ROYAL INSTITUTE OF TECHNOLOGY

TRITA-ESD-1998-07

ISSN 1104-8697

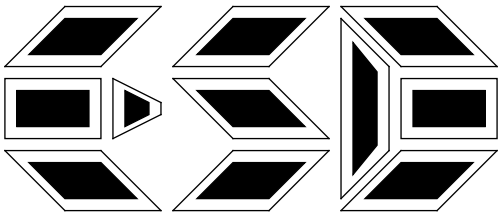
ISRN KTH/ESD/R--98/7--SE

A Survey of HW/SW Cosimulation Techniques and Tools

Thesis Work (Studienarbeit)
Heiko Hübert

Supervisor:
Dr. Axel Jantsch

Stockholm, June 1998



ELECTRONIC SYSTEMS DESIGN LABORATORY
ROYAL INSTITUTE OF TECHNOLOGY
ESDLAB/KTH-ELECTRUM
ELECTRUM 229
S-164 40 KISTA, SWEDEN

Abstract

In the last decade, electronic systems have become increasingly complex. The number of functionalities built on one chip have risen enormously. To solve the problem of cost and flexibility for such sophisticated, systems the usage of mixed hardware software systems has increased.

Due to the complexity, the system development has become more and more difficult and the simulation and evaluation has become a key position of the design process. Many tools which aid designers in the evaluation of electronic systems have been developed. However, until recently, tools for the simulation of mixed HW/SW systems have been lacking. An early approach to HW/SW co-design and cosimulation was presented by the University of California, Berkeley [8] in 1991 within their system design project **Ptolemy**.

During the last years some vendors have begun offering tools to close this gap. In 1996, Mentor Graphics presented **Seamless**, a cosimulation tool that supports as a backplane the connection between the software execution and the hardware simulator. Competing products are **EagleI** and **EagleV**, which have been developed concurrently by Eagle Design Automation (now merged with Viewlogic). The Tima Lab at Institute National Polytechnique in Grenoble, France, developed in co-operation with SGS-Thomson, a VHDL/C cosimulation tool called **CoSim** [11] as a part of their co-design tool **COSMOS**.

This paper surveys existing cosimulation tools and compares and contrasts their basic features and techniques. For example, commonly used communication mechanisms between HW and SW simulators and different processor models are introduced.

The first chapter gives a brief overview of cosimulation and the co-design process in general. Chapter 2 surveys the fundamental tasks of cosimulation. Processor models and tool structures are considered in Chapter 3. A number of tools and descriptions of their implementation are presented in Chapter 4.

Contents **Page**

1. Introduction

In the field of electronic systems, the term cosimulation is defined in several ways. In addition to HW/SW cosimulation, which is considered in this paper, it may indicate the simulation of mixed analog/digital systems. However, all cosimulations perform a simultaneous simulation of two or more parts of a system described on different levels of abstraction (e.g. hardware, software or analog components). The challenge of cosimulation is, building a bridge between the different simulators. The descriptions and simulations of the parts differ in many ways, e.g. data and signal types, abstraction level and scheduling. A cosimulation tool has to solve problems like synchronization and type conversions between the simulators to achieve a proper communication.

Asawaree Kalavade demands from a cosimulation environment:

> ”Simulation plays an important role in the system-level design process. At the specification level, it is possible that the design may be specified using a combination of one or more semantic models. Tools that allow the simulation of such heterogeneous specifications are required. At the implementation level, simulation tools that support mixed hardware-software systems are needed. Throughout the design process, it should be possible to simulate systems where the components are described at different levels of abstraction. As parts of the design evolve from specification to their final implementation, functional models can be replaced by more lower level structures. A simulation environment should be capable of supporting these aspects.” [12]

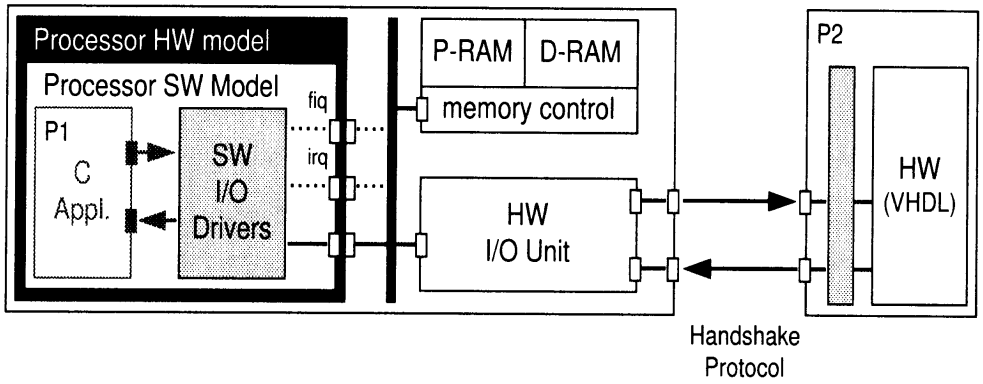


Figure 1: Cosimulation Environment [29]

In mixed HW/SW systems, processors build the bridge between the software, which runs on the processors, and the hardware, which e.g. consist of I/O ports, ASICs or other logic or analog components (figure 1). Therefore, the choice of an appropriate model of processor is of high importance for achieving the desired simulation result. Processor models differ in the level of abstraction, described in 3.1, and lead to differences in performance and accuracy of the cosimulation.

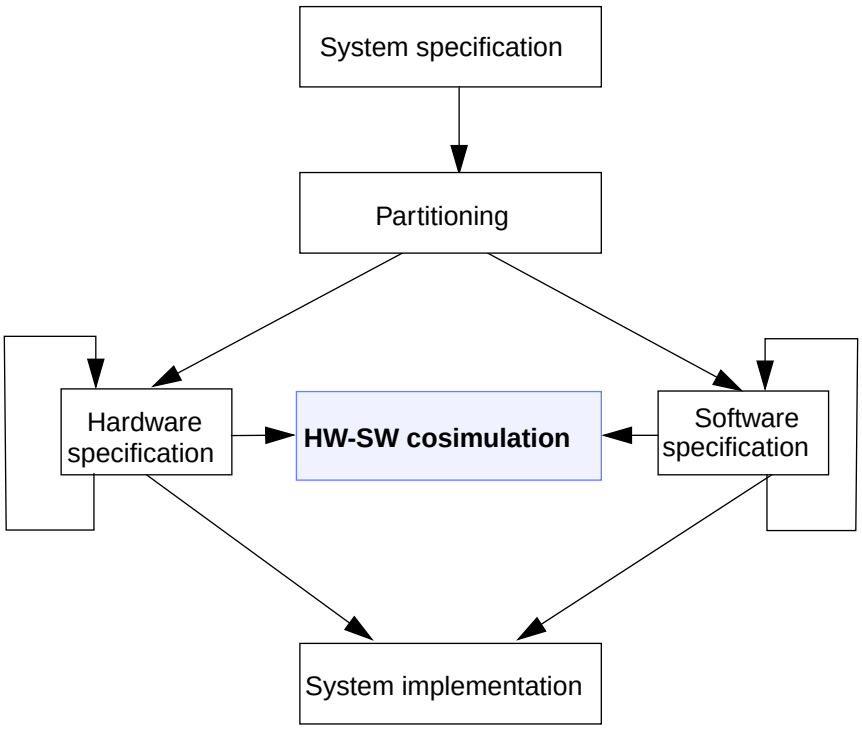


Figure 2: HW/SW Co-Design

HW/SW cosimulation has to be seen as a part of HW/SW co-design (see figure 2). Further, a glimpse at the design steps surrounding the cosimulation is helpful. Usually the design of electronic systems begins with a specification of the entire system in a natural language. The specification has to be refined by the consideration of requirements and finally translated into a language which allows a functional verification, for example SDL. A first simulation of the system is frequently performed at this stage in order to evaluate the functionality on a high level of abstraction. After that, the partitioning in hardware and software is performed in such a way that an acceptable trade-off between performance, flexibility and cost is achieved. Implemen-

tation in hardware leads to a fast system, whereas software solutions are cheaper and remain flexible after implementation. After the system has been divided into SW and HW parts, cosimulation can be used for more detailed simulations. The tasks of cosimulation are to validate the functionality of HW and SW components, to ensure that both work together and achieve the expected system functionality. The first step of the cosimulation is the validation of the components and their interactions on a high level of abstraction, which results in a high simulation speed. However, time accuracy cannot be tested in this way. Therefore, more detailed simulations have to follow, which sometimes achieve a nano-second accurate timing. Each bug results in a re-design of the HW/SW description, or even the system specification.

After a successfully accomplished cosimulation, the SW can be compiled to machine code for the target machine, and the HW can be implemented, e.g. on a FPGA, or as an ASIC. Final tests are run to validate the functionality of the ‘real’ system.

2. Dimensions of Cosimulation

This chapter presents the basic methods that determine cosimulation. It is hierarchically ordered, beginning with the underlying communication mechanisms, which provide the connection between the system components under simulation, and their synchronization. Further timing control mechanisms, ensured by scheduling policies, are considered. Based on this, the computational models define the behaviour of the components and their interaction.

2.1. Communication

The communication between HW and SW simulators can be considered as one of the most important issues of cosimulation. The manner in which different simulators are connected influences the performance and accuracy of the result enormously.

Currently, cosimulation tools are primarily developed to work under Unix. Unix supports rather usable communication mechanisms between processes and applications, which can be used for all the interconnections required in cosimulation tools. Therefore, the frequently utilized Unix communication mechanisms are briefly presented.

2.1.1 Communication under Unix

Communication mechanisms under Unix are gathered under the term InterProcess Communication (**IPC**), which includes several ways to connect processes running under Unix workstations. Some of these mechanisms focus on the communication between processes running on the same workstation, e.g. **pipes**, while others allow message passing between processes running on separate machines, for example Socket connections. To give an example of such a mechanism, the **socket interface** connection based on TCP/IP is subsequently introduced [20].

TCP is part of the TCP/IP Internet Protocol Suite, which collects the standards for Internet communication. TCP defines a protocol for a reliable transport of data between two processes. Since processes cannot be the endpoint of a connection itself, TCP defines protocol ports as endpoints. A unique destination is given by the identifier of the host machine (IP address) as well as the port number. However, one destination can be used for more than one connection, because TCP identifies connections by the pair of its endpoints. This has the advantage that a

process using one port can have multiple connections.

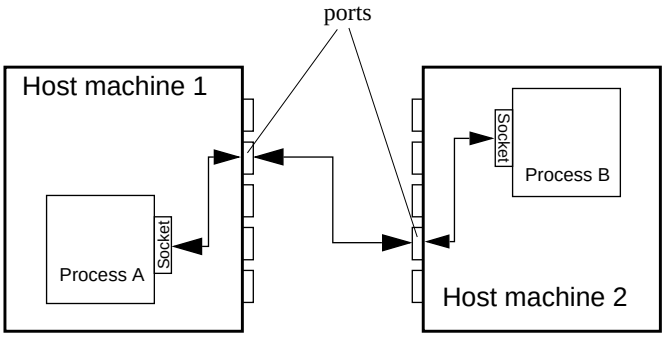


Figure 3: Socket Interface

Socket interfaces use the TCP protocol to establish connections between processes. Such Interfaces are mainly used for client-server connections with one server and multiple clients. The server initiates a connection by opening a socket and binding it to a port (figure 3). The interface software checks this port continuously for incoming client requests. A client can establish a connection by addressing the server socket through its IP address and port number. Once they are connected, data can be transferred from both sides. They use Unix I/O primitives (e.g. read, write) to send and receive data. Thus, for the application program, the communication appears as a typical file operation.

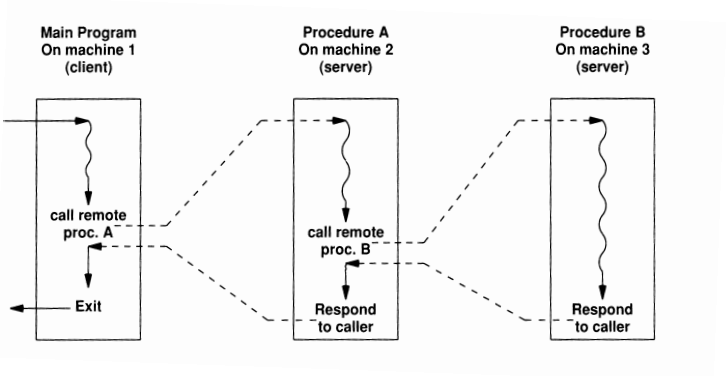


Figure 4: Remote Procedure Calls (RPCs) [21]

The hierarchical structure of communication layers (organized via the OSI-model) allow the programmer to establish connections between processes on different levels of detail. For example, remote procedure calls (**RPC**) allow process interaction on a higher level of abstraction. They are based on socket interfaces, but appear for the programmer almost as normal procedure

calls, as illustrated in figure 4.

2.1.2 Communication between Simulators

Based on the InterProcess Communication (IPC) interfaces introduced in the previous chapter, the cosimulation tool designer has to include these interfaces in the simulators. It is useful to discuss in some detail possible ways of connecting C-code with VHDL or Verilog simulators.

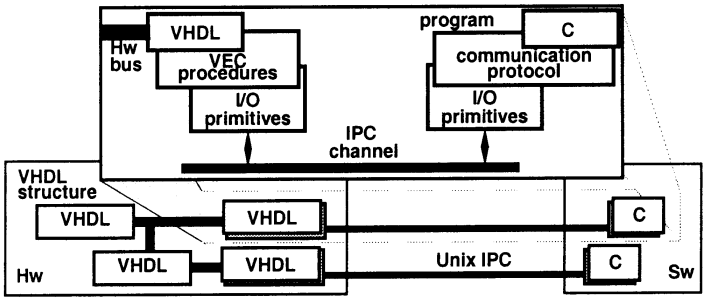


Figure 5: VHDL-C Interface [17]

VHDL supports the link of procedures, written in C-code, to the simulator through its **foreign language kernel**. Valderrama [17] presents a sophisticated way to use this feature for cosimulation tasks. Figure 5 illustrates his approach, which is realized by using special entities and the so-called VHDL emulation C-procedures (VEC procedures). The entity encapsulates the ports used by the I/O primitives, which should be connected to variables in the C-program. The access from VHDL to IPC channel is performed through the VEC procedures, which are also responsible for VHDL-C type conversion and synchronization. In addition to the original software, the C-process also contains a communication protocol, based on the mechanism described in 2.1.1, which ensures the connection to the IPC-channel.

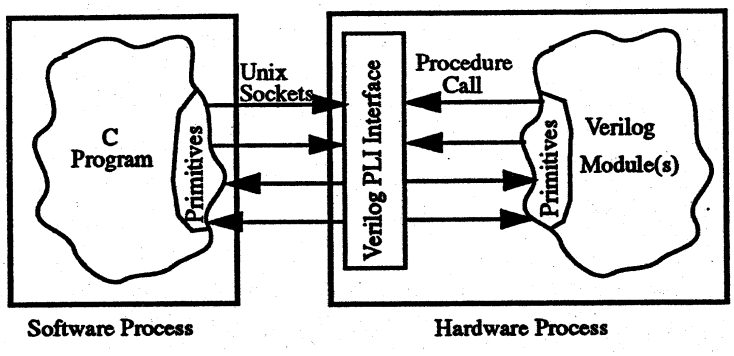


Figure 6: Verilog-C Communication [6]

Becker [18] describes a communication method between Verilog and C-programs. Verilog supports invocation of C-functions by a so-called Programming Language Interface (PLI). He uses this to establish a connection from Verilog to an IPC channel in almost the same way Valderrama [17] did it with VEC procedures in VHDL. He uses socket interfaces, described in 2.1.1, to connect the C-program (SW) to the IPC channel, see figure 6.

Based on such VHDL/Verilog-C connections, a cosimulation environment can be build. An example of such a cosimulator is the **Application Program Interface (API)** presented in [4]. It supports controlling functions like synchronization in addition to a simple connection. Chapter 4 gives a more detailed description of different tools.

2.2 Synchronization

Mechanisms are required to handle concurrency in distributed timed cosimulation. The control of timing in interactions between system parts, e.g. HW and SW components, is essential to achieve a correct simulation. Components may interact through shared memory or message passing (data transfer) systems. In communication through **shared memory** a read-modify-write mechanism is commonly used for synchronization. The accesses are totally ordered and the memory location is locked during read and write operation.

In the following message passing systems are considered in greater detail as they are employed by most of the tools introduced in this paper. They use one of the underlying communication mechanisms described in 2.1 to establish a connection. Thomas and Coumeri [6], distinguish between synchronized data transfer, unsynchronized data transfer and synchronization without data transfer.

In **unsynchronized data transfer** the sender and receiver are not coordinated, thus the sender does not know if data is received correctly. Further, this transfer is unbuffered and therefore, sending data may overwrite previous data before the receiver reads it. Hence, despite the fact that this transfer mode is fast and easy to implement, it is unacceptable for most interactions. Additional synchronization signals without data transfer can be used to improve this transfer mode.

Synchronized data transfer includes methods to ensure a reliable exchange of data. In [24] FIFO buffered and rendezvous mechanisms are given as examples of synchronized data transfer. The first uses a FIFO buffer of a finite size for storing data between writing and reading. The latter demands writing and reading operations to be performed simultaneously. This is comparable to the way assignments operate in programming languages. However, one of the connected processes has to block until the other process is ready to communicate, since the two operations do not always initiate simultaneously.

Blocking is another important issue of synchronization which is necessary for most of the methods. Reading or writing operations may or may not block the calling process. Writing operations in particular are critical, if they are not blocked until a reading operation is performed. They may overwrite previous data, for example if the buffer used in FIFO buffered synchronization is full. Blocking all reading operations, which occur in a given process, can help to impose totally ordered input signal treatment. The disadvantage of blocking is an increasing simulation time. Therefore, blocking should only be used if needed because of dependencies between the interacting processes.

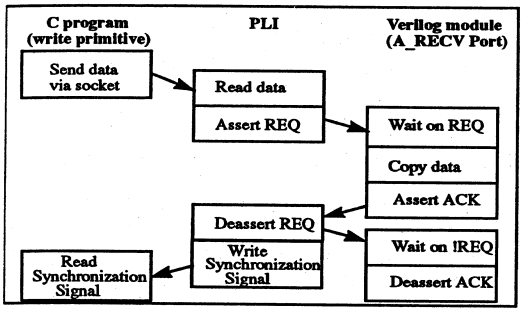


Figure 7: Handshake mechanism

To control synchronization, the **handshake** mechanism is commonly used (figure 7). The master starts every interaction with the slave through a request. This request has to be acknowl-

edged by the slave before data can be transferred. After the transfer, the receiver has to acknowledge that the transfer was successful.

2.3 Scheduling

In comparison to synchronization, scheduling describes a more global mechanism of control. Not only the proper interaction between two components has to be considered, but also the entire simulation process has to be controlled, which is primarily realized by a **global timing concept**. This method provides system timing by keeping either a (local time/global time) pair [15] or a (local time/time stamped event) pair [2,26,28] of time variables, which differ only in their description of the method. In both methods every component carries its own local times. Using the global time/local time concept at every execution step, the local times are compared to a global system time, and the component with a local time equal to the global time is activated. In the second description, every event on a signal consists of a value and a time stamp, which indicates the time when the event occurs. The component compares local time and time stamp to decide which event to consider next.

Two groups of schedule policies exist: conservative and optimistic scheduling. The local time/global time description [15,23] is used to explain them.

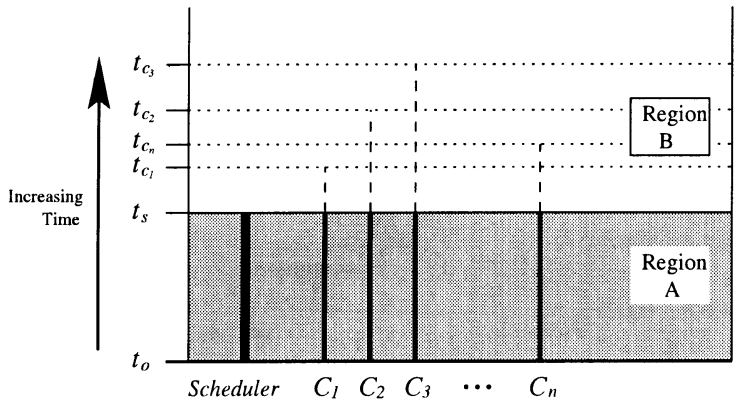


Figure 8: Conservative Scheduling

Conservative scheduling:

Figure 8 shows a snapshot of the global time (t_s) and local times (t_{Cn}) in conservative scheduling. To ensure safe timing the global time must follow several rules:

1. Global time is always less than or equal to all local times.
2. Global time is always monotonously increasing in real time.
3. Before a component can consume events from outside, that component's local time must be equal to global time.

Due to rule 3, components are always able to produce events, but must block consumption until the global time catches up. The global time 'pushes' all local times if it reaches them, regardless if they were activated or not.

Optimistic scheduling:

This group of schedule policies increases simulation speed. They countermand rule 3 in order to assume that certain components are independent of other components, and may thus components be activated without waiting for the global time. Thus less components stay in a blocked mode. If the prediction of data independence proves to be wrong, the system has to be restored to a state known as safe. To ensure such rollbacks the scheduler has to store states occasionally, which requires perhaps a large amount of memory.

Another rollback method is presented in [26]. Messages (time-stamped events) are distinguished in positive and negative (or anti-) messages. If an already sent message proves to be incorrect an anti message is produced to cancel the message

2.4. Models of Computation

In general a design consists of a set of components. A model of computation is needed to define the behaviour of the components and the interaction between them [24]. The most important models will be considered in this chapter. Some cosimulation tools enable the designer to choose the computational model, which should perform the simulation of the current design. Ptolemy even supports a free choice of a model for every single component. Other tools allow only the usage of one specific model for the entire simulation.

2.4.1 Discrete Event

The discrete event (DE) model is an asynchronous model. Events on signals are only produced if the system state should change. A global event queue sorts the events based on their time stamps and produces them if their time stamp is equal to the global system time. Hence all events are globally ordered. A great disadvantage of this model is the high effort spent on for sorting the events. Therefore the DE model is most appropriate for large systems in which a lot of components are idle most of the time. In this case, only a few events have to be sorted and only the system parts where events occur are processed instead of the whole system.

To handle simultaneous events the DE model requires a special mechanism. VHDL, a language that uses a DE model of computation, solves this problem by a two-level model of time. Every time step is divided in a potential infinite number of delta delays, which allow an ordered production of simultaneous events.

2.4.2 Communicating Finite State Machines

Due to their structure Finite State Machines (FSM) are suited for modeling systems with a sequential behaviour. Modeling systems with concurrent behaviour or memories result in a so called state explosion, i.e. a huge amount of states are needed to model concurrence. However, different ways exist to reduce the number of states. First, FSMs can be structured hierarchically in such a way that states on a higher level represent separate FSMs on a lower level. Second, concurrency can be achieved by the existence of several concurrent FSMs. Another possibility to reduce states is to allow non-determinism in the simulation. Finding the appropriate abstraction level for each component for the current simulation decreases both the system and the FSM complexity. Thus, details without importance are not simulated.

The co-design FSM (CFSM), used by [22] in the Polis co-design environment, is derived from this computational model.

2.4.3 Synchronous

In synchronous models all events are triggered by a clock synchronous [24]. In contrast to DE models, an event occurs on each signal during every clock step, even though the value car-

ried by the signal may be uniform. The processing of simultaneous events, produced at the same clock step varies from totally ordered to disordered dependent on the implementation of the model. This model fits for synchronous clocked systems since they correspond most.

2.4.4 Dataflow Process Network

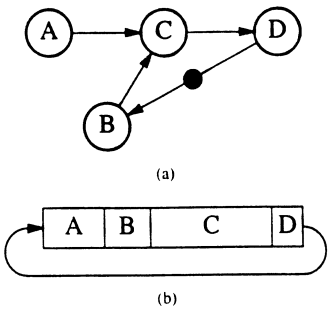


Figure 9: (a) directed graph,
(b) corresponding single-processor static schedule

The system is described through a directed graph if a dataflow model is used (see figure 9). The directed graph consists of nodes, which represent computations, and arcs connecting the nodes and representing an ordered sequence of events (called token). In this model the event production is called token firing. A sequence of firings, which is scheduled in a list, results in a process.

Models derived from the dataflow models are the synchronous dataflow (SDF), dynamical dataflow (DDF) and the Boolean dataflow (BDF), all realized in the Ptolemy cosimulation tools [8]. SDF is a sub model of the DDF with the constraint of a fixed number of token production and consumption for each firing. BDF is an intermediate between SDF and DDF with an additional support of asynchronous operations.

3. Cosimulation Techniques

The first part of this chapter introduces different processor models, which glue the software with the hardware. The models are distinguished in their levels of abstraction and the connected performance/timing trade-off is shown. Chapter 3.2 describes typical structures of cosimulation environments. Finally speed-up methods for higher simulation performance are glanced at.

3.1 Processor Models

Since the processor links the HW/SW components of a system it has a central position in most cosimulations. The choice of an appropriate processor model at a given stage in co-design is of crucial importance for achieving the desired simulation result. Several kinds of processor models on different abstraction levels exist and system designers have to make their decision due to the following factors [16]:

1. Model availability
2. Performance (considering the simulation time)
3. Timing accuracy
4. Debugging features (internal state access)

The availability of the desired processor model can give rise to the first obstacles. If several processor models exist the main trade-off is made between performance and timing accuracy since they are mostly incompatible. A high visibility of internal state, e.g. register values, is desirable but not always supported.

Processor models can be divided into two groups, hardware models (figure 10) and models which replace the processor with a software decription.[4]

3.1.1 Hardware models

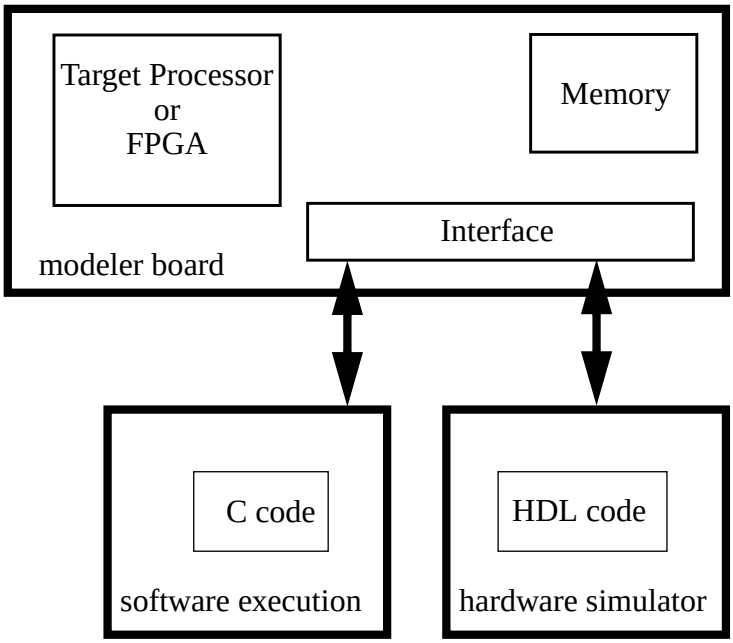


Figure 10: Hardware Model

The **target processor** set on a modeler board gives the most accurate timing results. The modeler board works as an interface between the processor and the rest of the system simulated on a computer [4]. The software is loaded into the memory on the board and runs on the processor. Disadvantages: This approach assume the target processor to be available before a first prototype of the system exists. The implementation of the modeler board, the, is costly and determine the simulation speed [25]. Since the processor cannot be slowed down to the interface speed, dynamic processes require a certain clock frequency, which has to be reset at every clock cycle. To ensure that the processor continues correctly after the reset, the input history up to the current position has to be stored, so that it can be reloaded after the reset. Due to a reduced memory space for storing the increasing history, only short SW programs can be run with a low performance on such models.

In **logic emulation** [25] the processor's functionality described at gate-level is mapped onto FPGAs. The simulation speed of the FPGA approaches the execution speed of the target processor. However, to avoid the store/reset/reload mechanisms, which are required with the hardware modeler, the clock frequency can be reduced to the simulation speed of the other simulators. Disadvantages are the low visibility of internal values, which makes debugging more difficult, and the non-accurate timing.

3.1.2 Processors Modeled in Software

Several processor models, which are defined in software with a varying accuracy from gate-level to instruction set description, are available. The factors named above (e.g. performance and time accuracy) are determined by the level of detail of the model. However, simulation is also possible without a behaviour model of the target processor. In this case the software can be compiled to run on a workstation. If a functional model is not available either (or not desired) the processor model can be replaced by an operating system.

In [16,25] the techniques are distinguished and described as follows:

	speed (instructions. per second)	debugging ability	processor model requirements
hardware modeller	10-50	no processor state	timing only
logic emulation	fast	limited	none
nano-second accurate	1-100	excellent	hardest
cycle accurate	50-1000	good	hard
instruction set	2000-20,000	OK	medium
synchronized handshake	limited by hardware simulator	no processor state	none
virtual operating system	fast	no processor (and no hardware) state	easier
bus functional	limited by hardware simulator	no processor state	easier

Table 1: Processor Descriptions [16]

a) Software processor model:

Software models, for example provided by the processor designer, give a functional description of the processor. They are frequently written in a hardware description language (HDL) or C and available on different levels of abstraction. Internal states of the processors are visible and can be used for debugging, which is a significant advantage compared to hardware models.

The models are grouped in:

1. **The nano-second accurate processor model**:

This is the most detailed functional description, often on gate level, which ensures a very high timing accuracy. Such models are not recommended for early phases in co-design, because they suffer from low performance. The simulator has to calculate every single internal transition, therefore it only reaches a simulation speed of a few instructions per second.

2. **The cycle accurate processor model:**

This model ensures that transitions occur at the right clock edges, but does not consider the exact delay time needed for a transition (zero delay). This way all events are produced only at given points of time reducing the synchronization overhead dramatically. This model runs 50 to 1000 instructions per second.

3. **The instruction set accurate (ISS) processor model:**

Such models, often written in C, guarantee the processors functionality only on instruction-set level (figure 11). Register values are simulated correctly, however timing is not considered. ISS models work on a high performance in 2000 to 20,000 instructions per second. They are commonly used for software simulation.

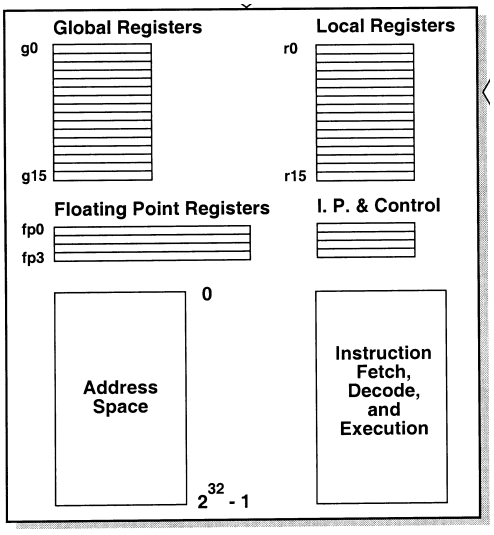


Figure 11: Instruction Set Accurate Processor Model [13]

b) Techniques requiring no processor model:

Instead of using models of the target processor the designer may choose other techniques to glue SW and HW components. This can simply be due to the fact that no model exists. Other reasons are, for example, that the processor's behaviour is well known and so its detailed simulation is not required, or that the choice of an appropriate processor should be left to the final co-design phase to take new developments into account. A timing evaluation is impossible with these techniques.

Often used mechanisms are:

1. **The model-free synchronized handshake:**
   (also called **host code execution (HCE)**)
   Together with ISS models this technique is often supported by cosimulation tools [13,19]. The SW, supposed to run on the target machine, is compiled to the host workstation. The advantage is the increasing performance, which is dependent only on the HW simulation since the SW is executed at workstation speed. Communication between SW and HW is performed by a synchronized handshake mechanism.

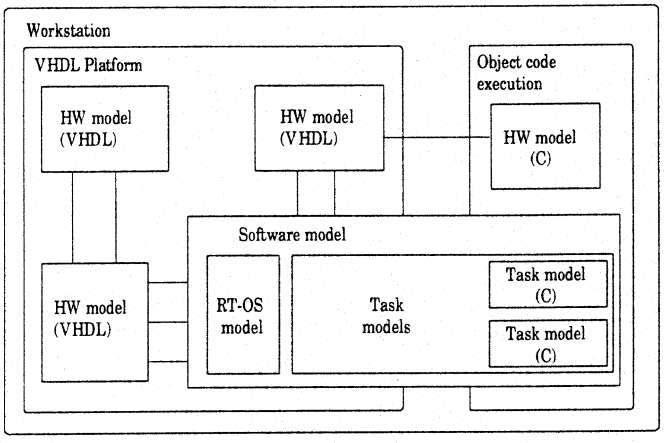


Figure 12: Virtual operating system [5]

2. **The virtual operating system:**
   A virtual operating system replaces the processor. Hardware is either emulated [16] or simulated [5]. In the first case hardware does not exist at all, consequently

this method can just be used to simulate the software. This method achieves the highest performance, since the entire system simulation runs on workstation speed. In case of hardware simulation, the virtual operating system performs the interfacing between HW and SW (figure 12).

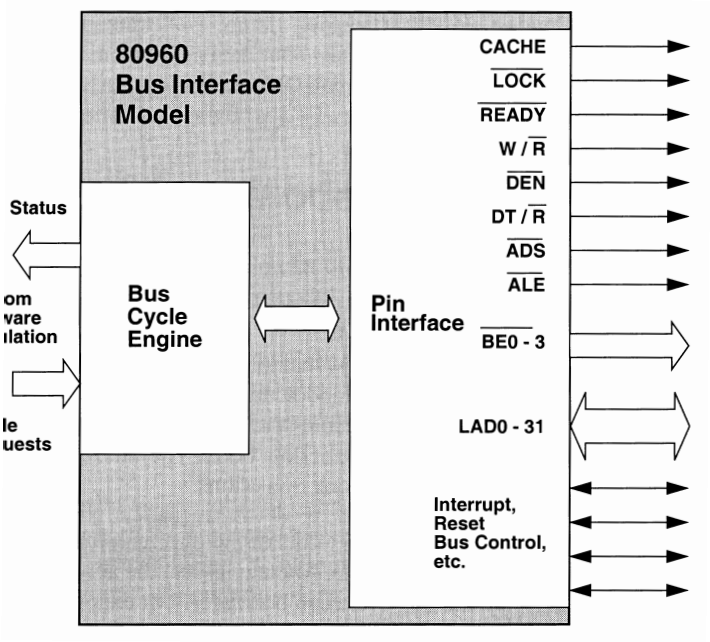


Figure13: Bus Functional Model (BFM)[13]

A processor description beyond this array of models is the **bus functional processor model (BFM)**, which is illustrated in figure 13. It is not derived from the functional processor description like the one presented in 3.2.1 a) but based on the processor's bus specification document [27]. Only the pin activities are modeled, therefore it fits for detailed bus and hardware simulations.

A common approach used in many cosimulation tools connects the BFM with an ISS model, thus both, a fast software and hardware simulation, can be achieved without the disadvantage of low timing accuracy [13,19].

3.2 Cosimulation Environment Structures

This chapter surveys the global structures of a cosimulation environment. The different approaches point to different views and intentions of cosimulation, however they have the simul-

taneous simulation of hardware and software as their main task in common. Commercial cosimulation tools are often based on a so called simulation backplane, which connects all simulators with one universal interface. This is not the case in single process simulations, since the entire system validation is performed in one process. Finally a heterogeneous approach is presented, which ensures cosimulation through single connections between every pair of simulators.

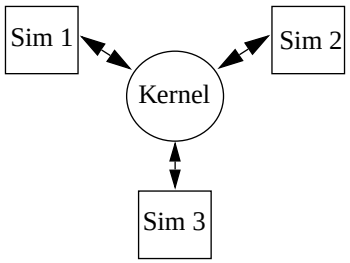


Figure 14: Backplane Based Simulation

3.2.1 Simulation Backplane

The characteristic of this structure, shown in figure 14, is that all involved simulators are connected to one kernel, which is responsible for the proper connections and translations between them. The tools which use this backplane either provide or support the different parts of a cosimulation environment, for example simulators, processor models and debugging tools. They connect the simulators, initialize them and control the entire cosimulation process. The control mechanism is responsible for correct communication, type conversions, synchronization, scheduling and often provides higher tasks, e.g. record/replay of simulations. In case of HW/SW cosimulation these kernels usually consist of a processor model, which provides the required translations, for example from software variables to signals translations. In addition they often support some of the speed-up mechanisms described in chapter 3.3 and enable designers to accommodate the control mechanism to their needs through parameter setting.

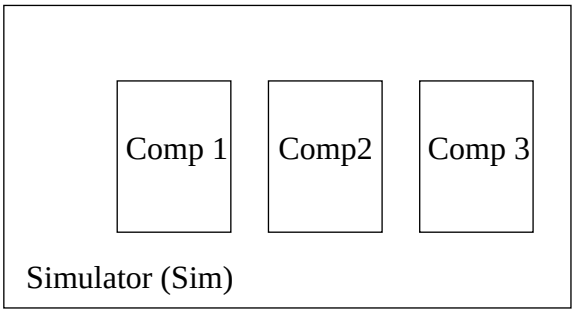


Figure 15: Single Process Simulation

3.2.2 Single Process Simulation

A lot of cosimulation approaches are based on already existing simulators. System design groups working on dedicated systems are often forced to build their own simulation environment, if existing tools are not usable for their specific applications. For this reason they use common simulators for models written in, for example, VHDL, Verilog or C, and extend them in a manner so that all system parts can be included in one simulator (see figure 15). As an example for such methods the **VHDL/Verilog-based simulation** should be considered in more detail.

In this technique the VHDL/Verilog simulator is used for both HW and SW simulation. Both VHDL and Verilog support a foreign language kernel or a programming language interface (PLI), respectively, to include C-code in the simulation. In Verilog for example, a preprocessor links procedures, written in C, to the Verilog code. In the Verilog code these procedures can be invoked by normal procedure calls. Therefore a communication mechanism including an inevitable overhead is not needed. The disadvantage of such simulation is the master/slave structure of procedure calls. With every call the procedure is started at the same point. Since SW is not always written in such a way, it must be altered for the simulation, which puts a fresh burden upon the designer.

This technique fits best systems with a small amount of functions implemented in SW. The designer can use the existing hardware simulator for simulating the entire system without requiring an interface between HW and SW and simulator.

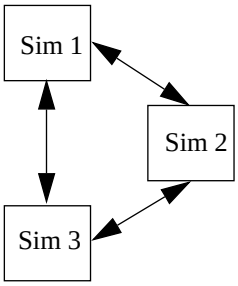


Figure 16: Heterogeneous Simulation Environment

3.2.3 Heterogeneous Simulation

In heterogeneous simulation the simulators are directly connected (figure 16) via the communication mechanism described in chapter 2.1. All communication control mechanisms, for example synchronization and blocking, have to be done explicitly and be supported by the communicating simulators since a central control mechanism is missing. Usually simulators do not have such mechanisms initially but they provide the linkage of user written functions to the simulator (see chapter 2.1.2). Heterogeneous cosimulation tools use this opportunity to link interfaces functions, which build up the required connection, to the simulators. A typical HW/SW heterogeneous cosimulation technique is shown in figure 6.

3.3 Speed-Up Mechanism

Several mechanisms to increase the simulation performance exist, but mostly at the expense of accuracy or requiring faster simulation environments. Some of them, for example higher processor model abstraction and optimistic scheduling, are already presented in the previous chapters. In the following a few more commonly used methods should be explained.

Communication between the components often appears as the bottleneck of the simulation. Therefore speed-up possibilities should be considered in more detail. The important factors in communication speed are the frequency and the size (due to the level of detail) of messages. To improve the latter some tools support communication on different levels of abstraction [3,23]. In [23] a so-called **multiple communication model** is presented. It enables the user to choose an appropriate communication level, e.g. stream transfer (information only) or detailed hardware simulating transfer (bus transactions) for every single connection. In this way repeated de-

tailed simulations of already known results (e.g. initialization phase) can be avoided and only required details are simulated (figure 17).

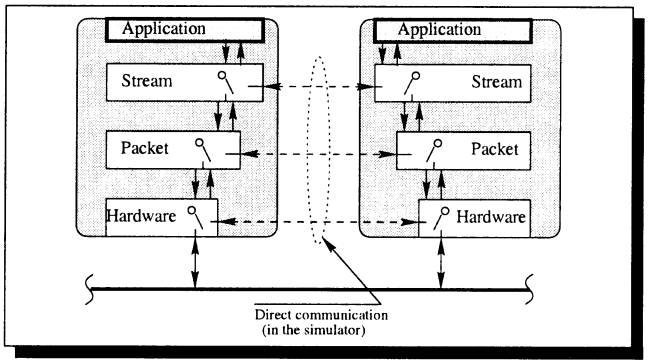


Figure 17: Multiple Communication Model [15]

To decrease the frequency of transactions simulation processes can be merged, for example multiple VHDL models in a system design can be simulated in one simulator and thus HW/HW simulator communications can be avoided. However, other factors, which will be regarded in connection with distributed simulations, can decrease simulation speed.

Using a **memory image server** is a special way to increase the abstraction level of communication and frequency of transactions between HW and SW. The cosimulation tool keeps two memory models, one model for the HW simulation and the other one is the memory image server supporting direct SW access. The designer can decide between timing accurate simulation of memory interactions by accessing the hardware model and fast simulation by accessing the memory image server. A disadvantage is that an updating mechanism is required to keep a consistent view of the memory. In chapter 4.2 an efficient implementation of an memory image server is presented.

Another mechanism which can increase the simulation performance drastically is **distributed simulation [6].** System designers have the choice between describing and simulating a system as a single process and dividing it into multiple communicating processes. Although the possibilities of separation are often limited by the partitioning of the system (for example HW/SW partitioning) separating and merging of system parts is often possible, for example, if the system contains several VHDL models. The multiple processes can run on separate machines and be connected through IPC channels, discussed in chapter 2.1. This leads, in particular for computational extensive simulations, to a high speed-up rate, however, a performance decreasing factor is the communication overhead. The following can be considered as a general law

for distributed simulation: The communication overhead of the slower process (mostly hardware simulation) has to be less than the computation time of the faster process (usually C-process) to increase the overall performance [6]. In [6] experimental results with a distributed C/Verilog simulation are presented. They increased the computational complexity of the C-code to determine the cross-over point from where a distributed simulation becomes worthwhile.

A special case of this method is the usage of **high powered coprocessors** [25], which are able to execute simulation of gate level models at high speed (about 4000 instructions per second). However, they are mostly too expensive for common co-design applications.

4. Tools

The previous chapters have described the underlying methods and techniques that are necessary for a cosimulation. Further the most important cosimulation tools, which are partly commercial products and partly research systems, are introduced. A general description of the cosimulation environment is given and in addition the employed cosimulation methods are presented.

4.1. Ptolemy

Ptolemy is a simulation framework, which enables designers to create their own cosimulation environments for the validation of heterogeneous systems. The resulting cosimulator consists of several processes, so called domains, which define different simulation behaviour. Ptolemy pre-determines the basic structure for domains and ensures the proper communications between them. Subsequently Pia [30] and an approach to cosimulation using Polis, which are both built on Ptolemy, are introduced.

Ptolemy [8,9,12] uses object-oriented software technologies in order to build the environment in an efficient manner. The Ptolemy kernel, written in C++, defines the basic classes, which are used to derive all the components of the environment. This method of developing a tool provides extensibility and the smooth co-operation of all components. Ptolemy allows the simulation of heterogeneous systems described in multiple design styles. Different domains support the different styles and computational models. They consist of a number of functional blocks and classes, which are derived from the basic classes.

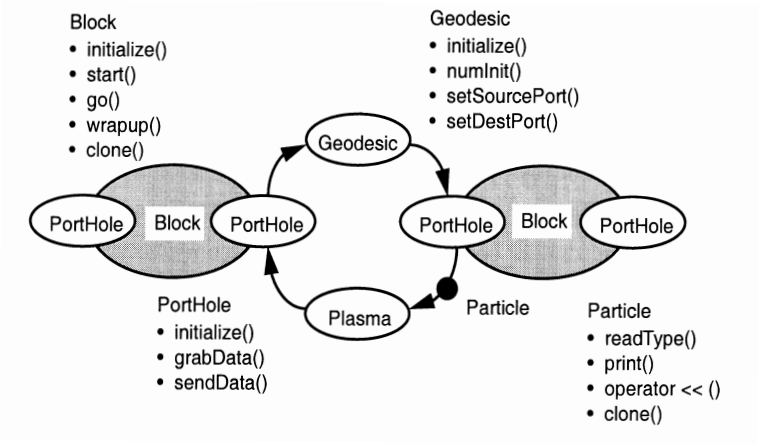


Figure 18: Basic Classes in Ptolemy [8]

The basic unit of Ptolemy, shown in figure 18, is a *block*, which includes a couple of functions. For example, the go() function activates a *block*, i.e. reading input messages, evaluation and writing output messages. *Portholes* enable the connections between blocks through send (sendData()) and receive (grabData()) functions. The messages, which are sent between *portholes*, have a standard format, since all of them are derived from the base type *particle*. Messages types may consist of arithmetical values, e.g. float or integer, complex data structures or control signals, such as tokens. Ptolemy ensures type conversions for the supported basic types, for example, the real to float type conversion. The geodesic and the plasma classes, which route the particle streams from the sending to the receiving block and handle transfer errors, control the communication. All classes and types used in Ptolemy are derived from these basic classes.

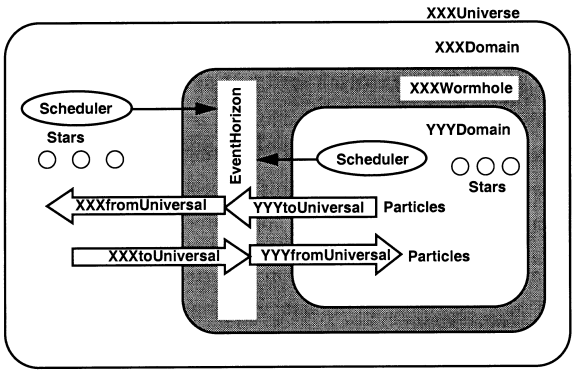


Figure 19: Domains and Wormholes [8]

The simplest type derived from a *block* is a *star*, which for example defines an arithmetical function between its input and output signals. *Galaxies*, which are also derived from *block*, are a collection of *stars* and perhaps other *galaxies*. An entire cosimulation environment in Ptolemy is called a *universe*. It contains *galaxies*, which describe the functionality of the system under test, and the class *runnable*, which determines the manner of cosimulation. *Runnable* contains a *target*, which defines the simulation process and controls the included *scheduler* during the entire simulation in order to achieve the desired result. The *scheduler* is responsible for the initialization and execution control of all blocks in the *universe*, for example, through invocation of the initialize() and go() method.

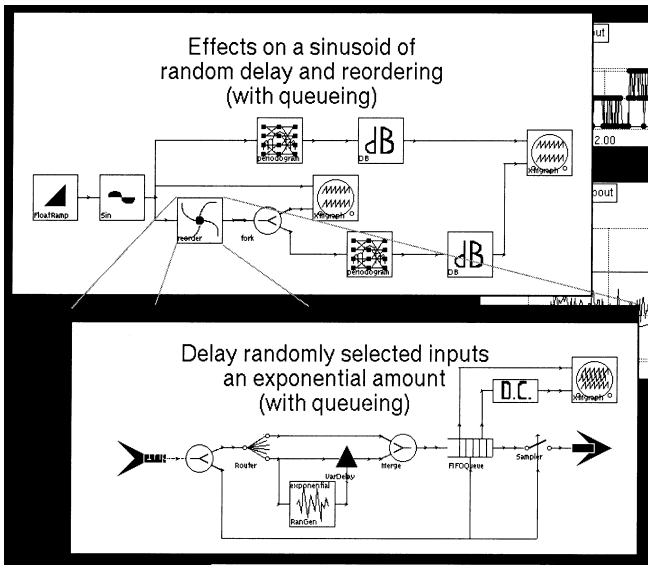


Figure 20: Signal Flow Graph in Ptolemy [8]

The entire design can be divided into *domains,* which consist of stars and targets, with the included scheduler. Inside of a *domain* all components are described by means of a given computational model, which are introduced in chapter 2.4. Table 2 gives an overview of the currently supported *domains* in Ptolemy. In addition to simulation domains code generation are provided. However, designers are also enabled to build their own *domains*. If the entire system is defined in one computational model, the *domain* corresponds a *universe,* otherwise *wormholes* are needed to connect the domains. As shown in figure 19 a domain can include other *domain* and the interface between them is a *wormhole*. For the outer domain the *wormhole* appears as a normal *star* but internally it contains a *domain*. An interface called *eventhorizon* performs the necessary translations between the different models of computation. The *particle,* which should pass between the *domains*, is converted to a universal type by objects of the classes *touniversal* and *fromuniversal*. This includes, for example, type conversions and adding or deleting of time stamps. The coordination of the multiple schedulers is another task, which is performed by the *eventhorizon*. To achieve a proper simulation process of the entire system the multiple schedulers have to be merged to a single scheduler.

To give an example, the coordination of two DE domains (see chapter 2.4.1) should be considered. Both schedulers keep in addition to their current time a stop time, which can be set (setStopTime()) from outside. If the wormhole is activated by the outer scheduler (similar to the activation of a normal star) it obtains the current time of the outer scheduler and assigns it to stop time of the inner scheduler. The inner domain is only active until its current time reaches the stop time. Hence, the inner domain can never get ahead of the outer *domain*. Finally the

inner *wormhole* sends a self-scheduling event to the outer scheduler, which contains the next time, when the inner *domain* should be activated.

Name	Expansion	Principal Use
SDF	synchronous dataflow	synchronous signal processing
DDF	dynamic dataflow	asynchronous signal processing
BDF	boolean dataflow	asynchronous signal processing
MDSDF	multidimensional dataflow	multidimensional signal processing
DE	discrete event	communication network modeling and determinate high-level system modeling
FSM	finite state machines	control
HOF	higher-order functions	graphical programming
Thor	(name given at Stanford)	RTL hardware simulation
MQ	message queue	telecommunications switching software
PN	process networks	real-time systems
CP	communicating processes	communication network modeling nondeterminate system modeling
CGC	code generation - C	software synthesis (SDF or BDF model)
CG56	code generation - DSP56000	firmware synthesis (SDF model)
CG96	code generation - DSP96000	firmware synthesis (SDF model)
Silage	a functional language	VLSI hardware synthesis (SDF model)
VHDLF	VHDL - functional	high-level modeling and design (SDF)
VHDLB	VHDL - behavioral	hardware modeling and design (DE)
Sproc	a multiprocessor DSP from Star Semiconductor	firmware synthesis (SDF)

Table2: Supported Domains in Ptolemy [9]

4.1.1 Cosimulation Approach utilizing Polis/Ptolemy

Polis [22] is a cosimulation environment, which is based on the codesign finite state machine (CFSM) and simulated in the DE *domain* of Ptolemy. Sangiovanni-Vincentelli et. al. [27] present an approach to HW/SW cosimulation using Polis. They specify the entire system as a group of communicating CFSMs, and map this to *stars* in the DE *domain*. In addition they use an ISS, which refines the timing calculation.

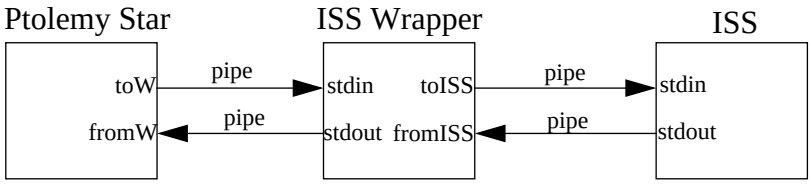


Figure 21: ISS Integration in Ptolemy [27]

The ISS is linked to Ptolemy through a so-called ISS wrapper and a special *star* (see figure 21). For each kind of ISS a specific ISS wrapper exist, which performs the necessary interface unification and command translation. Thus, the same *star* fits for the connection from all kinds of ISSs to Ptolemy. To speed-up the simulation they use a so-called cached refinement scheme, that is a store and reuse method of previously calculated timing information. Thus unnecessary re-evaluation of already existing timing results can be avoided. Described in more detail, the method assigns keys to particular parts of the execution. If these parts are executed by the ISS the returned delay time is stored together with the corresponding key in a table. The next time, when the same part should be executed, the table obtains the previously estimated delay time instead of a re-activation of the ISS.

4.1.2. Pia

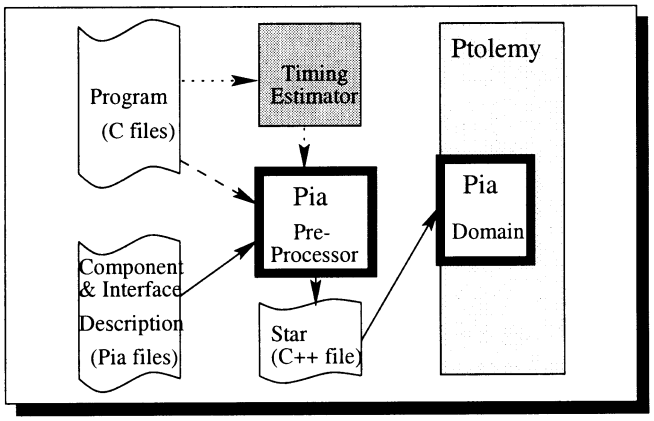


Figure 22: The Cosimulation Tool Pia [22]

Pia [22] is a cosimulation tool developed at the University of Washington, Seattle as a Ptolemy domain. Figure 22 shows the Pia tool structure, which includes the Pia preprocessor and the generated Pia *domain* in Ptolemy. The tool uses a specific language, the Pia language, to describe the functionality of a simulated system. In Pia, a system consists of objects of the types component, interface, port and wire, see figure 23. Components describe the behaviour of a physically component and contain interfaces. Interfaces are a collection of ports and drivers. Ports are senders and receivers of events. Wires provide the connections between ports. These objects describe only the hardware of a system. However, software, which for example should run a processor, can be included in the processor's component. The Pia language support file operations, which allow to link software code to the components.

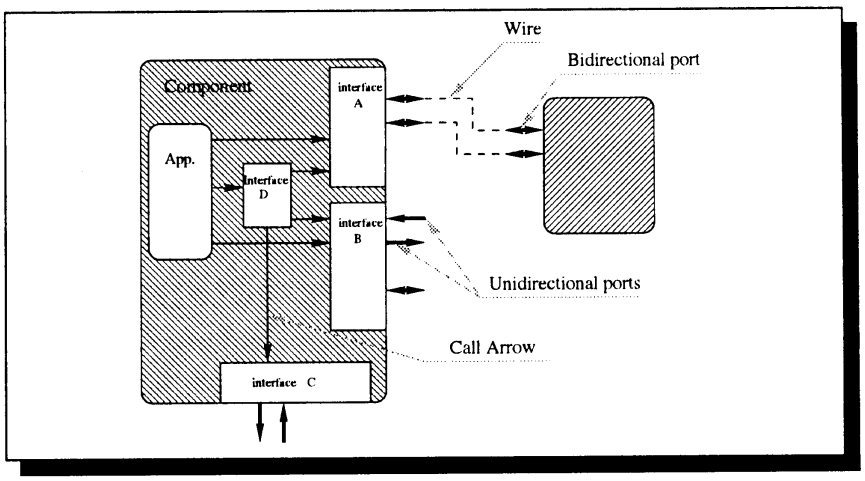


Figure 23: Pia Objects [22]

The entire system specification, i.e. hardware defined in Pia language and C code describing the software is translated by a compiler to Ptolemy *stars*. The stars are imported in the Pia domain. The Pia domain is similar to the standard DE domain in Ptolemy (see 2.4.1), except that it contains some additional features, for example, checkpoints and restore functions are introduced to enable optimistic scheduling, see chapter 2.3.

In the following the most important features of the Pia tool are presented:

1. **Multiple communication models**: (see also chapter 3.3)
   The SW usually communicates with the HW through drivers. These drivers are often hierarchically layered, with an increasing amount of additional information in the hardware layers, which induces a decreasing simulation speed. In Pia the designer is able to define multiple communication between all layers of the driver model, so that the higher level layers can communicate directly with each other, if a detailed simulation of the driver is not necessary.

2. **Advanced scheduling**:
   In Pia, a single scheduler controls the timing of the entire simulation. The scheduler works with the global time/local time concept, presented in chapter 2.3. In addition the simulator distinguishes between active and reactive components. Reactive components only run after they got a stimuli, whereas active components inform the scheduler when the have to be activated again, thus they determine the next run time themselves. The local time of active

components always have to be in track or ahead the system time to ensure that their states are valid, due to rule 3 in chapter 2.3. However, the local time of reactive components only has to be updated if they got a stimuli, thus their local time can also be less then global time, which is incompatible to rule 1. To ensure that rule 1 is still valid, the system increments must be reflected for the delayed component, so that it can catch up without missing information about changes in the system.

Both, conservative and optimistic scheduling are supported.

3. **Multiple processor models**:

   Processors and processor blocks can be described at different level of abstraction, see chapter 3.1. This allows a speedup if detailed information about the processor simulation is not required.

   It is also possible to compile the source code for the host machine (HCE) and run it much faster than on a model of the real processor.

4.2 Seamless CVE

Seamless [13,14] is a tool that establishes connections between hardware and software simulators based on a backplane via open APIs. This backplane is tool-independent so that it is applicable to a wide range of hardware and software simulators.

Seamless is a cosimulation environment that supports a lot of simulation optimization parameters, which enable the user to find an appropriate trade-off between speed and the level of detail of the simulation.

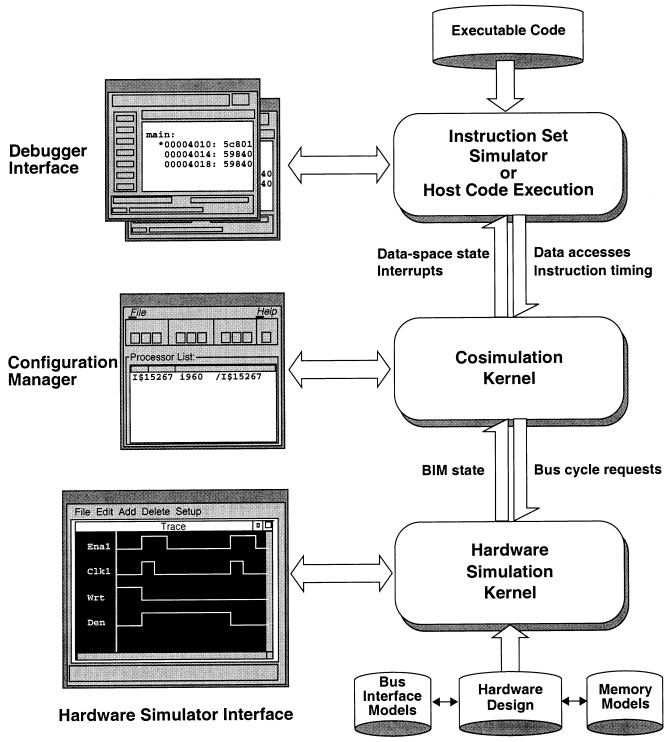


Figure 24: Seamless CVE Tool Environment [13]

The whole CVE architecture consists of the following three parts (figure 24):

The software kernel, which can be realized as an instruction set simulator (ISS) or host code execution (HCE) (chapter 3.1.2), the hardware simulation kernel, different hardware simulators are possible, and the connecting interface between both, the cosimulation kernel. The user is enabled to control and view all parts via graphical interfaces.

If available, an ISS for the specific processor can be applied for software simulation. This is much quicker, than simulating the whole processor in hardware. The ISS consists of instruction handling, interrupt control, registers and the address space (see figure 11). To enable a cosimulation, the CVE has to be introduced to the ISS. Thus, the cosimulation kernel can control the communication between the HW simulator and the SW simulator (ISS or HCE). The ISS sends data-access requests and timing instructions to the hardware simulator, and receives data-space states and exception requests from it, such as interrupts and resets. It is also possible to apply HCE, which has the advantage to be much faster then the ISS. For such a cosimulation the designer has to place calls to the API functions in the source code, in order to enable interconnection between the software and the cosimulation kernel.

The hardware simulator contains the according bus interface model (BIM) of the target processor and additional hardware components. The communication between the hardware and

the software part includes read/write operations to and from the memory and the I/O ports. The hardware simulator simulates the executions and calculates the required time. Usually the read/ write requests are passed from the ISS (or HCE) to the BIM, which creates the port signal and reports the amount of bus cycles used including possibly occurring wait states. In addition to read/write operations, CVE supports exception and interrupt handling.
To speed-up the simulation CVE supports the following three optimization methods which avoid such hardware accesses for user defined operations:

1. **Data access optimization:**
   Accesses to the user specified address ranges are only simulated in software
2. **Instruction fetch optimization:**
   This mode avoids hardware simulations for all instruction fetches
3. **Time optimization:**
   Hardware and software simulators are unsynchronized.

CVE Memory Model

To accomplish these optimizations a special memory model is necessary, in order to allow the software to access the memory without activating the hardware simulator. Therefore CVE provides a memory image servers, which is introduced in chapter 3.3. That is, in addition to the model of the memory that is part of the hardware simulator, a model for direct software accesses, the memory image server is provided.

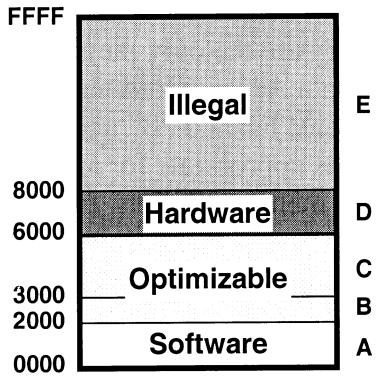


Figure 25: Memory image server [13]

Figure 25 shows how the address range of the memory image server is divided in four

different parts. The illegal region is not accessible from the software simulator at all. Accesses to hardware addresses are always passed to the BIM and cause a detailed hardware simulation. The range of software addresses is only accessible from software and they generate no bus-cycle simulation. The optimization region defines the address range, in which the previously presented optimization method can be utilized.

A so-called memory instance mapping defines, which of these address ranges in the memory image server cover which instance in the hardware design.

4.3 Eagle Design Automation: EagleI and EagleV

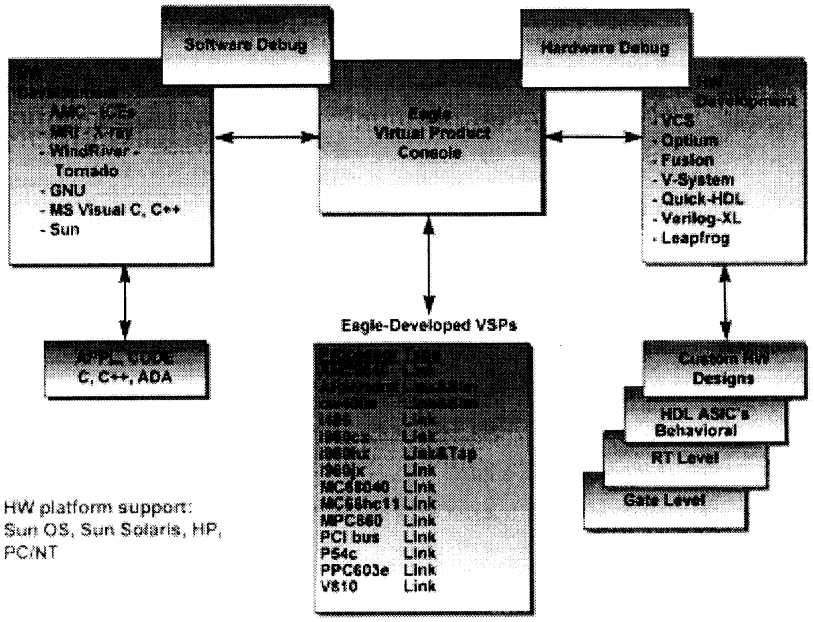


Figure 26: EagleI/EagleV Tool Structure [19]

EagleI and EagleV are two HW/SW cosimulation tools offered by Eagle Design Automation, which apply a cosimulation method similar to Seamless. Therefore, only the tool structure and differences to Seamless are considered.

Both, EagleI and EagleV consist of two main components, the Virtual Software Processor (VSP) and the Virtual Product Console (VPC), see figure 26.

Virtual Software Processor (VSP)

For each provided processor type a specific processor model (VSP) exist. However, a generic

model supporting essential operations will be supported in the future. Three different types of Processor models exist, according to the degree of detail and performance (see chapter 3.1):

1. **VSP/Link:**
   contains a bus functional model (BFM) and applies host code execution (HCE)
2. **VSP/Sim:**
   is based on an instruction set simulator (ISS)
3. **VSP/Tap:**
   includes an in-circuit emulator (ICE)

The BFM is written in C code containing a set of pins and the bus cycle behaviour identical to the real processor. This C code is connected to VHDL or Verilog through the foreign language interface or the Verilog PLI, respectively (see chapter 2.1.2).

In contrast to Seamless the VSPs do not support configurable optimization methods. They gain a high performance by invoking the HW simulator only for read and write operation with a memory location in the HW design, such as I/O ports, and to service a hardware interrupt. However, interrupts are maskable and priorities might be defined.

Virtual Product Console (VPC)

The VPC works as control and connection unit in the simulator. It connects the HW simulator and the SW simulator with the VPS and controls the simulation process in a similar manner than the cosimulation kernel in Seamless. VPC provides debug windows, which support the designer during the simulation. In addition, it allows the simulation of embedded systems containing multiple processors, which have local synchronization methods.

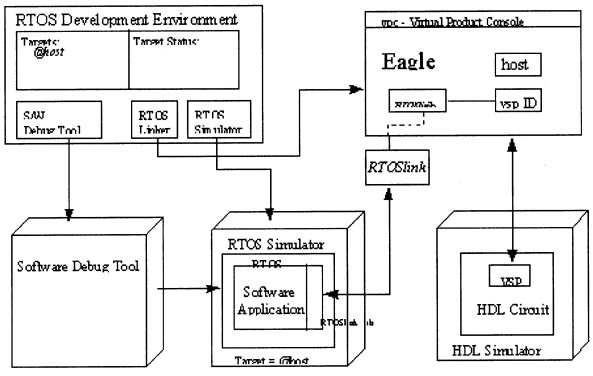


Figure 27: RTOS environment linked to EagleI [19]

EagleI and EagleV permit a distributed simulation (see chapter 3.3) and support the connection with a real-time-operating system (RTOS) environment instead of a software simulator, see figure 27.

4.4.Cosimulation in a VHDL-based Test Bench Approach [7]

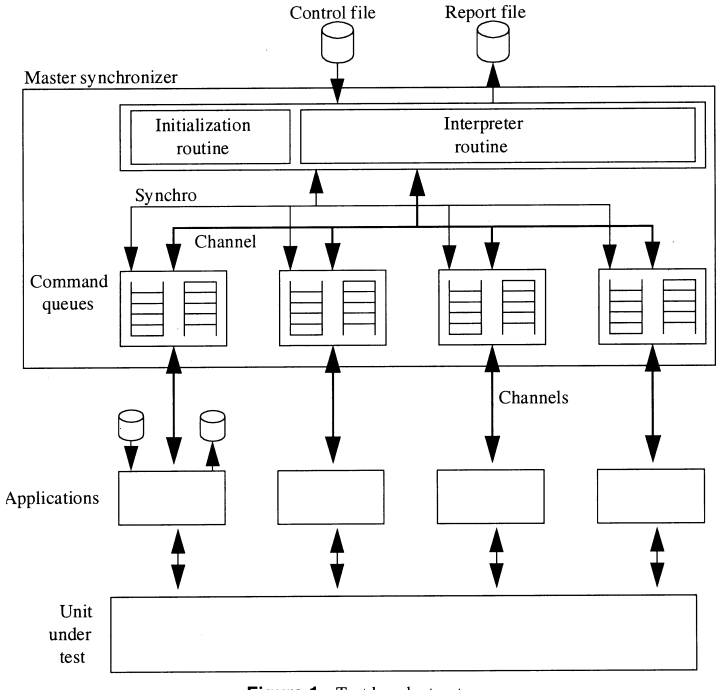


Figure 28: Cosimulation Utilizing a Master Synchronizer [7]

This approach is based on a VHDL simulation environment. The simulation is controlled and synchronized by the master synchronizer, written as a test bench in VHDL code (figure 28). Models of hardware components, so called applications, establish the connection to the unit under test. A microprocessor is an example for an application.

The microprocessor model consists of a bus functional model of the CPU and the message handler that establishes the communication between test bench, bus functional model and software. In addition some macros, such as interrupt routines, may exist as VHDL procedures.

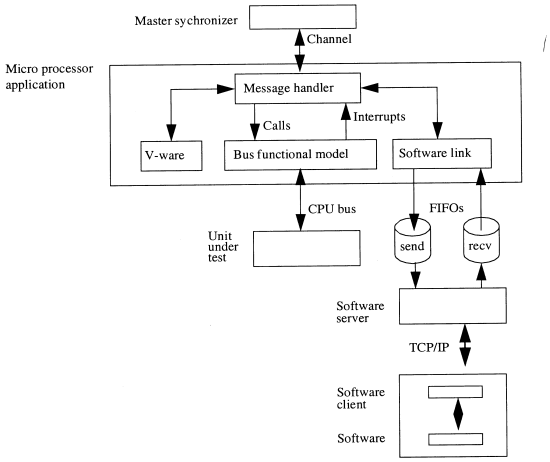


Figure 29: Processor Model [7]

The software is written in C, compiled for and executed on a workstation. As shown in figure 29, the software is connected to the VHDL simulator via TCP/IP and pipes (see also chapter 2.1.1). Through the two pipes (FIFOs), one for each direction, data is transferred as ASCII text, which has the advantage that the VHDL-C interface is tool independent. For transferring data from C to VHDL a file is opened during the entire simulation. For the other direction a file is opened and closed for each operation in order to flush the buffer every time. Both, the VHDL simulator and the software execution process are connected via UNIX-sockets to an IPC channel.

A control file that contains high-level commands for testing drives the simulation. The simulation results and error messages are written to a report file. Both files are written in ASCII code.

4.5 Virtual CPU

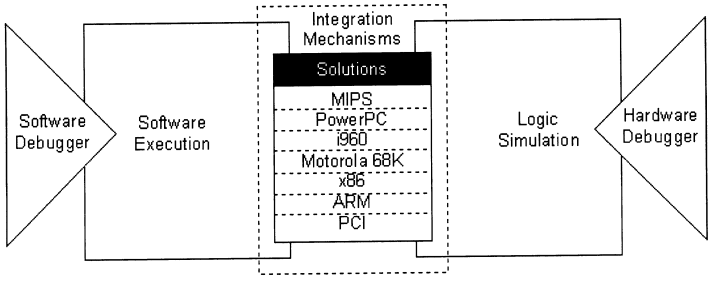


Figure 30: Virtual CPU [31]

Summit Design has developed a cosimulation tool called Virtual CPU, which is similar to the previously presented tools Seamless and EagleI. The structure, given in figure 30, shows the three main components of the simulation environment. For the software execution the designer can choose between an instruction set simulator (ISS) and host code execution (HCE). An HDL simulator performs the hardware; currently the tool supports the Cadence Verilog simulator.

The connecting component between the hardware and the software is a, so-called, M.Core, which contains a bus functional model (BFM). Virtual CPU offers BFMs for MIPS, PowerPC, Intel i960, Motorola 68k, x86, ARM and PCI bus.

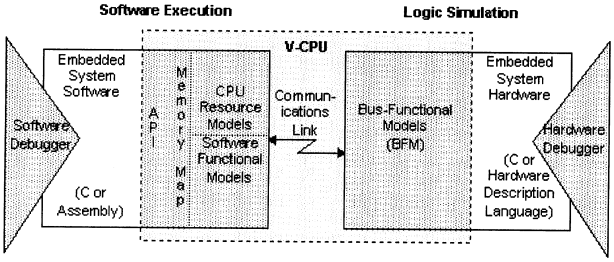


Figure 31: Internal Structure of Virtual CPU [31]

However, in comparison with the other tools, it provides the following additional features:

1. CPU resource model
2. Software functional model
3. Implicit hardware access
4. Record/Playback functions

Figure 31, which presents the tool structure in more detail, shows the so-called CPU resource models and the software functional models. The first enables the simulation of configurable cache models, with parameter setting such as cache size, cache line size and write mode. The software functional models are similar to operating systems, described in 3.1.2. They contain a functional description of the hardware written in C code, which is executable in the software simulation. Thus the hardware simulator is not requested if applying this method, which ensures a fast cosimulation.

The implicit hardware access is a convenient feature for HCE simulations. This mechanism

adds the necessary API calls, which establish the connection to the cosimulator, automatically to the source code for the HCE. In Seamless, see chapter 4.2, the designer has to do this manually. In addition, the record and playback functions of bus cycles ease the debugging process.

4.6 CoSim

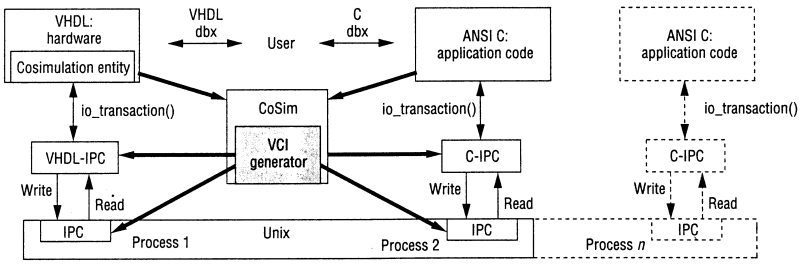


Figure 32: The CoSim Connection Scheme [28]

CoSim has been developed at the Tima Lab at Institute National Polytechnique, Grenoble in cooperation with SGS-Thomson. It is part of the COSMOS codesign tool, which allows a system specification in SDL and provides an automatic translation to the corresponding VDHL and C code. CoSim, see figure 32, serves the required interface between C and VHDL and performs the cosimulation.

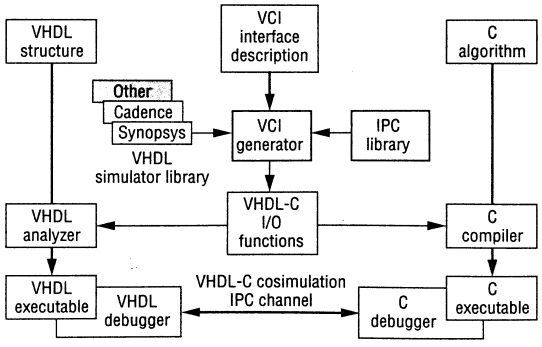


Figure 33: VCI Generator [28]

The VHDL-C-interface (VCI) generator (figure 33), which applies the foreign language kernel in VHDL and IPC channels (chapter 2.1.2), establishes the connection between the C code and the VHDL simulator. It generates automatically VHDL entities and C procedures,

which carry the hardware ports and software variables that should be connected. These additional codes are linked to the original code in order to enable the interconnection.

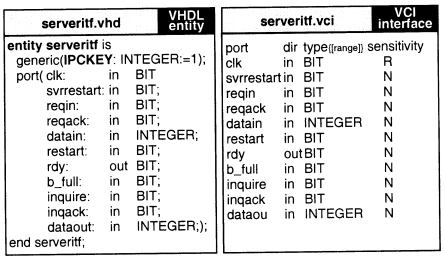


Figure 34: VCI Input File and Equivalent VHDL Entity [28]

The system designer has to define these hardware/software connections in an interface description input file together with a corresponding sensitivity types [28] (figure 34). This type determines the occurrence of communication for a given variable/port pair. Three sensitivity types exist, rise and fall, which initiate a transfer at a rising or falling flank of the signal, respectively, and an additional type, which initiates a transfer at every event. Thus, the designer can specify different synchronization methods, e.g. rendezvous mechanism, unsynchronized transfer (see chapter 2.2).

5. Conclusion

This paper presents a survey of the simultaneous simulation of hardware and software components in embedded systems. The fundamental mechanisms and methods, ensuring a proper cosimulation of heterogeneous systems, are summarized. The selection of tools presented in this paper should give an overview of existing commercial tools and academic approaches and introduce potential concepts to achieve the objective of cosimulation.

Cosimulation, which is an essential part in the process of codesign and of crucial importance for the validation of heterogeneous systems, is still a challenge for tool designers. Although many codesign environments perform the task of cosimulation, most of them are restricted to special system conditions. For example, many tools only support particular systems structures and a few numbers of processor models. Others cover only special applications, such as data processing or telecommunication, or set constraints in the choice of applicable simulators. A general and flexible, yet efficient, approach to cosimulation is still missing.

Cosimulation enables an early system validation, which facilitates the development of HW/SW systems and decreases the design time. In addition, hardware and software designers are empowered to co-operate at an early design stage. Therefore, system designers, who are involved in such fields, should take advantage of this opportunity. In order to find the appropriate cosimulation environment, they have to study the system under development meticulously. As a result, they might choose one of the existing tools or consider the opportunity of developing their own simulation environment, which adapts to their demands.

6. List of Acronyms

ASCII	text format
API	application program interface
ASIC	application-specific integrated circuit
ASS code	Assembly code
BIM/BFM	bus interface/functional model
CPU	central processing unit
DE	discrete event
FIFO	first-in-first-out (buffer)
FSM	finite state machine
FPGA	field-programmable gate array
HCE	host code execution
HW	hardware
I/O primitives	read/write instructions in communication processes
ICE	in-circuit emulator
IPC	interprocess communication (under Unix)
ISS	instruction-set simulator
PLI	programming language interface (in Verilog)
RTOS/OS	(real time) operating system
RTL	register transfer level
SDF	synchronous dataflow
SDL	specification and description language
SW	software
TCP/IP	protocol suite for Internet communication
VEC	VHDL emulation C-procedures (also called foreign language kernel)
VHDL/Verilog	hardware description languages

9. References

[1] Wayne H. Wolf, ”Hardware-Software Co-Design of Embedded Systems”, *Proceedings of the IEEE*, Vol. 82, No. 7, pp. 965-989, July 1994.

[2] Richard W. Earnshaw, Lee D. Smith, Kevin Welton, ”Challenges in Cross-Development”, *IEEE Micro*, Vol. 17, No. 4, pp. 28-36, July/August 1997.

[3] Jay. K. Adams, D. E. Thomas, ”The design of Mixed Hardware/Software Systems”, *Proceedings of the 33rd Design Automation Conference*, Las Vegas, pp. 515-520, June 1996.

[4] Benny Schnaider, Einat Yogev, ”Software Development in a Hardware Simulation Environment”, *Proceedings of the 33rd Design Automation Conference*, Las Vegas, pp. 684-689, June 1996.

[5] J.-P. Soininen, T. Huttunen, K. Tiensyrjae, H. Heusala, “Cosimulation of Real-Time Control Systems”, *Proceedings of the EuroDAC 95*, pp. 170-175, 1995.

[6] Sari L. Coumeri, Donald E. Thomas, ”A Simulation Environment for Hardware-Software Codesign”, Proceedings, International Conference on Computer Design, pp. 58-63, October 1995.

[7] Matthias Bauer and Wolfgang Ecker, ”Hardware/Software Co-Simulation in a VHDL-based Test Bench Approach”, *Proceedings of the 34th Design Automation Conference*, Anaheim, June 1997.

[8] J. Buck, S. Ha, E. A. Lee, D.G. Messerschmitt, ”Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems”, *International Journal of Computer Simulation*, January 1990.

[9] Brian L. Evans, Alan Kamas, Edward A. Lee, ”Design and Simulation of Heterogeneous Systems using Ptolemy”, *Proceedings of the 1st Annual Conference of the Rapid Prototyping of Application Specific Signal Processors (RASSP) Program*, 1994.

[10] W.-T. Chang, Asawaree Kalavade, Edward A. Lee, ”Effective Heterogeneous Design and Co-Simulation”, G. De Micheli, M. Sami (eds.), *Hardware/Software Co-Design*, pp. 187-212, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1996.

[11] Clifford Liem, Francois Nacabal, Carlos Valderrama, Pierre Paulin, Ahmed Jerraya, ”System-on-a-Chip Cosimulation and Compilation”, *IEEE Design and Test of Computers*, pp. 16-25, April-June 1997.

[12] Asawaree Kalavade, *System-Level Codesign Of Mixed Hardware-Software Systems*, Ph.D. Dissertation, Dept. of EECS, University of California, Berkeley, CA 94720, Sept. 1995.

[13] *Seamless Co-Verification Environment User's and Reference Manual*, V 2.2, Mentor Graphics Corporation, Wilsonville, Oregon, 1996-98.

[14] Russel Klein, Serge Leef, "New Technology Links Hardware and Software Simulators", *Electronic Engineering Times*, June 1996.

[15] Ken Hines, Gaetano Borriello, "Dynamic Communication Models in Embedded System Co-Simulation", *Proceedings of the 34th Design Automation Conference*, Anaheim, June 1997.

[16] J. Rowson, "Hardware/Software Co-Simulation", *Proceedings of the 31. Design Automation Conference*, pp. 439-440, 1994.

[17] Carlos Valderrama, *VCI The VHDL-C Interface Generation Tool*, http://tima-cmp.imag.fr/Homepages/valderr/Vci/index.html.

[18] David Becker, Raj K. Singh, Stephen G. Tell, "An Engineering Environment for Hardware/Software Co-Simulation", *Proceedings of the 29th Design Automation Conference*, Anaheim, June 1992.

[19] Viewlogic and Eagle Design Automation, http://www.viewlogic.com/products/eagletools.html .

[20] Douglas E. Comer, *Internetworking With TCP/IP Volume I: Principles, Protocols, and Architectures*, Prentice-Hall, London, UK, 1991.

[21] Douglas E. Comer, *Internetworking With TCP/IP Volume III: Client-Server Programming And Applications*, Prentice-Hall, London, UK, 1993.

[22] C. Passerone, L. Lavagno, C. Sansoe, M. Chiodo, A. Sangiovanni-Vincentelli, "Trade-off Evaluation in Embedded System Design via Co-simulation", *Proceedings of the ASP-DAC'97*, Chiba, Japan, January 1997.

[23] Ken Hines, Gaetano Borriello, "Optimizing Communication in Embedded System Co-simulation", *Fifth International Workshop on Hardware/Software Codesign*, Braunschweig, Germany, pp. 121-125, 1997.

[24] S. Edwards, L. Lavagno, E. A. Lee, A. Sangiovanni-Vincentelli, "Design of Embedded Systems: Formal Models, Validation and Synthesis", *Proceedings of the IEEE*, Vol.85, No. 3, March 1997.

[25] Phil Dreike, James McCoy, "Co-Simulating Software and Hardware in Embedded Systems", *Embedded System Programming*, Vol. 10, No. 6, June 1997.

[26] Sungjoo Yoo, Kiyoung Choi, "Optimistic Distributed Timed Cosimulation Based on Thread Simulation Model", *Sixth International Workshop on Hardware/Software Codesign*, Seattle, Washington, pp. 71-75, March 1998.

[27] Jie Liu, Marcello Lajolo, A. Sangiovanni-Vincentelli, "Software Timing Analysis Using

HW/SW Cosimulation and Instruction Set Simulator", *Sixth International Workshop on Hardware/Software Codesign*, Seattle, Washington, pp. 65-69, March 1998.

[28] C. A. Valderrama, Francois Nacabal, Pierre Paulin, A. A. Jerraya, "Automatic Generation of Interfaces for Distributed C-VHDL Cosimulation of Embedded Systems: An Industrial Experience", *7th International Workshop on Rapid Systems Prototyping*, Greece, June 1996.

[29] Steven Vercauteren, Bill Lin, "Hardware/Software Communication and System Integration for Embedded Architectures", Design Automation for Embedded Systems, Vol. 2, No. 3/4, p. 363, May 1997.

[30] Ken Hines, "Pia: A Framework for Embedded System Co-Simulation with Dynamic Communication Support", *Technical Report UW-CSE-96-11-04*, Department of Computer Science & Engineering, University of Washington, Seattle, January 1997.

[31] "Virtual CPU Coverification Environment", Summit Design Inc., http://www.summit-design.com/products/verification/vcpu.html, June 1998.