Simulation based deadlock analysis for system level designs

Conference Paper · January 2005

DOI: 10.1109/DAC.2005.193812 · Source: DBLP

Xi Chen
University Carlos III de Madrid

Alberto L. Sangiovanni-Vincentelli
University of California, Berkeley

Yosinori Watanabe
Cadence Design Systems, Inc.

Simulation Based Deadlock Analysis for System Level Designs

Xi Chen[1], Abhijit Davare[2], Harry Hsieh[1], Alberto Sangiovanni-Vincentelli[2], Yosinori Watanabe[3]
[1]University of California, Riverside, CA 92521, {xichen, harry}@cs.ucr.edu
[2]University of California, Berkeley, CA 94720, {davare, alberto}@eecs.berkeley.edu
[3]Cadence Berkeley Laboratories, Berkeley, CA 94704, watanabe@cadence.com

ABSTRACT

In the design of highly complex, heterogeneous, and concurrent systems, deadlock detection and resolution remains an important issue. In this paper, we systematically analyze the synchronization dependencies in concurrent systems modeled in the Metropolis design environment, where system functions, high level architectures and function-architecture mappings can be modeled and simulated. We propose a data structure called the dynamic synchronization dependency graph, which captures the runtime (blocking) dependencies. A loop-detection algorithm is then used to detect deadlocks and help designers quickly isolate and identify modeling errors that cause the deadlock problems. We demonstrate our approach through a real world design example, which is a complex functional model for video processing and a high level model of function-architecture mapping.

Categories and Subject Descriptors: I.6 [Simulation and Modeling]: Model Validation and Analysis; D.2.5 [Software Engineering]: Testing and Debugging - *Monitors*

General Terms: Verification

Keywords: simulation, deadlock, synchronization, cyclic dependency, system level, Metropolis

1. INTRODUCTION

Today's electronic systems have become highly complex, highly heterogeneous, and highly concurrent. The platform-based system level design methodology is increasingly being adopted as the primary method to deal with the complexity of these modern systems. In system level design, the desired function and the architecture at hand are captured separately at high levels of abstraction. The design procedures refine the abstract function, refine the abstract architecture, and map the function onto the architecture to realize the final implementation [3, 11]. High level design procedures allow designers to tailor their architectures to the desired functions or to modify their functions to suit the available architectures. The orthogonalization of various aspects of design concerns (e.g. function v.s. architecture, computation v.s. communication, and datapath v.s. control) makes both design exploration and verification easier.

Even with careful methodological guidance, it is still possible to introduce unintended and undesirable behaviors into function specifications, high level architecture models or function-architecture mappings. Foremost among these are deadlock, livelock and starvation. Being semantic in nature, their complete and precise characterization requires formal analysis or verification, which can only be done at a high level of abstraction due to the state space explosion problem. In this work, We look for a practical solution to deal with these design problems in realistic and complex system designs. A simulation based analysis methodology is proposed for the detection and elimination of these "semantic errors". Designers are responsible for coming up with simulation vectors and scenarios that are important and may lead to undesirable behaviors such as a deadlock. Our approach automatically analyzes the simulation status and reports deadlocks once they occur.

While our deadlock monitoring approach can apply to any system level design environment, we focus our effort on the synchronization dependency and deadlock analysis for simulation in the Metropolis design environment [3]. Metropolis is a system level design framework for modern embedded systems. In the modeling language of Metropolis, Metropolis Meta-Model (MMM), a design is specified as asynchronous processes with communication specified with media and with its overall behavior limited by the synchronization constructs: function-architecture mappings, *await* statements, interface function calls, constraints, and schedulers. The function and abstract architecture of a system are specified separately and correlated by the synchronization of the functional events with architectural events (*mapping*). An *await* statement can be used to make a process wait until some conditions hold, establish critical sections that guarantee mutual exclusion among different processes, and prevent interface function calls by other processes. To limit the behavior of processes, designers can put high-level LTL (Linear Temporal Logic) [14] or LOC (Logic of Constraints) [2] constraints on the system specification without giving any specific scheduling algorithm, and leave the implementation to the lower levels of abstraction. Designers can also write their own schedulers in architecture models at a high abstraction level, which are called *quantity managers* in Metropolis. The high flexibility of the design platform allows designers to use different modeling constructs freely in a system design. Without a platform-supported systematic analysis mechanism, this flexibility can lead to vulnerability to design errors that may cause deadlocks.

In this paper, we identify and analyze deadlock problems in the Metropolis simulation environment. We propose a data structure called the dynamic synchronization dependency graph (DSDG) that reflects the runtime blocking dependencies among processes. We also devise an associated deadlock detection algorithm to monitor the simulation. The goal of the synchronization dependency analysis is to help the designer identify the components (e.g. processes

DAC 2005, June 13–17, 2005, Anaheim, California, USA.

and media) and synchronization constructs (e.g. *await* and *synch*) that are causing any deadlock problem and provide an error trace or a history of dependency snapshots that show how the system arrives at this state. We use a real world Metropolis design, the *resize* component in a picture-in-picture (PiP) video processing system, to demonstrate the usefulness and effectiveness of the deadlock analysis approach. We also use a high level mapping model that includes a functional specification, an abstract architecture model and the mapping to illustrate how the design problems from the function-architecture mapping can be analyzed.

1.1 Related Work

Deadlock detection and resolution techniques have already been extensively studied in the areas of operating systems and database systems [5, 17, 12, 16]. In those domains, deadlock prevention is possible if particular resource allocation policies are applied. Deadlock avoidance is used as a part of scheduling algorithms to choose at least one possible execution path where no deadlock will occur. A resource allocation graph or state graph is usually used to analyze and identify deadlock situations for deadlock detection. Though it is possible to incorporate these techniques in a system design to eliminate deadlocks, they are not general enough to apply to arbitrary designs due to the design flexibility required by today's platform-based embedded system designs. Our deadlock analysis mechanism is integrated in the design framework (rather than the designs) to help designers analyze design errors while allowing full design flexibility.

In communications and concurrent software, various formal verification techniques are employed to exhaustively search deadlock situations in concurrent protocols [8, 7, 15, 10]. In essence, synchronization protocols at a high level of abstraction, either extracted from the design or defined *a priori*, are formally verified. In the latter case, lower level implementations are then developed manually keeping as close as possible to the higher level protocols. The current approaches suffer from at least three problems. Firstly, the abstraction of synchronization protocols from a complex design is non-trivial and error-prone. Secondly, complex modern synchronization structures are becoming too complex and their analysis also suffers from state explosion problem. Thirdly, when a protocol is formally verified *a priori*, it is still quite difficult to get designers to follow exactly the verified protocols, not to mention that the design flexibility is considerably reduced. Our approach is based on simulation, so it can handle real complex system designs.

In simulation verification, assertions that are based on temporal logics can be used to check safety properties in a certain period of execution [1]. However, temporal assertions have to be designed according to particular applications. They are usually used to check the overall behavior of a system, and not suitable for identifying the causes of those undesirable behaviors due to their "trace checking" nature. A general deadlock detection mechanism is proposed in [13] for discrete event simulation models. However, no implementation on real simulation models are discussed in the literature. In the emerging simulation environment for heterogeneous system level designs, an effective and efficient deadlock analysis tool that can be tightly integrated in the design methodology is needed, which is the main focus of this paper.

In the next section, we introduce the synchronization constructs in Metropolis Meta-Model and show how deadlock situations can be caused by these modeling constructs. In Section 3, we present the dynamic synchronization dependency graph (DSDG) and an associated algorithm for deadlock analysis in Metropolis simulation. In Section 4, we use two case studies to demonstrate the approach. We conclude the paper in Section 5.

2. SYNCHRONIZATION IN METROPOLIS

In this section, we review the synchronization constructs in the Metropolis Meta-Model language and discuss how deadlock situations are caused by the synchronization mechanism in a concurrent system model.

2.1 Synchronization Constructs

The modeling constructs for synchronization in MMM include *synch* constraints, *await* statements, interface functions, quantity managers and LTL and LOC constraints. Most of these synchronization constructs are not unique to MMM, and their counterparts are also used in other concurrent modeling languages.

In Metropolis, the system function and the architecture are modeled as separate networks of processes communicating through media. In a functional network, functional processes run concurrently and communicate with each other through media. In an architectural network, computing and storage resources are modeled with media. Services that the architecture can provide are modeled with 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 *synch* constraints. A designer is allowed to implement particular schedulers as *quantity managers* to manage architectural resources and services in an architecture model. Quantity managers are basically scheduling media that implement a particular set of functions that can be invoked by processes to issue service requests. An architectural mapping process may be suspended by a quantity manager if it requests resources (quantities in Metropolis terminology) from it. The corresponding functional processes that are mapped to the mapping process can then be blocked through *synch* constraints.

A *synch* constraint is an alternative of a rendezvous used in the concurrent programming [9, 4]. 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 *synch* can also require that an event cannot occur until any of the other events occur. The execution of a process has to be blocked at a certain event until all the *synch* constraints containing the event are satisfied. For example, assume functional process p_0 and mapping processes p_1 and p_2 have events e_0, e_1 and e_2, respectively, and are synchronized by a *synch* constraint $synch(e_0 => e_1||e_2)$, 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 in the architecture. The execution of p_0 may be blocked by either p_1 or p_2, as illustrated in Figure 1.

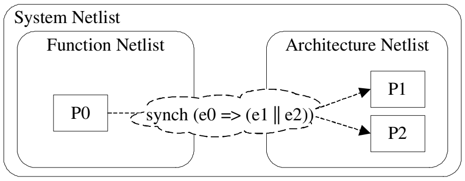

Figure 1: An example of *synch* constraint.

An *await* statement is used to establish mutually exclusive sections and to synchronize processes. It contains one or more statements called *critical sections*, each controlled by a triple (*guard*; *testlist*; setlist). The *guard* can be any Boolean expression, and the *testlist* and *setlist* denote sets of interfaces, which essentially work as integer semaphores that can be incremented or decremented. A critical section is said to be *enabled* if its *guard* is evaluated to true and none of the interfaces in the *testlist* has been set by other processes in the system. A critical section may start executing only if it is enabled. While the critical section is being executed, the "semaphores" specified in the *setlist* are incremented and can block other processes that require the semaphores. The interface function calls are also prevented if the interface is set by an *await*. If no critical section is enabled, the execution blocks. If more than one critical section are enabled, the choice is non-deterministic. For example, an *await* statement has two critical sections:

await {
(*foo*(); $intf_{00}$; $intf_{01}$) {$critical\_section_0$;}
(*true*; $intf_{10}$, $intf_{11}$; $intf_{10}$, $intf_{11}$) {$critical\_section_1$;}}

The first critical section is enabled only if guard *foo*() is evaluated to true and $intf_{00}$ is not set by other *awaits*. If a process enters this critical section, $intf_{01}$ will be set . The second critical section is enabled only if none of interfaces $intf_{10}$ and $intf_{11}$ is set by other processes. If a process enters this critical section, $intf_{10}$ and $intf_{11}$ will be set by the process. Note that an interface can be set by multiple processes at a given time and must be unset by all of them to be released.

A designer can also add general LTL and LOC constraints to a system to further restrict the behaviors of the system. We do not present these constraints directly in the paper since their specification semantics are not for execution and it is up to the simulator to make sure that the execution is consistent with the constraints.

2.2 Deadlock in Metropolis

Many different definitions can be found in the literature concerning deadlock. In our approach, we define deadlock for Metropolis designs as follows:

DEFINITION 2.1. *A* ***deadlock*** *is a situation where two or more processes are blocked in execution while each is waiting for some conditions to be changed by others.*

Given the constructs considered in MMM, only the following situations may block the execution of a running process:

1) A process has to wait for synchronization from other functional or architectural processes as required by one or more *synch* constraints.

2) A process cannot execute an interface function due to the fact that the interface is included in the setlist of a critical section being executed in another process's *await* statement.

3) A process is blocked at an *await* statement due to the unsatisfaction of all its guard/testlist conditions.

4) A mapping process is suspended by a quantity manager when it is requesting some quantity from it but cannot be satisfied.

The interaction of these synchronization constructs can be quite complicated. A deadlock exists if and only if there exist dependency loops among the processes in a system. We will identify and analyze the deadlock situation and report the processes and the media to which they are connected.

3. SYNCHRONIZATION DEPENDENCY AND DEADLOCK ANALYSIS

In this section, we introduce a deadlock analysis methodology for system level designs. We propose a data structure called the dynamic synchronization dependency graph (DSDG) used in Metropolis for deadlock analysis. Once the synchronization dependencies are captured by the graph, an algorithm can be used to detect deadlock situations.

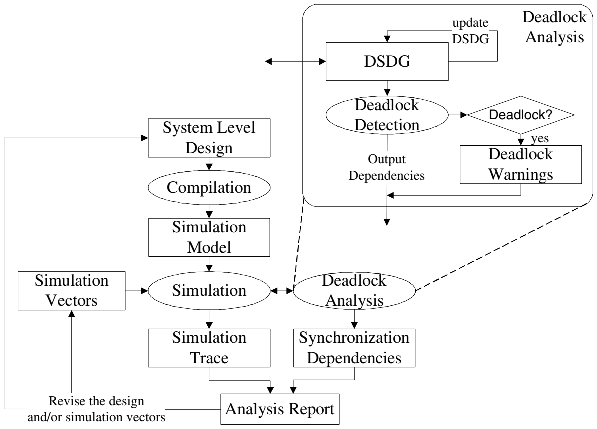

Figure 2: Deadlock Analysis Methodology.

3.1 Deadlock Analysis Methodology

Our deadlock analysis methodology is illustrated in Figure 2. By integrating deadlock analysis tools in a simulation environment for system level designs, designers can efficiently analyze complex concurrent systems with simulation and quickly identify design problems that may cause deadlocks. The task of design analysis becomes much easier with the help of runtime synchronization information combined with the regular simulation trace and the static network structure. They can be used to guide a designer to revise the design to eliminate the problems or modify the simulation vectors to explore different execution paths looking for other design errors. This methodology allows full design flexibility and is able to handle large models. The details of the deadlock analysis mechanism will be discussed in the rest of this section.

3.2 DSDG

DEFINITION 3.1. *A dynamic synchronization dependency graph (**DSDG**) is a directed graph S=(V, E). V is a set of four categories of vertices representing processes in the network, or-dependency, and-dependency, and eval-dependency. E is a set of directed edges between vertices indicating dynamic synchronization dependencies.*

In a DSDG, each process in the network is represented by a process vertex. Other dependency vertices and edges are added or deleted dynamically as dependencies between processes change in the execution. *And-dependency* requires a process to be blocked until all the conditions become satisfied. *Or-dependency* indicates that as long as one of the conditions becomes valid, a process can be released. *Eval-dependency* is used to represent that a process is blocked by a guard of an *await* or by a quantity manager. Guards are not analyzed but simply evaluated to get the valuation of "true", "false", or "blocked". Similarly, quantity managers are invoked to decide if processes that are making requests need to be suspended or not. If a process is blocked by a guard or a quantity manager, it has dependencies on the processes that may change the evaluation of the guard or the quantity manager sometime later. A DSDG is automatically built and updated during the simulation, and it describes the status of dependencies among all the concurrent processes of a system at a particular execution state. If a process is actively running, there is no outgoing edge from it in the graph. If it is blocked or released, dependency edges and vertices will be added to or deleted from the graph dynamically (see Algorithm 1 to 4). Initially, V only includes all the process vertices and E is set to $\emptyset$. During the simulation, UPDATE_DSDG() is called to update S every time the synchronization dependencies of the system

Algorithm 1 Main procedure to build and update a DSDG.

```
procedure UPDATE_DSDG()
  for each process $p_i$ in the system do
    if $p_i$ is unblocked by one or more synch. constructs then
      delete all the dependency vertices and edges from $p_i$ caused by those synch's;
    end if
    if $p_i$ is blocked by one or more synch. constructs then
      UPDATE_PROCESS($p_i$);
    end if
  end for
end procedure
```

Algorithm 2 Procedure to handle a blocked process.

```
procedure UPDATE_PROCESS($p_x$)
  for each synchronization construct that blocks $p_x$ do
    if $p_x$ is blocked by a *synch* constraint that requires its waiting for any of processes
    $p_1, p_2, \ldots,$ and $p_n$ then
      add an or-dependency vertex $o_x$;
      CONNECT($p_x$, $o_x$, $\{p_i : i \in [1,n]\}$);
    else if $p_x$ is blocked by an interface function $I$ then
      add an and-dependency vertex $a_x$;
      CONNECT($p_x$, $a_x$, {processes that prevent the interface $I$});
    else if $p_x$ is blocked by a quantity manager $Q$ then
      add an eval-dependency vertex $e_x$;
      CONNECT($p_x$, $e_x$, {processes that are managed by $Q$});
    else if $p_x$ is blocked by an *await* then
      UPDATE_AWAIT($p_x$, $\{C_i :\ i \in [1,n]$ and $C_i$ is a critical section$\}$);
    end if
  end for
end procedure
```

Algorithm 3 Procedure to connect newly added vertices

```
procedure CONNECT($src$, $mid$, $\{dest_i : i \in [1,n]\}$)
  add an edge from $src$ to $mid$;
  for i := 1 to n do
    add an edge from $mid$ to $dest_i$;
  end for
end procedure
```

Algorithm 4 Procedure to handle a blocking *await*.

```
procedure UPDATE_AWAIT($p_x$, $\{C_i : i \in [1,n]\}$)
  add an or-dependency vertex $o_x$;
  for each critical section $C_i$ ($1 \le i \le n$) do
    add an and-dependency vertex $a_i$;
    if the guard condition is evaluated to false then
      add a eval-dependency vertex $g_i$;
      CONNECT($a_i$, $g_i$, {processes that may change the guard});
    else if the evaluation of the guard is blocked then
      recursively call UPDATE_PROCESS($a_i$) to add dependency vertices and edges
      as if $a_i$ is a blocked process;
    end if
    for each prevented interface $intf_{ij}$ in $C_i$'s testlist do
      add an and-dependency vertex $a_{ij}$;
      CONNECT( $a_i$, $a_{ij}$, {the preventing processes});
    end for
  end for
  CONNECT($p_x$, $o_x$, $\{a_i : i \in [1,n]\}$);
end procedure
```

are changed. UPDATE_PROCESS() is called to update the DSDG for each blocked process and UPDATE_AWAIT() is called for a blocking *await*. CONNECT() connects newly added vertices with directed edges.

Figure 3A shows an example DSDG of a process p_0 being blocked by a constraint $synch(e_0 => e_1 || e_2)$ (the same example used in Section 2.1), which requires that e_0 cannot occur until e_1 or e_2 occurs. Figure 3B shows an example of a process p_0 being blocked by an *await* (the same example used in the previous section). The *await* has two critical sections C_0 and C_1. Assume that, in C_0, the guard is evaluated to be false and is accessible by p_2 and p_5, which is represented by a guard vertex. The interface *intf*$_{00}$ in its testlist is blocked by p_3 and p_4. In C_1, the guard is always evaluated to be true, but the interfaces *intf*$_{10}$ and *intf*$_{11}$ are blocked by other processes. Figure 3C shows an example where a process p_0 is blocked by 2 *synch* constraints at the same time.

Each dependency vertex is labeled to indicate the exact location in the source code that it is corresponding to. This information can be made available for the designer to help identify the problem quickly.

3.3 Deadlock Detection Algorithm

Given a dynamic synchronization dependency graph $S = (V, E)$ and a set of processes that are blocked from running P, we use Algorithm 5 to detect deadlock situations. Generally, the algorithm traverses the graph, searches for cyclic dependencies, and determines deadlocked processes. The algorithm not only decides if there is any deadlock but also identify all the processes and synchronization constructs that are involved in deadlock situations. In the worst case, the first step of the algorithm is to find all the simple cycles in the graph. Its complexity is $O(|V| \cdot (|V| + |E|))$ assuming that the adjacency-list representation is used for the graph. The rest of the algorithm will traverse all the simple cycles at most twice with a complexity of $O(|V|)$. If a simple cycle only contains process vertices and and-dependency vertices, then it is a deadlock. If a simple cycle also contains or- or eval- dependency vertices, there is a deadlock only if other edges from these or- or eval- dependency vertices all lead to cycles. Therefore, the complexity of the algorithm is $O(|V| \cdot (|V| + |E|))$. $|V|$ and $|E|$, the numbers of vertices and edges in a DSDG, are determined by the number of process instances, interface instances, critical sections of *await* statements and quantity managers in a system.

3.4 Implementation

The dynamic synchronization dependency graph and deadlock detection algorithm have been implemented in the Metropolis simulator. During the simulation of a design, a dependency graph is built and updated as the dependency state of the system changes, i.e. as one or more processes in the system are blocked from running or released from blocking. Whenever one or more processes are blocked from running, the deadlock detection algorithm is invoked to search the DSDG for any deadlock situation. Once a deadlock is detected in the simulation, the history of DSDG updates provides a trace that shows how the system execution goes into the deadlock. Due to the incremental nature of the DSDG update and deadlock detection algorithms, this simulation monitoring mechanism will not introduce significant overhead to the regular simulation.

4. CASE STUDIES

In this section, we use two examples, a real design of a complex function model for video processing and a high level model of function-architecture mapping, to demonstrate the usefulness and effectiveness of our deadlock analysis approach for system level designs.

4.1 Function Model for Video Processing

Figure 4 shows a picture-in-picture (PiP) video processing design. TS_DEMUX demultiplexes the single input transport stream (TS) into multiple packetized elementary streams (PES). PES_PARSER parses the packetized elementary streams to obtain MPEG video streams. Under the control of the user (USRCONTROL), decoded video streams can either be resized (RESIZE) or directly feed to JUGGLER that combines the images to produce the picture-in-picture videos. RESIZE is the major component of PiP that computes and adjusts the size of MPEG video frames according to user inputs. It consists of about 9,000 lines of Metropolis Meta-Model source code and contains 22 concurrent processes and more than 300 media.

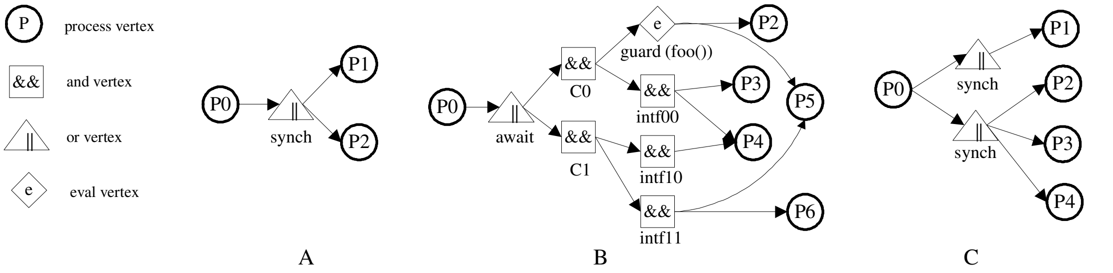

Figure 3: DSDG Examples.

Algorithm 5 Deadlock detection.

```
procedure DETECT_DEADLOCK($S$, $P$)
   search for simple cycles in $S$ from process vertices in $P$;
   let $\mathbb{L} = \{L_i=(V_i, E_i)\}$ be the set of all these simple cycles;
   if $\mathbb{L} = \emptyset$ then
      return NO_DEADLOCK;
   end if
   for each $L_i \in \mathbb{L}$ do
      if $L_i$ is already marked then
         continue;
      end if
      mark $L_i$;
      if each vertex in $V_i$ is either a process or and-dependency then
         the processes in $L_i$ are deadlocked, return;
      else
         $D$ := {eval- and or-dependency vertices in $V_i$ that have two or more outgoing
         edges};
         $\mathbb{L}'$ := {$L_i$};
         repeat
            find unmarked cycles in $\mathbb{L}$ that contains vertices in $D$;
            mark all these cycles;
            $D := D \cup$ {eval- and or- dependency vertices with two or more outgoing
            edges in these cycles};
            $\mathbb{L}' := \mathbb{L}' \cup$ {these cycles};
         until $\mathbb{L}'$ becomes stable
         if $\exists$ vertex in $D$ that has an outgoing edge $\notin \mathbb{L}'$ then
            continue;
         end if
         the processes in $\mathbb{L}'$ are deadlocked, return;
      end if
   end for
   return NO_DEADLOCK;
end procedure
```

The video frames and control signals are passed between processes through around 80 communication channels specified with media. The communication channels are modeled at the task transition level (TTL) with bounded first-in-first-out (FIFO) buffers [6]. The mutual exclusion and boundary checking of the bounded FIFO buffer is guaranteed by a central protocol. To simulate the RESIZE unit, three additional processes are used to mimic user inputs (USER), send MPEG video streams to the unit (SOURCE) and absorb the data from it (SINK) as shown in Figure 5A.

In the simulation with our runtime deadlock monitoring mechanism enabled, a deadlock is reported immediately after TMUX_UV and TMEM_CTL_U block each other through two await statements and their synchronization dependencies are captured in the DSDG as shown in Figure 5C. As it turns out, there is a design error in process TMUX_UV, which fails to read all the data sent by TMEM_CTL_U.[1] The data in the bounded buffer of the channel between the two processes accumulates until the buffer becomes full. Then a deadlock occurs where TMEM_CTL_U is blocked waiting for the buffer space to be released by TMUX_UV while TMUX_UV is also blocked waiting for reading signals from TMEM_CTL_U. The designer can now focus on the two processes and the communication channels between them to identify and correct those design errors. A solution is to modify process TMUX_UV and make it absorb all the data from its input channels even if not all the data is useful. We observe that, without the deadlock detection mechanism, the simulation will continue and the regular simulation trace won't show any apparent sign of deadlock until most of the processes in the system are eventually blocked. By that time, the simulation trace is long and a large number of processes are blocked. Our approach automatically catch the deadlock as it first occurs. Designers can then focus on solving the deadlock without complicating themselves by the consequences of the deadlock. The simulation and analysis results are summarized in Table 1.

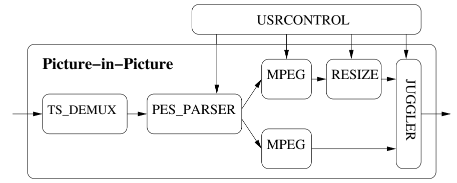

Figure 4: Picture-in-Picture Design.

4.2 Function-Architecture Mapping

In the platform-based design, the mapping is the key procedure that correlates the function to the architecture. In this design example (as shown in Figure 6A), two source processes (S1 and S2) write the data into two independent channels. A separate process (Join) then reads data items from both channels, manipulates them, and then sends the result data to another process (Sink) through another channel. In the abstract architecture model, there are two CPU/RTOS units, a bus unit, a memory unit and a quantity manager (i.e. scheduler) for each architectural unit.[2] A CPU unit can be shared among several software tasks that may request services from it. When more than one service request is issued to a CPU, arbitration is needed. The mapping procedure synchronizes the processes in the function model and the mapping processes (representing software tasks) in the architecture model. In this example (as shown in Figure 6A), functional processes S1 and S2 are mapped to mapping processes SwTask1 and SwTask2, respectively, which are associated to CPU1 and the other two processes are mapped to CPU2. The CPU quantity managers implement a non-preemptive static-priority dynamic scheduling policy. The two CPUs are connected to the bus and the bus is connected to the memory unit.

[1] As Figure 5B shows, process TMUX_UV gets video data from both TMEM_CTL_U and TMEM_CTL_V, combines two streams of data and sends them to its successor process.

[2] An architectural unit is modeled as a medium in Metropolis.

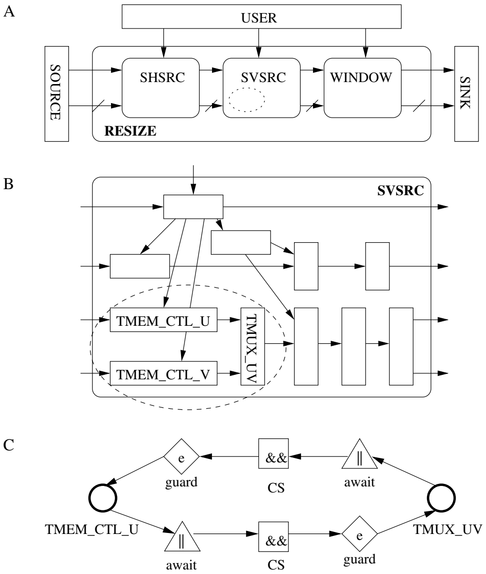

Figure 5: The RESIZE unit and its synchronization dependencies.

Our deadlock detection mechanism reports a deadlock within one minute of simulation. Due to the boundedness of the channels between processes, process S1 can not complete a task of writing data before Join reads from and releases the channel buffer. Therefore, with the current CPU scheduling policy, the deadlock occurs when S1 obtains the CPU service but cannot complete a writing task while Join is still waiting for data from S2 who cannot get CPU service. The deadlock situation involves five processes, two await statements, two *synch* constraints and a quantity manager as shown in Figure 6B. This analysis also suggests several possible deadlock resolutions. The deadlock can be resolved by making the channel buffer large enough to store all the data from a single writing task, increasing the number of CPUs, or changing the CPU scheduling policy. We also observe that such deadlocks only occur in the mapped design and are not inherent in the function specification or in the architecture model. The simulation and analysis results for this mapping model are also listed in Table 1.

Table 1: Simulation and analysis summary for both case studies.

Example	RESIZE Unit	Mapping Model
Code Size	9000 lines	5900 lines
Processes	22	8
Media	300+	16
Deadlocked Processes	2	5
Time to Catch Deadlock	2min	< 1min

5. CONCLUSIONS

In this paper, we study the deadlock problem in system level designs that include complex synchronization constructs and function-architecture separation and mapping. We present our deadlock detection approach with a data structure called the dynamic synchronization dependency graph and an associated deadlock detection algorithm. We use two examples, a complex function model for video processing and a model of function-architecture mapping, to demonstrate the usefulness and effectiveness of our deadlock detection approach for system level designs.

We believe that the dynamic synchronization dependency graph can be used to not only detect deadlock situations but also help search for livelock or starvation problems. We are currently working on combining our simulation-based deadlock detection mechanism and existing formal verification techniques to search for more subtle problems such as livelock and starvation in system level designs.

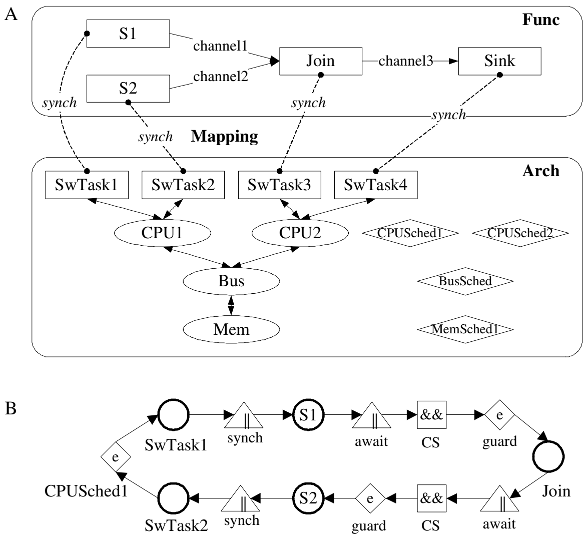

Figure 6: A mapping model and its synchronization dependencies.

6. REFERENCES

[1] http://www.eda.org/vfv, 2003.

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

[3] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: an Integrated Electronic System Design Environment. *IEEE Computer*, 36(4):45– 52, Apr. 2003.

[4] A. Charlesworth. The multiway rendezvous. *ACM Transactions on Programming Languages and Systems*, 9(3):350–366, 1987.

[5] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971.

[6] O. Gangwal, A. Nieuwland, and P. Lippens. A scalable and flexible data synchronization scheme for embedded hw-sw shared-memory systems. In *Proceedings of International Symposium on System Synthesis*, Oct. 2001.

[7] P. Godefroid and D. Pirottin. Refining dependencies improves partial-order verification methods. In *Proceedings of the 5th Conference on Computer Aided Verification*, volume 697 of *Lecture Notes in Computer Science*, pages 438–449. Springer-Verlag, June 1993.

[8] A. N. Habermann. Prevention of system deadlocks. *Communications of the ACM*, 12(7):373–377, 1969.

[9] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[10] G. J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–258, May 1997.

[11] K. Keutzer, S. Malik, A. R. Newton, J. Rabaey, and A. Sangiovanni-Vincentelli. System level design: orthogonalization of concerns and platform-based design. *IEEE Transactions on Computer-Aided Design*, 19(12):1523–1543, Dec. 2000.

[12] E. Knapp. Deadlock detection in distributed databases. *ACM Computing Surveys*, 19(4):303–328, 1987.

[13] M. Krishnamurthi, A. Basavatia, and S. Thallikar. Deadlock detection and resolution in simulation models. In *Proceedings of the 26th Conference on Winter Simulation*, pages 708–715. Society for Computer Simulation International, 1994.

[14] Z. Manna and A. Pnueli. The temporal logic of reactive and concurrent systems: Specification. *Springer-Verlag*, 1992.

[15] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[16] J. L. Peterson and A. Silbershatz. *Operating System Concepts*. Addison-Wesley, 1983.

[17] M. Sfinghal. Deadlock detection in distributed systems. *IEEE Computer*, 22(11):37–48, 1989.