Chapter 1

A Methodology for Bridging the Gap between UML and Codesign

Ananda Shankar Basu[1], Marcello Lajolo[1], Mauro Prevostini[2]

[1] *NEC Laboratories America*
Princeton NJ, USA

[2] *ALaRI, University of Lugano*
Lugano, Switzerland

Abstract The Unified Modeling Language (UML) is getting more popular among system designers due to the need to raise the level of abstraction in system specifications. We present here a methodology that integrates UML specifications with a hardware/software codesign platform. This work aims to give a contribution toward SoC Design Automation starting from system level specification down to hardware/software partitioning and integration.

1. Introduction

With the increasing design complexity and the reduction of the time to market windows, the design of electronic systems has become a challenging task to be handled by traditional methodologies. Embedded systems design in comparison to traditional software development requires not only to verify the functional correctness, but also to satisfy tight performance and cost constraints. Hence, new methodologies are needed to improve design productivity and derive high performance low cost implementations keeping in mind the reuse of pre designed components.

The software community, after several years of work, converged on a set of notations for developing specifications of object oriented systems known as the Unified Modeling Language or UML [17] that has been very successful as a visual way for describing software. However, UML is not limited to software modeling and the development of UML 2.0 has been undertaken with the express

intention of producing a language that has benefits for a much wider audience than just software developers, including the world of systems engineering.

In this work, we present an integration of a UML based modeling methodology with a C based design technology called ACES (**A**pplication to **C** to **E**xploration to **S**ystem LSI) [8] that leverages on high level synthesis and coverification tools and aims to assist the designer in the hardware/software partitioning and architecture selection phases. ACES has the unique advantage with respect to all similar approaches to be able to leverage off the strengths of two key pieces in NEC's C based design flow [23]: CYBER, a behavioral hardware synthesis tool and CLASSMATE, a hardware/software coverification tool. UML complements ACES with an object oriented modeling language with both graphical and textual notations, organized in a set of diagrams, each diagram capturing a different aspect, or level of abstraction, of the system. The result is a unified design flow from system specification down to system implementation.

This chapter is organized as follows. Section 1.2 talks about the state of the art in electronic system level (ESL) design and focuses on our main contributions. Section 1.3 describes the ACES codesign flow, which is an integral part of our methodology. Section 1.4 talks about the hardware oriented modeling aspects of UML. Section 1.5 describes how the model can be verified in the UML environment. Section 1.6 talks about the link between UML specifications and the codesign environment. Section 1.7 presents our conclusions.

2. State of the Art and Contribution

As the complexity of systems increases, so does the importance of good specification and modeling techniques. Many factors contribute to the success of a project, and certainly one we cannot do without is a rigorous modeling language standard (see, e.g., [7, 13]). Introduced in recent years,UML [17] is now widely used, historically for requirements specification and for the design of complex software systems and since at least a couple of years also for hardware modeling and for embedded systems design. Although UML has a lot of advantages, is still not fully reliable for hardware description, especially for event semantics [3, 9]. This lack of semantics for hardware modeling exists because UML was originally thought by the software development community. The Object Management Group (OMG) [14] is at the moment assessing it in order to define standard semantics able to improve hardware description modeling. These new standards have been recently adopted by OMG through UML 2.0 [22].

On the other hand, Electronic System Level (ESL) design has been a hot area for Electronic Design Automation (EDA) vendors and startups in particular, but there are so many entries now that marketplace confusion is more likely than widespread adoption. ESL point tools are many, but flows that can go

from concept to implementation are few. For example, commercial hardware and software coverification tools from companies such as Mentor Graphics, CoWare, VAST, Virtio and Axys can provide fast instruction set simulators linked to various hardware simulators. They mainly focus on the functional and performance modeling problem for software dominated embedded systems, although they do not address the issues of high level hardware modeling and refinement. The main limitation of these tools is that they often require to model the hardware at the RT level and even though recently some of these vendors have started to offer the possibility to perform a mix C/RTL coverification (e.g. C Bridge from Mentor Graphics), none of them offers yet an automated behavioral synthesis path from behavioral specifications.

An emerging area is also the one of coprocessor synthesis [11, 19, 4], where the main idea is to combine the software compilation and the hardware synthesis technologies to provide a system that allows designers to explore and implement their designs directly from descriptions written in algorithmic C. The main limitation of this approach is that it is based on the assumption that the designer has already been able to come up with a feasible hardware/software partitioning for the entire design and the coprocessor synthesizer can then provide the possibility to perform some software acceleration by offloading compute intensive algorithms from the CPU to dedicated hardware. Although very useful, tools of this type can only provide a partial support to a complete SoC design flow because it is well known that many decisions regarding the efficiency (performance, power, area etc.) of the system have largely been fixed by the time a designer commits to a particular architecture.

Alternative and complementary methodologies and solutions must hence be provided in order to help the designer during the initial phases of the design process when coarse hardware/software partitioning tradeoffs have to be analyzed. Our work is an attempt to try to fill this gap by proposing a practical integration between a UML based modeling methodology and an existing hardware/software codesign technology.

3. The ACES Codesign Flow

The overall flow presented in this chapter is shown in Figure 1.1. Our proposed methodology starts with the UML specification of the system, followed by exploration of the UML database for extraction of functional and structural information. This is followed by an interactive process performed through a web based interface that allows to capture UML specifications and design constraints provided by the designer, like architectural specifications and hardware/software partitioning, and export the entire structure of the design into the ACES [8] codesign environment.

In ACES the system is described at the behavioral level as a network of components that can communicate by both means of events as well as shared variables. A web based interface acts as an intermediate layer between UML and codesign through which the user can drive the codesign process by performing the important tasks of component and communication mapping. A library of precharacterized architectural templates is provided in order to allow the designer to explore different design solutions.

The following sections describe in detail the various phases in this design flow.

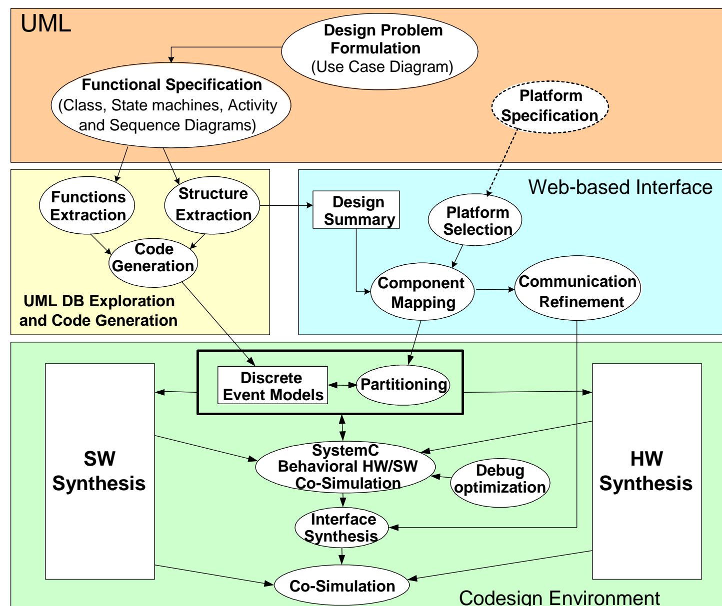

Figure 1.1. Design Flow

3.1 UML Specifications

UML is an object oriented modeling language that consists of graphical and textual notations, organized in a set of diagrams, each diagram capturing a different aspect or level of abstraction of the system [17]. After getting the requirements specification of the system to be designed, the first step is to capture the functionality of the system as a whole using Use Case Diagrams. In the second step, the functionality is decomposed into components within Class ,describing the SoC's structure, State Machines, Activity and Sequence Diagrams, describing the SoC's behavior. Constraints (i.e. performance) are captured using Stereotypes, which are simple extension mechanism of UML,

and propagated and budgeted to the components. In the following step, the model is simulated in the UML environment in order to check whether the functional behavior of the system matches the original specifications.

For a first analysis of a possible integration between UML and codesign, we have started by considering a UML specification flow in which first an Object Model Diagram (OMD) is defined to capture the structural decomposition into interacting components. An OMD contains two sets of classes: the ones whose behavior could potentially be implemented either in hardware and in software and others that do not have to enter in the codesign flow, for example, test-benches and strictly software oriented components. The first set of classes are distinguished by a specific set of UML stereotypes and additionally they are also used to differentiate between the types of behavioral specification associated with a particular class. For example, the stereotype *Partitionable_StateMachine* is used for classes with statemachine behavior and *Partitionable_Text* is used for classes with textual specifications. We will talk in more details about this item in Section 1.4 Communication among objects of classes can be specified through links connecting ports of different objects. Ports are stereotyped as *output* or *input* which allows us for semantic verification of connection between ports during the structural information exportation process Visually, interfaces among classes, are described by means of ports and connectors (see [18]). The behavior of the classes participating in the codesign process can be specified graphically using state machines as well as textually in the form of behavioral SystemC code, attached as a description to a class.

As a next step, the UML Functional Specification must be translated into ACES Discrete Event Models to conjugate the convenience of using the graphical UML Platform interface for specification with the possibility to use the analysis and synthesis tools available using the ACES codesign methodology. We have also proposed a possible way to specify the architectural platform in which system modules would be deployed onto different architectural components using the UML Deploymemt diagrams.

3.2 System Level API

We provide a specific API, basically an extended UML library, in order to allow the user to describe the type of communication that he wants to be performed. The API combines transaction level modeling for the hardware interface and OS and device driver levels for the software interface into a unified semantics. The objective is to provide designers with a minimal set of high level primitives that can be used to abstract and specify the behavior of the system. The proposed generic API for design specification is presented in the following Table. It is based on the POSIX [21] standard, a well defined and accepted programming interface for Operating Systems. The API is divided in four parts:

Process Management, Communication, Synchronization and Timing. Process management includes functions to control process creation and execution. The Communication part encompasses shared memory and message passing based communication, both blocking and nonblocking style. Synchronization includes primitives for process synchronization, like mutexes, semaphores and condition variables. Finally, the Timing section allows some control over the timing behavior of the system, providing a timed wait and controlling timeouts for blocking operations.

Process Management	process_create(id, param, func, arg) process_delete(id) process_suspend(id) process_resume(id)
Communication	port_send(port, data, mode) port_receive(port, mode) shared_mem_read(mem, offset, mode) shared_mem_write(mem, offset, data, mode)
Synchronization	mutex_lock(mutex) mutex_unlock(mutex) sema_wait(sem) sema_post(sem) cond_var_wait(var, mutex) cond_var_signal(var) cond_var_broadcast(var)
Timing	time_wait(time) process_join(id) mutex_lock_tmo(mutex, time) sema_wait_tmo(sem, time) cond_var_wait_tmo(var, mutex, time)

Table: The API functions

The API is thought to be integrable with any system level specification language like, for instance, SystemC. The range of specification styles possible to target with the API is very broad. Hardware oriented specifications might use bit manipulation and low level constructs more intensively, while software oriented specifications could use pointers, memory allocation and stack manipulation more frequently. Nevertheless, the API we propose is neutral and can accommodate either style.

In the Process Management section of the API, four functions are defined. The function `process_create` is used to instantiate and start the execution of a new process. The function `func` is the entry point of the process. Note that the actual code of the process, be it hardware or software, is already available. The API function will create a new context for the new process and start executing the initial function. Also note that in case of hardware processes, if more

than one process share the same hardware implementation, there is a need to synthesize a scheduler within the hardware implementation, so that time sharing of the hardware is possible. `Process_delete` stops and removes a process from the scheduler list forever, freeing all the resources that were held by that process. Finally, `process_suspend` and `process_resume` are used to stop and resume the execution of a process, respectively. A process is suspended by a `process_suspend` call, and stays suspended until some other process executes `process_resume` for that specific process.

Two different communication models are supported in the API, message passing and shared memory. Message passing is abstracted by the concepts of ports, and provides the primitives `port_send` and `port_receive` to implement the communication. Blocking and nonblocking styles are supported, and are specified by the designer through the argument `mode`. A blocking send blocks the sender until the receiver reads the message. Similarly, a blocking receive blocks the receiver until a message is available in the corresponding port. Shared memory communication is modeled with the `shared_mem_read` and `shared_mem_write` primitives. Here, two styles are also possible, synchronous and asynchronous, specified in the parameter `mode`.

In the Synchronization section, three different synchronization mechanisms are defined by the API: mutexes, semaphores and condition variables. A call to `sema_wait` will block the calling process if the semaphore value is zero, meaning that none of the shared resources are available, while a call to `sema_post` increments the value of the semaphore, and unblocks a possibly waiting process. Mutexes are similar to binary semaphores, i.e., semaphores initialized with the value of one. The process calling `mutex_lock` will block in case the mutex value is zero, and `mutex_unlock` will set the mutex value to one, allowing one of the possibly waiting processes to continue. Finally, condition variables allow processes to wait for some event or condition to happen. The process calling `cond_var_wait` will block until the condition is met and the corresponding `cond_var_signal` is invoked. Alternatively, `cond_var_broadcast` can be used to signal an event when multiple processes should resume execution as a result of one event.

Finally, the Timing section allows the specification of the timing behavior of processes. Processes can wait for a fixed amount of time using the API called `time_wait`. The waiting time is provided in the parameter `time`. Additionally, it is also possible to specify timeouts for each of the blocking synchronization primitives, with `sema_wait_tmo`, `mutex_lock_tmo` and `cond_var_wait_tmo`.

3.3 Interface Synthesis

When the input design description contains communication primitives from the System Level API, there is a need to synthesize the communication interface

between the processes. Depending on the design partitioning, the interface will need to connect two hardware modules, two software modules, or a hardware and a software module. This phase is controlled through a web based interface that acts as an intermediate layer between UML and codesign and that will be presented in detail in Section 1.6.4 in the context of a real application.

In this section, we show examples of custom interface synthesis for different partitions. We refer to the process sending data as the producer, and the processor receiving data as the consumer.

Hardware to Hardware Communication. In the case where two processes that communicate through ports are mapped to a hardware implementation, there are different alternatives for interface synthesis. However, since this is a hardware to hardware communication, it is not necessary to generate RTOS code or software to handle this specific communication.

One possible architecture for a port based hardware to hardware communication is shown in Figure 1.2. In this case, there is a direct data connection between producer and consumer. Additionally, control lines are synthesized according to the API usage. If the port is ever used for a blocking send, then an acknowledge line from the consumer to the producer is necessary. Therefore, the producer is suspended until it receives an acknowledge from the consumer in case of a blocking communication. For communications with multiple consumers, the producer wait for the acknowledge of all consumers. This behavior is implemented with a logic OR of the individual acknowledges of the consumers, as shown in 1.2. Similarly, an event line is added from the producer to each consumer for the case when blocking receives are specified. Since the event and acknowledge control signals are only synthesized when needed, they are shown with dashed lines in Figure 1.2.

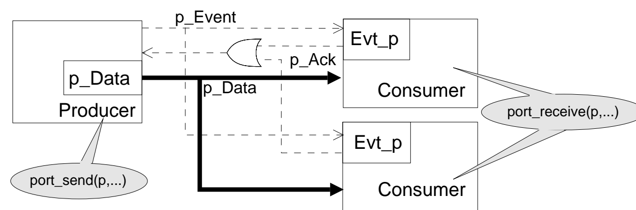

Figure 1.2. Interface Synthesis for HW to HW Communication

Other architectures are also possible from the same System Level API. For instance, it is possible to generate a Transaction Level Model with AMBA bus transactions for each port primitive. In this case, the `port_send` and `port_receive` primitives are replaced by a set of calls to the AMBA Transaction Level API [1].

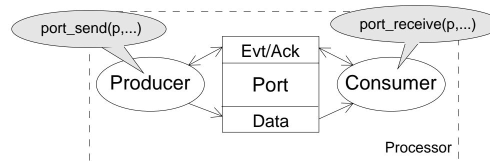

Figure 1.3. Interface Synthesis for SW to SW Communication

Software to Software Communication. When two software processes are mapped to the same processor, the interface synthesis is simpler. Our framework will generate a software data structure in memory, shared between the processes, that will keep the data along with event and acknowledge control signals. All the producer has to do is to update two memory locations, with data and event signaling (in case of blocking receives), while the consumer will read the data memory and update the acknowledge bit of the same port. Figure 1.3 shows the interaction between the processes.

Hardware to Software Communication. Hardware to software communications can be implemented by either interrupts or polling, using memory mapped addresses in the latter case. In both cases, we will need some RTOS support in order to coordinate the processes. One possible solution is shown in Figure 1.4. Our framework will generate a bus adaptation layer for the hardware module, so that it can send and receive data from the bus. In the case of a memory mapped communication, a device driver is also generated and runs inside the processor, monitoring the bus for activity in the memory mapped region. The device driver is responsible for transferring data from the bus to the processor memory, to an equivalent port structure as the one shown in Figure 1.3. The software process will access the port data structure as it did in the software to software case, retrieving data and updating event flags. If instead an interrupt based communication is specified, then an Interrupt Service Routine (ISR) needs to be synthesized. The ISR will be responsible for receiving the event signaling from the producer. In the interrupt based communication, the actual data is still transferred through a memory mapped location to the port structure.

Software to Hardware Communication. In software to hardware communications, the producer is running in a processor, communicating with a hardware module. In our model, this kind of communication is always memory mapped. The producer will update a *port* data structure, and a device driver propagates data and events to and from the bus. Events and acknowledge signals are generated for the receiver whenever necessary.

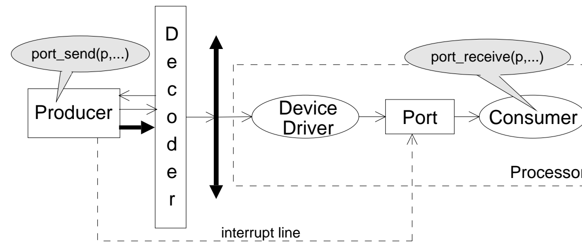

Figure 1.4. Interface Synthesis for HW to SW Communication

Note that the device driver can be unique for all the software to hardware and hardware to software communications. It has to monitor a set of software ports, transferring data to the bus, as well as monitor the bus for memory mapped communications.

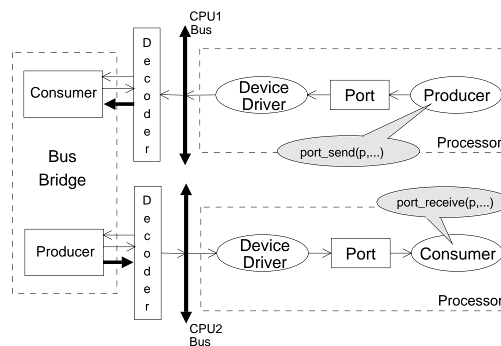

Figure 1.5. Interface Synthesis for Multiprocessor Communication

Multiprocessor Communication. Finally, in case the processes are mapped to different processors, with different buses, a bridge will also be synthesized. Figure 1.5 shows the proposed architecture. In this scenario, the producer runs on processor 1, connected to System Bus 1, while the consumer runs on processor 2, connected to System Bus 2. The producer will see the bridge as the consumer. Meanwhile, the consumer will see the bridge as the producer. Both processes will see a hardware to software communication, and the port will be accessed through a memory mapped address. In addition to the bridge, device driver code is synthesized for both processors, linking the software process to the RTOS and to the bridge hardware.

For shared memory communication, two different architectures are possible, depending on synchronous or asynchronous communication. In the syn-

chronous mode, a locking structure is generated for each shared memory, so that access is granted exclusively for each process. Every memory access has to obtain the lock first. In the asynchronous mode, only the memory is synthesized. The locking mechanism is implicit in the API call for shared memory access. Every shared memory will be directly connected to the system bus, accessible by the CPU. Additionally, a dedicated memory port will be available for each hardware module accessing the memory, so that using the bus is not necessary while accessing shared data. Therefore, there is less contention and higher parallelism in the implementation.

RTOS Synthesis. In addition to communication interface synthesis, the generation of RTOS support is required. In this case, our System Level API has to be mapped to OS specific resources, adapting the generic API to the functionality available in the target RTOS. Since our API is based on POSIX, the mapping is trivial when targeting a POSIX compliant OS, like Embedded Linux [20]. Alternatively, it is possible to target non POSIX RTOSes by mapping the API calls to the specific RTOS. That is the case with eCos [10]. Finally, the API based description can be used as input to tools that generate a customized OS infrastructure, like Polis [2] and Phantom [12].

3.4 Our HW/SW Codesign Environment

Input to our codesign environment is a set of modules M_1, $M_2 \ldots M_n$ that implement a design. Modules are described in SystemC extended with the proposed API functions. The SystemC modules are partitioned into hardware and software implementations. Currently, the partitioning process is manual. Once the design is partitioned, hardware, software and interfaces are synthesized. Hardware synthesis is handled by an inhouse SystemC behavioral synthesizer, that produces synthesizable RTL for each SystemC module. Software modules are generated according to the operating system support desired by the designer. At the time, our environment can generate software modules based on the POLIS framework [2], the Phantom Compiler [12] and any POSIX based operating system, like Embedded Linux [20] or eCos [10] with the POSIX adaptation layer. Software is compiled to a specific processor, which can be a NEC V850 or an ARM946. Finally, the interface is generated according to the partition and the communication style specified. We have simulators available that allow us to simulate the synthesized hardware, selected processor (cycle accurate in the case of V850 and instruction based on the case of ARM), software and communication interfaces.

4. Modeling Hardware Related Aspects in UML

This section deals with the hardware oriented modeling aspects of UML. In particular, it describes methods for specifying a system using the different flavors of UML diagrams, some depicting the structure and some depicting the behavior. It shows how hierarchy in hardware design can be represented at the specification level using available UML features. It also talks about our proposed enhancement of textual specifications and ways to integrate that in our codesign flow. Lastly, it speaks about the UML2.0 enhancements relevant to hardware oriented support, in particular the usefulness of timing diagrams as well as the specification of interfaces, ports and connectors. To create our model we used Rhapsody V5.2 which is the UML tool provided by I-Logix Inc.

4.1 Object Model Diagrams (OMD)

The OMD helps designers in modeling the structure of the system by means of classes. In our design flow we assume that each instantiated class is a functional system component. Figure 1.6 describes the top level OMD views for our example model that implements a simple matrix multiplication algorithm. It shows the block IndexControl, that controls the execution of the algorithm, a memory object and a hierarchical block MatrixMult. The OMD shows the static structure of the specified system, in particular, classes and their internal structure like the objects instantiated within them and relationships among the objects. The OMD can also show the relationships of a class with respect to other classes, such as inheritance or generalization and associations. We have used the OMD in a way to show the hierarchical view of a design, where each OMD shows the details pertinent to that hierarchy. In this way, the OMD can be used to represent hardware modeling. In our system, there are three basic partitionable objects (IndexControl, DataRetrieve, Multiplier) and a memory object. We have created two OMD's to show the hierarchical break down of the design. The top level object model diagram in figure 1.6 shows the highest level view of the design under test. Hierarchical objects are marked with stereotype *net* while basic partitionable objects are marked either as *Partitionable_Text* or *Partitionable_StateMachine*. The top level view also shows the input and output stimuli that needs to be generated from the test benches in order to simulate the model. The hierarchical component MatrixMult is described in figure 1.7 which shows its component classes like DataRetrieve and Multiplier. Also note that the same Memory object appears in both the OMD views to show the relationship it shares with different objects across different levels of hierarchy. All relationship between objects are specified using *links*, which are connected via *ports*. The links can specify event based communication or pure data communication based on the stereotype attached to them. Event based

communication triggers a transition in a state machine (see section 1.4.3), but pure data communications do not trigger any transitions. Another kind of link is used to specify the relation of an object to a memory. These links have an associated direction and are shown as an arrow. For example, in figure 1.6, the link between IndexControl and Memory belongs to one of this type.

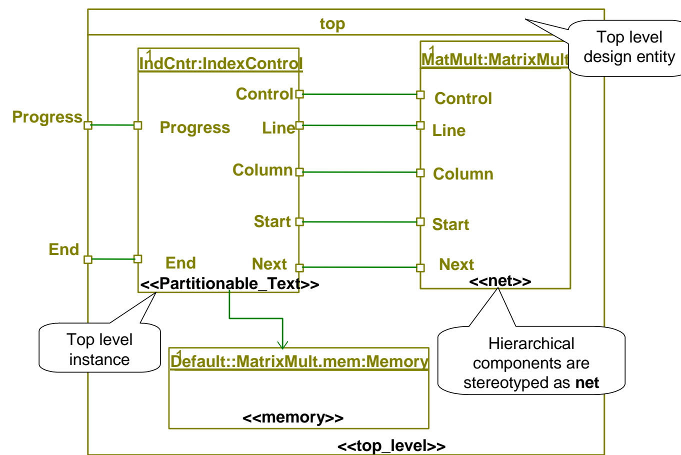

Figure 1.6. Top Level Object Model Diagram

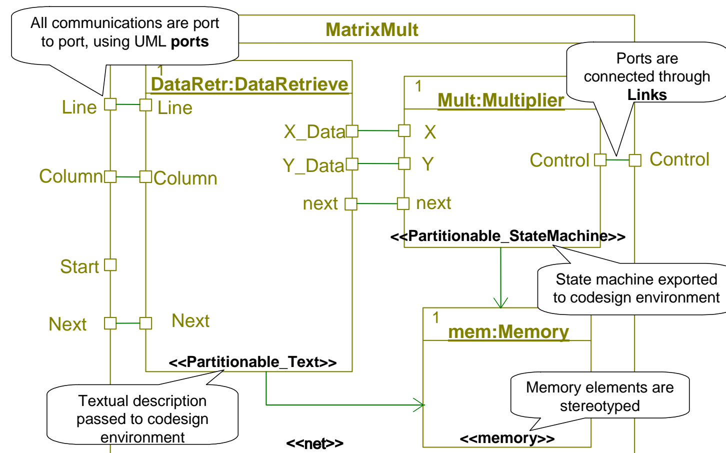

Figure 1.7. matrixMultiplier Object Model Diagram

4.2 Sequence Diagrams

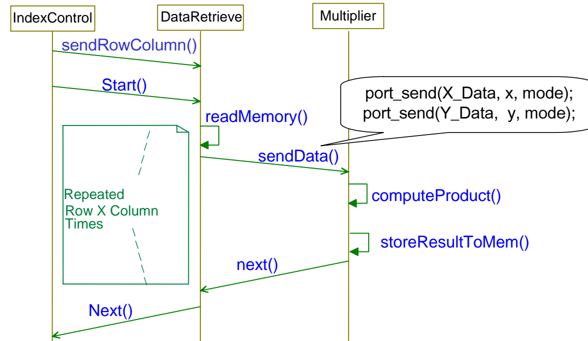

Figure 1.8. Sequence Diagram of matrixMultiplier

A sequence diagram shows object interactions arranged in time sequence. It shows the objects participating in the interraction and the sequence of messages exchanged between them. After Use Cases and OMDs have been developed, a Sequence Diagram can be specified as an additional form of interaction to help creating testbenches. Figure 1.8 shows the exchange of signals and their sequence between the various objects in the matrixMultiplier system. Data communication across the objects through the function calls is shown in the figure. API calls can appear inside the functions, for example the function *sendData()* calls internally the API *port_send()*.

4.3 State Machines

The next step is to create the state machines, that are descriptions based on Harel statecharts [7], used to model the behavior of each instantiated class in the system. The designer is responsible to figure out for each objects what the states are, and how transitions happen between them. The transition indicates one movement from one state to another. Each transition has a label that comes in three parts: `trigger-signature [guard]/activity`. All the parts are optional. States can also have some internal activity, like actions on entry, actions on exit and actions in state, and there are some mechanisms to specify a delay for executing a transition. States can be broken into several orthogonal state machines that run concurrently and superstates can be used in order to share common transitions and internal activities among states. As an example, Figure 1.9 describes the state machine for the IndexControl object in the matrixMultiplier example. The Index Control is responsible for the execution sequence of the matrix multiplication. It is basically composed of two nested loops, that advance the current line and column of the multiplication. Current

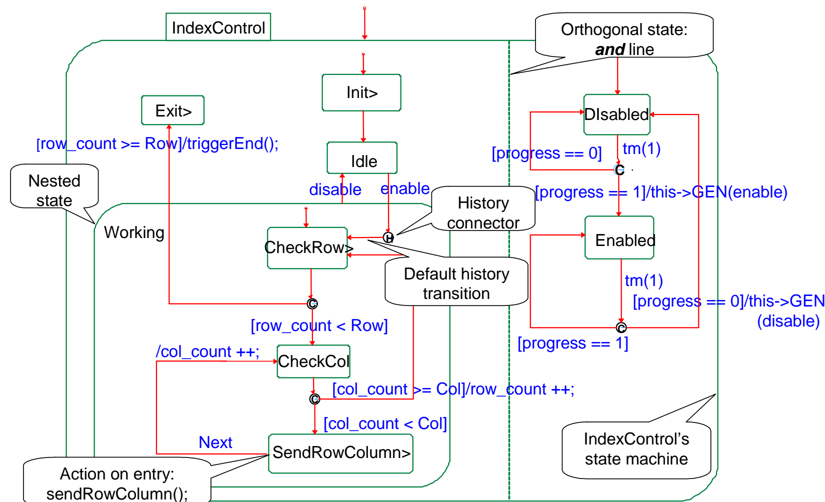

Figure 1.9. State Machine of IndexControl

line and column are communicated to the Data Retrieve module by two ports, named Line and Column.

Here is shown an `AND` state containing a nested state machine with a `history` connector. An `AND` state is an orthogonal state which represents simultaneous independent substates that an object can be in at the same time. A `history` connector stores the most recent active configuration of a state, so a transition to a history connector restores this configuration.

4.4 Textual Specifications

Figure 1.10 shows how we manage the textual format of the behavioral description. Through the Rhapsody tool we have to set, for each system component, which is the stereotype that specify the type of the behavioral description. In the "Description dialog box" we edit the textual (SystemC) description of the system module. A module can potentially contain both a state machine as well a textual description for its behavior in the form of SystemC. In the codesign phase, it will be possible to associate different instantiations of the same module to different form of specifications, i.e some to state machines, some to textual description.

4.5 UML 2.0 Enhancements

UML 2.0 enhancements are not changing dramatically the modeling elements of UML 1.x. As it is said in [5]: "UML 2.0 doesn't represent a substantial re-

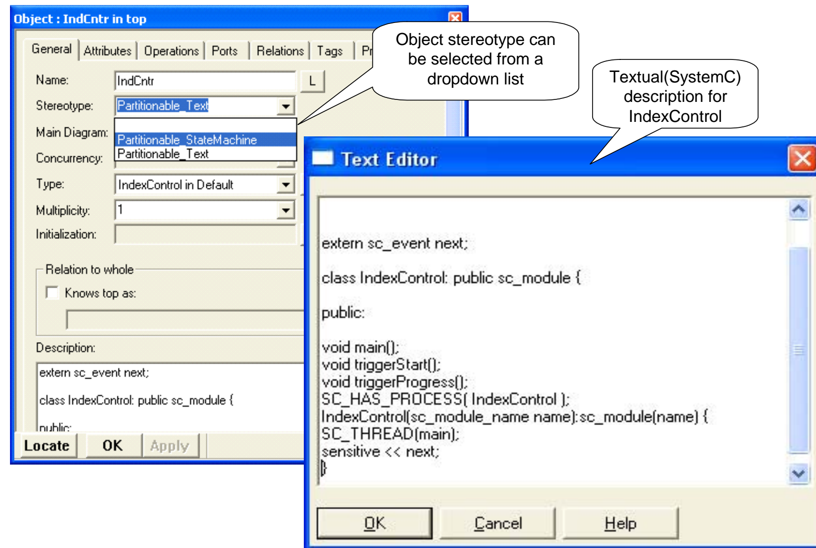

Figure 1.10. Behavioral Description: Textual Format

definition of the modeling elements". Most of the changes were performed in the behavioral diagrams rather than in the structural diagrams. For instance in the interaction diagrams the collaboration diagram disappeared, while the sequence diagram notation is now able to support nested diagrams notation and conditional behavior. In addition there are three new interaction diagrams: timing, communication and interaction overview diagram. Among the behavioral diagrams, the activity diagram was improved significantly and now it is possible to model the concurrent behavior relying on tokens similar to Petri Nets [15].

In this section we will talk more about UML2.0 enhancements relevant to hardware modeling. In particular we will focus on structural diagrams like the Deployment diagrams. We will also spend a few words on an interaction diagram, the timing diagram, and its related elements specialized for realtime systems. In addition we will talk about the specification of interfaces, ports and connectors.

Deployment Diagrams. A Deployment diagram captures the configuration of runtime processing elements and the software component instances that reside on them. It is a graph of nodes, representing the hardware resources, and communication paths representing physical connections among the resources. A node can be a CPU or some other processing element and can have its own memory. Components represent software modules, tasks or processes that run on a node. Hence deployment diagrams specify the runtime physical architecture of a system.

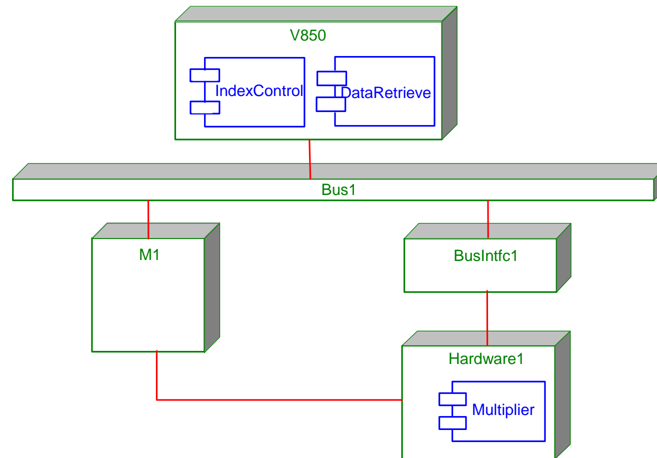

Figure 1.11. Deployment Diagram: Platform Specification and Component Mapping

Platform Specification using Deployment Diagrams. The deployment diagram can be used to specify a platform architecture in our proposed methodology. Normally the user has to select an architecture from a list of predefined platforms to be considered in the codesign phase. These predefined platforms are shown as an interconnection of hardware resources like CPU, hardware elements and memories, connected by means of buses or dedicated point to point connections. Depending on the user's choice to map a design module either to hardware or software, the modules are deployed to the corresponding elements in the architectural platform. We propose an additional stage in our design methodology where the platform can also be specified graphically by the user, making use of the UML Deployment diagram. Other than the predefined platforms, we would also provide the basic resources like the CPUs, switches, hardware elements, buses, bus bridges, etc., from which the user can select, and connect them using communication paths to build his own platform. There would be necessary checkings to ensure the semantic correctness of the usage of the architectural components as well as their interconnection protocol. The design components can then be deployed to the nodes in the architectural platform. Information from the deployment diagram would then be exported to the codesign environment for further steps to cosimulation.

An example of our proposed scheme is shown in Figure 1.11. It shows the V850 platform specification and the default configuration of the modules in the matrixMultiplier application.

Timing Diagrams. The timing diagram shows the change in state along a lifeline in terms of a defined time unit. Figure 1.12 describes the timing diagram related to the behavior of the IndexControl object. The diagram is related to Figure 1.9 which describes the state machine of IndexControl. The states represented by the timing diagram of IndexControl are: Init, Idle, Working and Exit. The object will change its internal state depending on the event that will occur. Events are: enable, disable and end. Figure 1.12 shows that when the event enable occurs, the IndexControl goes from state Idle to state Working, whereas it changes from Working back to Idle when the event disable occurs. The module goes in state Exit when, while beeing in state Working, the event end occur.

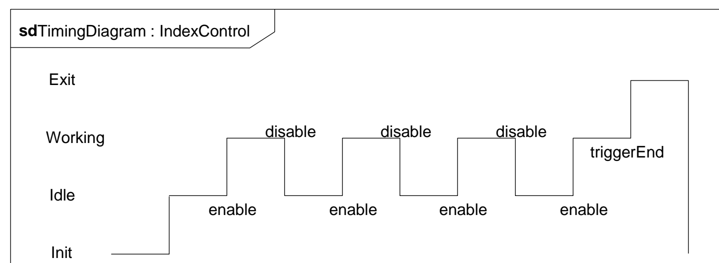

Figure 1.12. Timing Diagram for IndexControl

Timing diagrams are very helpful to specify the duration and timing constraints of realtime systems.

Interfaces, Ports and Connectors. Figure 1.13 shows how we model interfaces, ports and connectors between IndexControl and MatrixMult objects. Interfaces are specified through ports and connectors. Ports are identified by little squares on the object boundaries while connectors might be little plain circles or arcs. Circled connectors describe the provided interface (e.g the object sends a signal through this port), arc connectors describe the required interface (e.g. the object waits for a signal through this port).

Interface direction (input/output) can be shown graphically only using UML 2.0 semantics which allows to specify whether an interface requires (input) or provides (output) a service (a signal in our case study). For links, the direction cannot be specified, so we are using an input/output stereotype attached to a port in order to specify the direction. Connections to memories do not use ports, but direct links with objects.

A port is an interaction point assigned to an object and can exchange messages with other external objects or send messages to and from their parts. A port enables to specify instantiated classes independently of the environment in

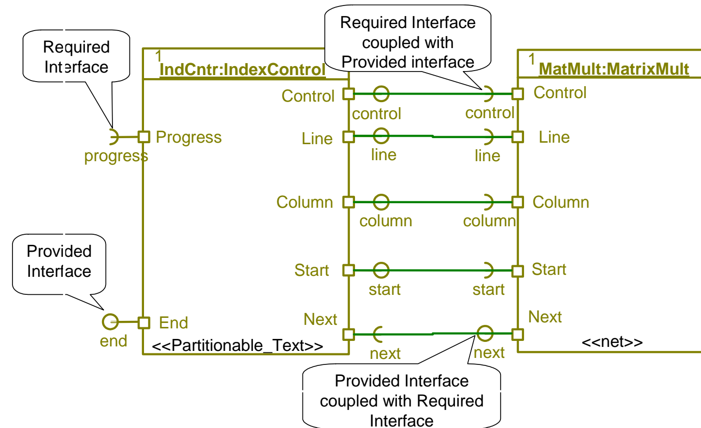

Figure 1.13. Interfaces, Ports and Connectors between IndexControl and MatrixMult

which they will be embedded. The internal part of the object can be completely isolated from the environment. In our methodology, ports are used to extract the interface signals of a module needed in its SystemC implementation (see section 1.4.1).

5. Model Verification in UML

This section deals with verification aspects of UML. In particular, we discuss the event semantics in UML, and propose some UML enhancements for supporting a pure discrete event simulation that is more suitable for hardware modeling. We also describe how to take advantage of other useful UML features like animated sequence diagrams and state machines during the system verification process.

During the design phase, designers should periodically validate their UML models so that they can find bugs very early in the project. Discovering bugs in the design phase is much cheaper than in later phases. In this section we present animated diagrams, that are important features provided by UML tools, and we also discuss event semantics in UML.

5.1 Animated Sequence Diagrams and State Machines

The first technique we use is a particular feature provided by the UML tool Rhapsody from I-Logix([16]), which allows the designer to simulate the model by animating its sequence diagrams and state machines. This allows the designer to visualize the system behavior during a specified test case and validate the model. Rhapsody also provides the possibility to compare the animated sequence diagrams with those developed during the Analysis phase. This helps

in validating the model versus the requirement specification. Figure 1.14 shows the animated state machine of IndexControl at the beginning of its behavior. It can be seen that the "IndexControl" main state and the initial "Init" state are highlighted by means of a violet colour. This means that the IndexControl object is in that particular state at that moment. This is very useful for designers when they need to verify the model behavior.

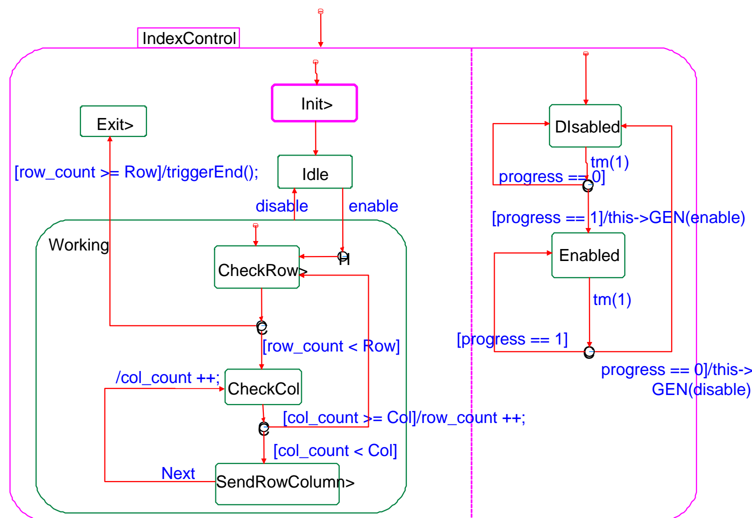

Figure 1.14. Animated State Machine

5.2 Event Semantics in UML

The native communication model in UML is based on asynchronous events with a single queue. In this model events are processed in the following fashion:

1. An event is created when it is sent by one object to another.
2. It is then queued on the queue of the target object thread.
3. An event that gets to the head of the queue is dispatched to the target object.
4. The event is processed by the receiving object and then deleted by the execution framework.

The main drawback of this semantics is that it is essentially SDL-like, and hence it is not adequate for hardware modeling where instead a real discrete event engine would be needed. In particular a global event queue and a support

for event ordering based on their timestamps is mandatory for simulating real hardware.

UML 2.0 has tried to address this issue by allowing to manage events as an event pool without defining a priori their order of dispatching. This leaves open the possibility of modeling different models of computation. Methodologies like the one presented in [6] have already shown how it is possible to extend UML very easily and efficiently in order to support new models of computation.

Unfortunately, many UML tools that are currently on the market still use the native communication model of UML and hence it is not yet possible to rely on a general solution for modeling hardware behavior in UML. For this reason, we have decided to use UML only for system specification but not for hardware/software cosimulation and validation.

6. Transformation from UML to Codesign

In this section we present the link between UML specifications and the ACES hardware/software codesign environment known as ACES. Items that will be discussed are: UML database exploration, behavioral code generation and export of structural information necessary for the codesign environment.

6.1 UML Database Exploration

After the application is modeled and analyzed using the UML tool, we get a repository that contains information of the model in the internal database. We have used Rhapsody from I-Logix, Inc. as UML tool. The database generated by Rhapsody is organized as follows. The main project consists of a list of packages. A package is a mechanism to organize different project elements into groups. A package consists of the list of classes, functions, objects, events, global variables, diagrams as well as packages. Each class has a list of attributes, methods, objects instantiated within it, links between the objects and other modeling elements.

6.2 Code Generation

In this phase, the behavior of the modules specified in UML is converted to SystemC code in order to be imported into the ACES codesign environment. From this SystemC representation, ACES is then able to perform both hardware and software synthesis. In our proposed methodology, the behavior of a module can be specified either through a state machine as shown in Figure 1.9 or a textual specification as shown in Figure 1.10. The type of the specification can be selected from a drop down list.

When the user selects the textual specification, the code generator just copies the user specified code into the file needed by ACES. Input and output descrip-

tion for the ports and signals would be automatically extracted by the code generator from the OMD.

On the other hand, when the user specifies the behavior through a state machine, the code generator has to explore the UML database in traversing the states of the state machine and generate the corresponding code. A sketch of the algorithm for this code generation process is shown in Figure 1.15. The algorithm needs to know the events in the model, the action/guard for the transitions, entry and exit actions in a state, in transitions to a state, out transitions from a state, etc. It is possible to browse through the entire object model and extract the relevant information from the state machine. Also UML allows the behavior to be specified using Activity Diagrams, which are very similar to State machines, and the same algorithm can be used for generation of the behavioral code.

The algorithm is called on the root state of each state machine for which code has to be generated. In every state, it first emits the code specified by the user in the action on entry portion of the state. Then it checks out transitions from the state. For the transitions triggered by events, it issues a wait statement on that event, then it emits the code specified in the action on exit portion, followed by a goto statement, the label being the target state. In case of a conditional transition, it issues an "if then else" statement with goto labels depending on the condition. It also issues the code (if any) specified in the action section of the transition. Then the algorithm is called recursively on each state reachable by the current out transition. For an **AND** state, the same code generation algorithm is called on each of the substates within the **AND** state. The behavior is also the same for a state with a nested state machine.

The output of the code generator is a list of SystemC files, each corresponding to the behavior of a specific object in the system to be considered in the codesign flow. Figure 1.16 shows a portion of a SystemC description generated for the state machine of object IndexControl corresponding to Figure 1.9. We have chosen an unstructured style for the generated code, due to its simplicity and efficiency, but many variants (e.g. nested switch, state pattern, state tables, etc.) are possible. Events are implemented as boolean terminals.

We have implemented and tested the algorithm using Rhapsody UML tool, which provides API functions that allow us to extract all required information from a UML project database. However we would like to emphasize that this code generation algorithm is very general and can be utilized also with other UML tools.

6.3 Exporting Structural Information

In order to start with the codesign process, the last thing we need is to extract a summary of the design, essentially a textual representation containing a list

```
codeGenerate(state S) {
 1. If S is visited, return;
 2. Mark S as visited.
 3. Issue code specified in the action-on-entry section (This code can
    be directly copied)
 4. Get out transitions {T}  from state S;
 5. {U} = empty;
 6. for each out-transition 't' of {T} do {
       if 't' is conditional {
          issue code specified in the action-on-exit section;
          s_t = target state if condition is true;
          s_f = target state if condition is false;
          issue if-then-else with goto label as 's_t' or 's_f' depending
          on condition;
          insert 's_t', 's_f' in {U}; }
       else {
          s = target state of 't', insert 's' in {U};
          if 't' is triggered by event 'e' {
             issue wait on event 'e';      }
          issue code specified in the action-on-exit section;
          issue goto with label as 's';  }
       issue code specified in the action section of transition 't'; }
    for each 'u' in {U} do
       codeGenerate(u);
  }
```

Figure 1.15. Algorithm to Extract Code from UML State Machine

```
#include <IndexControl.h>
extern sc_int<8> mem[Row*Column];  // External memories
 SC_MODULE(IndexControl) {
  sc_in_clk clk;
  sc_in<bool> rst;
// Input terminals
sc_in<bool> Next;       // input event */
// Output terminals
sc_out<bool> End;    // output event
sc_out<sc_int<8>> row;  // output data
… Omitted …
 SC_CTOR(IndexControl) {
   SC_CTHREAD(main,clk.pos());
   watching(rst.delayed() == 0);
 }
void main(void) {
 ...
CheckRow:
          col_count=0;
          if ( row_count < Row ) {
                   goto CheckCol;
          }
          else {     triggerEnd();   // Send End
                   goto Init;
          }
CheckCol:   ... Omitted ...
 };
```

Figure 1.16. Code Generated for Module IndexControl

of all the partitionable objects and their interconnections as well as description of the memory object. More specifically, this phase generates the files which are necessary by ACES as input to proceed to cosimulation.

```
Main {
  Open UML project database
  Get list of packages
  For each package do {
    get list of defined classes
    find class from list marked as top_level
    DFS_Traverse(top level class) }
}
DFS_Traverse(class C) {
  Mark class as visited
  get list of object instantiations in class
  For each object do {
    If object is a memory instance {
      generate memory descriptions }
    else {
      generate structural descriptions
      put object's master class in DFS_List }
  }
  For each master class in DFS_List do {
    DFS_Traverse(master class) }
}
```

Figure 1.17. Pseudocode for Extracting Structural Information

In order to export the structural information to ACES, we need to traverse the design hierarchy and generate the textual descriptions. The algorithm is described in Figure 1.17. The algorithm makes a breadth first search traversal of the design hierarchy and generates the text files. In order to identify the highest level of the hierarchy, the user needs to specify a stereotype *top_level* to the top level module. Any intermediate hierarchical modules are stereotyped as *net*, whereas the leaf level modules are marked either as *Partitionable_StateMachine* or *Partitionable_Text*. Example of the text files generated for the matrixMultiplier example is shown in Figure 1.18. The generated files consists of the following:

1. A file describing the structure of the system. It consists of all the class instantiations in a hierarchical fashion, showing the inputs and outputs at each level of hierarchy and also the port connection of the instantiated classes.

2. A file describing all the signals that are necessary for connecting the objects of the system. It shows the list of signals along with their source and destination objects, and also in particular the ports of the object with which the ends are connected. Any source or destination which is in the outer hierarchy is shown as *OUT*.

3. A description of the memory objects used in the system, specifying the memory name, objects that access the memory, and other details like total size of the memory, word length and access type.

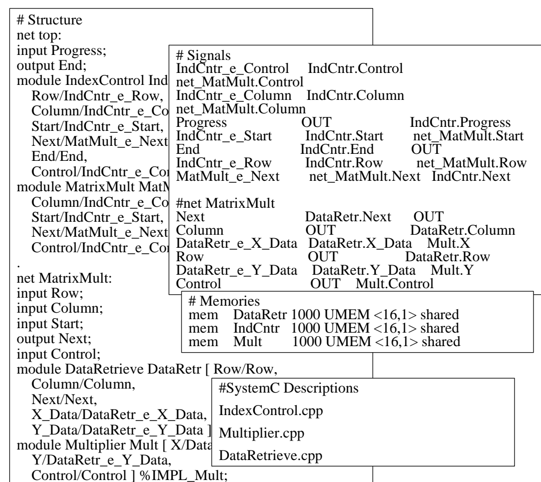

Figure 1.18. Generated Input Files for Codesign Environment

6.4 Web Based Interface

Figure 1.19 refers to the HTML page that is generated at the beginning of this phase. The two screen shots show the same page and respectively the top part (left side) and the bottom part (right side). This page can be opened using any web browser and is organized as follows. Starting from the top, there is a brief summary of the project containing its name and a short description. By clicking on a link, it is possible to see all the verbose report provided by the UML tool containing all the information about the project that has been collected in the UML database. The third line is used in order to select the platform onto which to implement the desired functionality. The selection is performed through a menu window where the user can pick any of the architectural templates available in a library provided with the codesign tool. An architectural template represents the platform for the system implementation and the user is responsible for selecting the platform that is best suitable for the system that he needs to implement (one or multiple CPUs, DSPs, simple or very complex bus hierarchy, etc).

The selection of the platform is directly reflected in the graphical content, presented in the middle of the page, where on the left side there is the functional view of the system exported from the UML specifications and on the right side there is the picture of the selected platform. By changing the target platform,

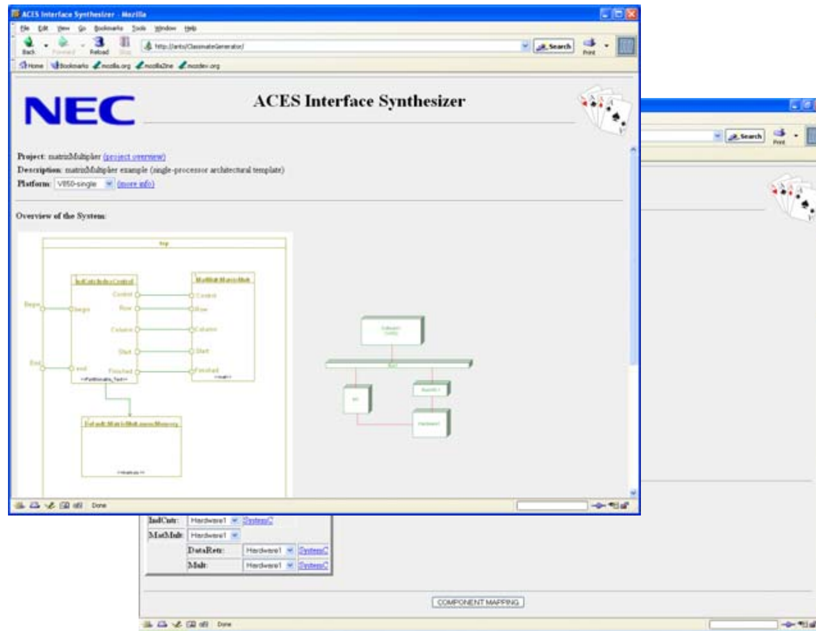

Figure 1.19. HTML Page Generated from UML Specifications

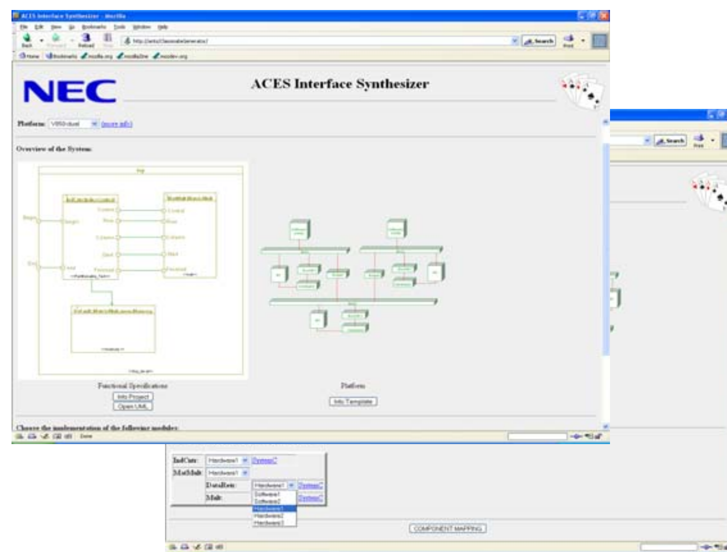

Figure 1.20. Mapping on a Dual Processor Architecture

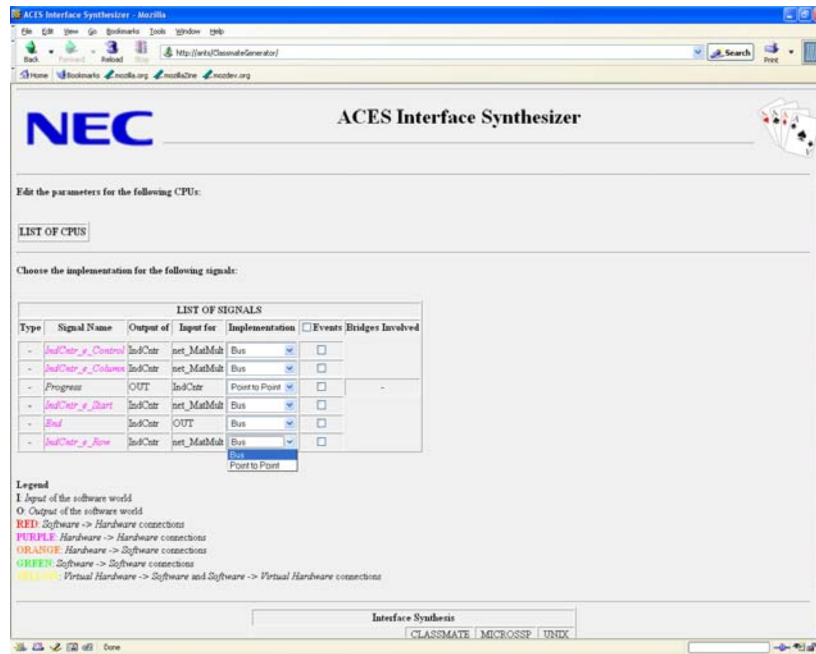

Figure 1.21. Communication Refinement

the picture on the right is automatically updated. For example in Figure 1.19 the platform contains only one processor, while in Figure 1.20 the platform contains two processors and a two level bus hierarchy. The idea behind this solution is to support a function architecture codesign approach that requires the separation of the functionality from the architecture selected for its implementation.

Finally, at the bottom of the page are listed all the objects present in the functional specifications and the user can specify the implementation (i.e., the hardware or software component of the platform onto which the functionality will be implemented.) for each of them through a menu window associated to each objects. This is what we call *component mapping phase*. The number of choices available for this mapping depends on the selected platform. For example, in the platform shown in Figure 1.19, only two choices are possible (Software1, Hardware1) due to the fact that it is a simple single processor architecture with one hardware component connected to the processor bus. But in the multiprocessor architecture shown in Figure 1.20, five different choices are possible, since in this platform there are two processors and three hardware units.

The component mapping phase ends when the user clicks the button "COMPONENT MAPPING". This starts the process of analysis and characterization

of all interface signals and opens a new html page, like the one shown in Figure 1.21, where all signals are listed.

At this point, the *communication mapping* phase can start. The table shows all different types of connections: hardware to software, software to hardware, hardware to hardware and software to software connections. A connection can be recognized by its name, a color associated to its type, its source and its destination. The last column shows the specific implementation of the connection. Software to hardware connections are implemented in memory mapped I/O, while hardware to software connections are by default implemented in memory mapped, but the user can alternatively specify an interrupt based implementation. Hardware to hardware signals are by default implemented as point to point communications, but the user can alternatively require the communication to be performed on the bus (memory mapped). Finally, software to software connections are implemented by the realtime operating system (RTOS). This list of signals presented in the table refers to a specific CPU in the selected platform and its associated system bus. In case of multiprocessor platforms, several list of signals, one per CPU, are generated.

When all implementation options have been specified, the user can proceed to the communication mapping phase. At this point physical addresses will be generated for all memory mapped communications and specific interrupt lines of the processor will be selected for signals implemented in interrupt. The result is a new page, not shown here, similar to Figure 1.21, but where the last column shows now the physical addresses and the interrupt lines that have been selected. After having examined all the communications, the user can still go back and change some implementation options or, if satisfied, proceed to the next hardware/software cosimulation phase.

7. Conclusions

The complexity of current embedded systems requires large teams of designers that interact especially at the early design stages when architecture selection and hardware/software partitioning take place. Models and tools that allow to visualize and document the design abstractions and the interactions between different components or levels of abstraction of a specification are essential. UML being platform independent and with a rich graphical notation can serve this purpose. We presented a methodology that specializes the UML standard notation for modeling embedded systems platforms and protocols leading to an integration with an existing hardware/software codesign technology.

References

[1] ARM Limited. AMBA AHB Cycle Level Interface Specification, 2003.

[2] F. Balarin et.al. *Hardware-Software Co-Design of Embedded Systems: The Polis Approach*. Kluwer Academic Publishers, 1997.

[3] A. S. Basu, M. Lajolo, and M. Prevostini. UML in an Electronic System Level Design Methodology. In *UML-SOC'04 (DAC Workshop), San Diego, California*, pages 47–52, 2004.

[4] CriticalBlue Homepage. `www.criticalblue.com`

[5] H. E. Eriksson, M. Penker, B. Lyons, and D. Fado. *UML 2 Toolkit*. Wiley Publishing Inc., 2004.

[6] P. Green, M. Edwards, and S. Essa. Enhancing UML to Support the Specification of Behavior for Embedded Systems-on-a-Chip. In *UML-SOC'04 (DAC Workshop), San Diego, California*, 2004. pages 53–58.

[7] D. Harel. Statecharts: A Visual Formalism for Complex Systems. In *Science of Computer Programming*, 1987.

[8] M. Lajolo. IP-Based SoC Design in a C-Based Design Methodology. In *IP Based SoC Design 2003*, 2003. pages 203–208.

[9] M. Lajolo, A. S. Basu, and M. Prevostini. UML Specifications Towards a Codesign Environment. In *Proceedings of FDL'04*, Lille, France, 2004. pages 313–324.

[10] A. Massa. *Embedded Software Development with eCos*. Prentice Hall, 2002.

[11] Mentor's Application Specific Assistant Processor. `www.mentor.com/asap`

[12] A. Nacul and T. Givargis. Code Partitioning for Synthesis of Embedded Applications with Phantom. In *Proceedings of ICCAD'04*, Nov 2004.

[13] S. Narayan, F. Vahid, and D. Gajski. System Specification with the SpecCharts Language. In *IEEE Design and Test of Computer*, 1992. pages 6–12.

[14] Object Management Group (OMG) Homepage. `www.omg.org`

[15] W. Reisig. *Petri Nets: An Introduction*. Springer-Verlag, Berlin and New York, 1985.

[16] Rhapsody Homepage. `www.ilogix.com/rhapsody/rhapsody.cfm`

[17] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.

[18] K. Scott. *Fast Track UML 2.0*. Apress, 2004.

[19] Synfora Homepage. `www.synfora.com`

[20] The Embedded Linux Consortium Homepage. `www.embedded-linux.org`

[21] The Open Group and IEEE. IEEE Std 1003.1, 2004. `www.opengroup.org/onlinepubs/009695399/toc.htm`

[22] Unified Modeling Language (UML) Homepage. `www.uml.org`

[23] K. Wakabayashi and T. Okamoto. C-Based SoC Design Flow and EDA Tools: An ASIC and System Vendor Perspective. *IEEE Trans. Computer-Aided Design*, 19(12):1507–1522, Dec 2000.