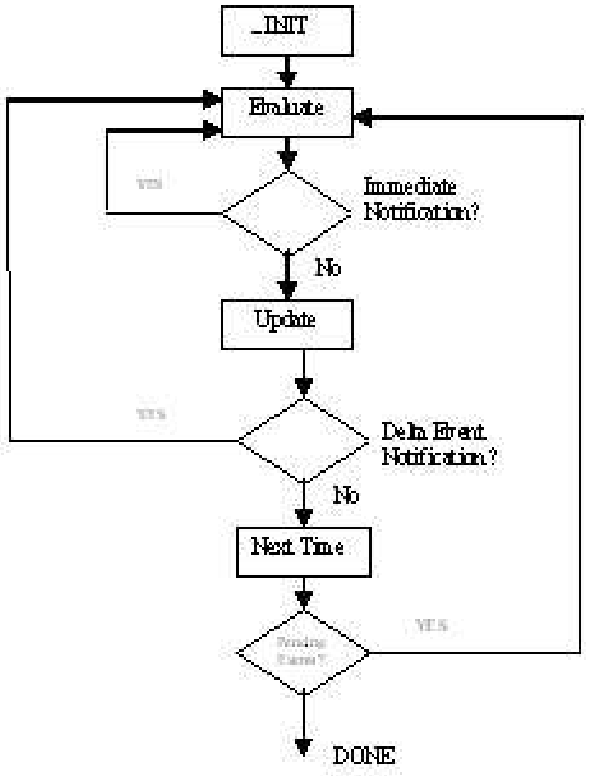

Fig. 3. Flow Chart Diagram of SystemC Scheduler

wait(e_2 | e_3 | e_4) where e_2, e_3 and e_4 are events in processes P_2, P_3 and P4 respectively. There may be cases when none of the three called events will finish execution. Such a case is when P_2 waits for P_3 and P_3 waits for P4 and P4 waits for an event in the P_1 which has not been executed yet.

3) Lets say that a process P_1 has an wait statement like wait (e_2 & e_3 &e_4) where e_2, e_3 and e_4 are some events in the process P_2,P_3 and P_4. Here the process P_1 cannot execute until all the three events are finished execution. So now if the same situations arise for e_2, e_3 and e_4 as described above in 1, we can get a blocking of the system. Also additionally if for every event e_2 or e_3 or e_4, there becomes a no determinism behavior as given in 1, then also P_1 will get blocked leading to blocking of all the wait statements in the modeling and so causing system level deadlock.
4) The event type and notify( ) also will cause nondeterministic behavior in systemC. For example let's say that process P_1 has events e_1 and e_2. The process P_1 calls event e_3 of process P_2 after its own execution of e_1 but before e_2. So now the execution of P_2 will be started just after the e_1. Now let's say that process P_2 has an event which notifies to event2 of process P_1. So here the execution order of P_1 from event e_1 to event e_2 will not be clear and any wait statements or any other event lists in that segment of the program may cause blocking of the system.

```
SC_MODULE(nondeterminism)

sc_in Trig;
int SharedVariable;
void proc_1()
  wait(sharedVariable);
  SharedVariable = 1;
  cout ≪ SharedVariable ≪ endl;

void proc_2()
  SharedVariable = 2;
  cout ≪ SharedVariable ≪ endl;

SC_CTOR(nondet)
  SC_THREAD(proc_1);
  sensitive ≪ Trig.pos();
  SC_THREAD(proc_2);
  sensitive ≪ Trig.pos();

;
```

Fig. 4. A Non Deterministic example of thread process

The interaction of these wait(), sc_event and notify() statements can be quite complicated and may land in the blocking of the system. A deadlock exists if there is nondeterministic behavior due to mutex sharing or if there exists dependency loops among the processes in a system. To identify and analyze these deadlock situations and report the processes before hand is our motive in this paper.

F. *Livelock in SystemC*

In systemC, live lock is defined as a situation where the system falls into dead loop and responds to no further interrupts. It is identified as infinite cyclic executions of any events. Different possibilities of live lock in systemC are described as below.

1) If a process P_1 has some events named e_1, e_2 and e_3 and all these events are specified as event objects and each one notifies the other by calling notify( ) or by calling notify(SC_ZERO_TIME) in a cyclic basis, then it will lead to a live lock of the process P_1 which will not let any other processes to execute.
2) When a method process is invoked, it executes until it returns. If there is an infinite loop inside certain number of processes due to the continuous calling of each other by notify(time, SC_NS, my_event) then a livelock situation will be occurred.
3) SystemC supports clock threading of processes by calling SC_CTHREAD. Clocked thread processes are only triggered on one edge of one clock, which matches the way hardware is typically implemented with synthesis tools. SC_CTHREAD processes typically have infinite loops that will continuously execute. So the execution is usually stopped from the looping by the use of watching construct. If the watching expression doesn't become

```
// datagen.h #include ”systemc.h”
SC_MODULE(data_gen)
sc_in_clk clk;
sc_inout¡int¿ data;
sc_in¡bool¿ reset;
void gen_data();
SC_CTOR(data_gen)
  SC_CTHREAD(gen_data, clk.pos());
  watching(event1.delayed());

;
//datagen.cc
#include”datagen.h”
voidgen_data()if(event1)⇒ Go to next simulation
time of the systemC scheduler.
while (true)
data = data + 1;
wait();
data = data + 2;
wait();
data = data + 4;
wait();
```

Fig. 5. Watching Event of SC_CTHREAD Process creating Livelock

true at all due to other constraints in systemC then the scheduler will keep executing the same process which will lead to a livelock.

G. Starvation in systemC

In systemC, a subtler definition of starvation is that a process is blocked and waiting for some events to occur and at least one process that is able to notify the event is still running, but will never notify the event to lease the waiting process. Some of the possibilities are described below.

1) If a process P_0 is blocked by a sensitivity list as Wait $(e_1\&e_2\&e_3)$, then if any one of these three events doesn’t get notified then P_0 will wait infinitely and it will lead to starvation of the PO. And we have already discussed some of the cases where an event keeps blocking or doesn’t get notified.
2) Circular Notify () event at immediate phase or at next delta cycle phase cause some of the processes to live lock. So the processes which wait for those live locked processes will be in starvation.
3) SC_CTHREAD will also cause some starvation situations when watching( ) condition doesn’t get satisfied or when wait_until() condition doesn’t get satisfied.

References

[1] Bettina Buth, Jan Peleska, and Hui Shi. Combining methods for the livelock analysis of a fault-tolerant system. *http://www.informatik.uni-bremen.de/agbs/jp/papers/amast98.html*, 1998.

[2] X. Chen, A. Davare, H. Hsieh, A. Sangiovanni-Vincentelli, and Y. Watanabe. Simulation-based deadlock analysis for system level designs. In *DAC ’05*, 2005.

[3] J. Burch L. Lavagno R. Passerone F. Balarin, Y. Watanabe and A. Sangiovanni-Vincentelli. Constraints speciﬁcation at higher levels of abstraction. In *Workshop on High Level Design Validation and Test*, November 2001.

[4] G. M. Karam and R. J. A. Buhr. Starvation and critical race analyzers for ada. *IEEE Transactions on Software Engineering, v.16 n.8, p.829-843*, August 1990.

[5] A. Basavatia M. Krishnamurthi and S. Thallikar. Deadlock detection and resolution in simulation models. *26th Conference on Winter Simulation*, 1994.