UML in an Electronic System Level Design Methodology

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

[1] ALaRI, University of Lugano, Lugano, Switzerland
[2] NEC Laboratories America, Princeton NJ 08540, USA

Abstract. The interest in System-On-Chip (SoC) design using the Unified Modeling Language (UML) has been growing significantly during the last couple of years. In this paper we would like to present a methodology that aims to address embedded systems design issues at multiple levels of abstraction and to support a function/architecture codesign process. Our approach integrates UML with high-level synthesis and hardware/software co-verification techniques in order to provide an automated flow for SoC design starting from system-level specifications down to hardware/software partitioning and integration. UML has been selected because it is platform independent and helps team members to share very efficiently relevant information during the various design phases, while high-level synthesis helps to evaluate constraints that the embedded system must satisfy: e.g. performance, power and cost starting from behavioral specifications. The paper aims to give a contribution towards SoC Design automation from System-level specification to hardware/software partitioning.

1 Introduction

The increasing complexity and the shortening of the time-to-market windows make the design of electronic systems a challenging task that can no longer be handled by traditional methodologies. New methodologies are needed to improve design productivity and derive high-performance low-cost implementations. This requires to develop formal methods that synthesize correct-by-construction implementations and maximize 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 [1] 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 [2].

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) [3] that leverages on high-level synthesis and co-verification 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 [4]: CYBER and CLASSMATE. CYBER is a behavioral hardware synthesis tool that can provide the link to implementation that is missing in all the alternative environments and CLASSMATE is a hardware/software co-verification tool that can be exploited in order to derive accurate and fast timed functional models for behavioral IPs. 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 paper is organized as follows. Section 2 shows the proposed flow and the Co-design environment used in this methodology. Section 3 describes our methodology for SoC design starting from UML specification and in Section 4 you will find our conclusions.

2 The Proposed Flow

The overall flow presented in this paper is shown in Figure 1. Our proposed methodology starts with the UML specification of the system, 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 codesign environment. The following sections provide additional details about this design flow.

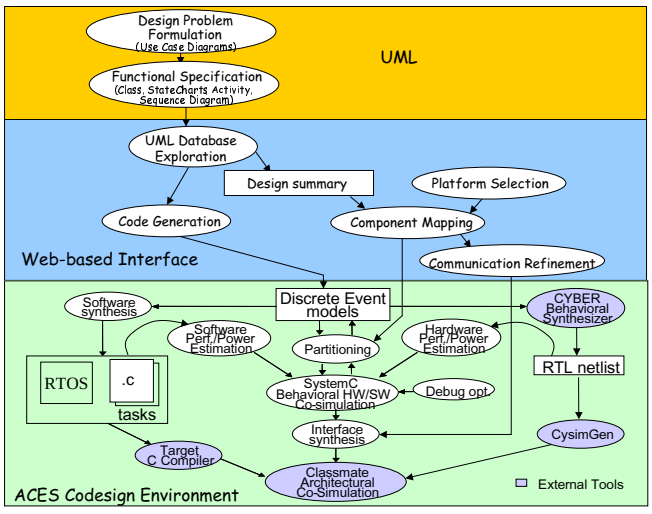


Fig. 1. The overall design flow.

2.1 UML Specifications

The Unified Modeling Language (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 [1].

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 is defined to capture the structural decomposition of the system into interacting components. Each class in this diagram corresponds to a functional component in the system specification. Classes are divided in two sets: the ones whose behavior could potentially be implemented either in hardware or in software and others that do not have to enter in the codesign flow. Examples of the second set of classes are testbenches and strictly software oriented components. Classes belonging to the first set are distinguished by the ones of the second set using the *Partitionable* stereotype that has to be specified manually by the user (see [5]). Communication among classes can be specified through uni-directional relationships, associated to events, or by means of shared variables and we provide a specific API (further details are given in section 3.2) in order to guide the designer in this modeling phase. All partitionable classes are required to have a state diagram associated for specifying its run-time behavior, while non-partitionable classes may or may not have a state diagram associated.

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 interface for specification with the possibility to use the analysis and synthesis tools available using the ACES codesign methodology.

2.2 The ACES Codesign Flow

The back-end of the proposed methodology is the ACES codesign flow that is depicted in the bottom part of Figure 1. The system is described at the behavioral level as a network of discrete event models (tasks) that can communicate by both means of events as well as shared variables. Those models have a precise semantics and are written in SystemC. For each module in the system specification, ACES can synthesize a hardware netlist, a software program and the interfaces between hardware and software, based on partitioning and communication mapping information given manually by the user on a module by module basis. Behavioral SystemC co-simulation is used to test the behavior of the system and to perform hardware/software partitioning in a closed loop. Good estimates of both hardware and software performance and power are of crucial importance in this phase in order to avoid costly design reiterations. During the co-simulation, hardware modules run concurrently, while the operation of the software modules is coordinated by a scheduler modeling the RTOS used in the final implementation. The interfaces between the modules are also abstracted out using a transaction-level paradigm. Once a suitable hardware/software architecture has been identified, hardware, software and the hardware/software interfaces synthesized by ACES can be exported into the CLASSMATE co-verification environment and can be simulated and verified at the architectural level.

ACES provides the unique possibility to change the hardware/software implementation of each component in the system by simply changing an implementation parameter in the web browser. The same simulation code is used to simulate the functionality for both hardware and software implementations. The only things that change are the delay annotations that are used for modeling performance and power consumption and also the scheduling policy of the module in order to model shared system resources like the CPU.

3 From UML to Co-design

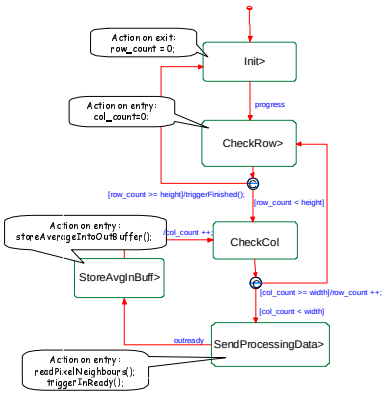
The link between UML specifications and an existing methodology for hardware/software codesign is the core of this paper. After the application is modeled and analyzed using the UML tool, we get a repository that contains information of the model in the internal database. We have used Rhapsody from I-Logix, Inc. as UML tool. We have found very useful the API's provided by Rhapsody to extract information from the repository and generate the input files for the ACES environment. The transformation process has two phases:

1. code generation for synthesizable models, and
2. export of structural information

3.1 Code generation from state diagrams

In our UML specifications all partitionable entities must have an associated state diagram, which is a description based on Harel statecharts [6], used to model the object behavior. The designer is responsible to figure out for each module 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. The `trigger-signature` is usually a single event that triggers a potential change of state. The `guard`, if present, is a boolean condition that must be true for the transition to be taken. The `activity` is some behavior that is executed during the transition. States can also have some internal activity, like actions on entry and actions on exit, and there are some mechanisms to specify a delay for executing a transition. States can be broken into several orthogonal state diagrams that run concurrently and superstates can be used in order to share common transitions and internal activities among states. The state diagrams describing the behavior of each partitionable class need to be converted into SystemC in order to be imported into the ACES codesign environment. From this textual representation, ACES is then able to perform both hardware and software synthesis.

Figure 2 shows on the left an example of state diagram and on the right the pseudocode of the algorithm that we have developed for automating this code generation process. The algorithm utilizes the Rhapsody's API in order to extract various information like list of classes, global variables and events in the




```
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);
}
```

Fig. 2. An example of state diagram (left) and the pseudocode of the code generation algorithm (right).

model, the action/guard for the transitions, entry-action and exit-actions in a state, in transitions to a state, out transitions from a state, etc.

The algorithm is called on the default state of each state diagram 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.

3.2 Exporting structural information

In order to start with the codesign process, the last thing we need is to extract from the UML specifications a summary of the design, essentially a textual representation containing a list of all the partitionable modules and their interconnections. In order to identify the partitionable modules, we require the user to specify the stereotype *Partitionable* on those modules that need to be considered in the co-design process. This is needed because the entire design usually contains some modules which do not need to be synthesized neither in hardware, nor in software, like the testbenches.

UML allows to specify the description of a model through a wide variety of styles, but in order to perform a tight link with a codesign tool, we had to impose some restrictions to the user. In our system, the communication between partitionable entities can be described using events, data ports and shared variables. Events are a point-to-point communication mechanism used to describe

the reactive behavior of a module and they generally trigger some transitions in a state diagram. Data ports are also a point-to-point communication mechanism, but they differ with respect to events because they do not trigger any transition. Their value can instead be used anywhere within the state diagram code. Finally, communication by means of shared variables is generally used in order to describe multiple access capabilities to a data that can be shared among different modules.

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 internal implementation of this API is completely transparent to the user. We have implemented and tested it within the Rhapsody UML tool, but it can be supported in any other UML-based kind of technology. The macros of this API can be identified very easily during the exploration of the UML database of a project and allow us to export the information that we need for the following codesign phase.

4 Conclusions

The complexity of current embedded systems requires large teams of designers that interact especially at the early stages where the design task is partitioned. 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 is a modeling language with a rich graphical notation, it is platform independent and can serve this purpose. We presented a methodology that specializes the UML standard notation for modeling embedded systems platforms and protocols. Then we proposed an integration with an existing hardware/software codesign technology. The paper aims to give a contribution towards system-level design automation.

References

1. J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1998.
2. I. Barnard, "Using UML 2.0 to solve Systems Engineering Problems," *Telelogic White Paper*, Dec. 2003.
3. M. Lajolo, "IP-Based SOC Design in a C-based design methodology," in *Proc. of IP Based SoC Design 2003*, pp. 203–208, Oct. 2003.
4. K. Wakabayashi and T. Okamoto, "C-Based SoC Design Flow and EDA Tools: An ASIC and System Vendor Perspective," *IEEE Trans. Computer-Aided Design*, vol. 19, pp. 1507–1522, Dec. 2000.
5. A. Minosi, S. Mankan, A. Martinola, F. Balzarini, A. Kostadinov, and M. Prevostini, "UML-based Specifications of an Embedded Systems Oriented to HW/SW Partitioning: a Case Study," in *FDL'03 Proceedings*, pp. 226–237, Sep. 2003.
6. D. Harel, "A visual formalism for complex systems," in *Science of Computer Programming*, 1987.