Timed Executable System Specification of an ADSL Modem using a C++ based Design Environment : A Case Study

Dirk Desmet Michiel Esvelt Prabhat Avasare Diederik Verkest Hugo De Man
IMEC, Kapeldreef 75, B-3001 Leuven, Belgium

Abstract

In this paper we propose a C++ based cosimulation and codesign environment, that allows to specify the timing behavior of the components of a complex hardware-software system independently of the functional refinement. While the hardware models are at a high functional abstraction level, thus resulting in a high simulation speed, yet the timing behavior can be specified with sufficient granularity to give relevant feedback concerning the timing of the software tasks. We demonstrate this method on the design of the digital part of an ADSL modem.

1 Introduction

The integration of ever larger systems on a chip (SOC) requires an integrated design environment, in which the interactions between the various components of the complete system can be designed and verified at an early stage in the design. Yet, the heterogeneity of the systems calls for a unified system design environment to integrate various design flows. This allows the different design teams to continue to use their familiar design flows, especially when the system includes large amounts of embedded software.

In this paper, we present our experience with building an executable system model for the digital part of an ADSL modem. This system is highly heterogeneous, since it includes both data-flow and control-flow oriented hardware components, most of which are in some degree programmable, and a large amount of embedded software. An important constraint has been to be compatible with the existing software development environment, because we wanted to reuse large portions of existing software.

In the design flow, as shown in Figure 1, the aim of a virtual prototype is to provide the software development team with a unambiguous model of the hardware before actual hardware is produced. In most industry design environments, hardware and software design teams operate separately, and use very different tools and methods. In most cases, after system design is finished, the hardware team will start its design. The software design will only start after the hardware design is finished, and testing of the software is only possible when a prototype is really built. However, by building a virtual prototype as side-product of the hardware design, testing of the software becomes possible much earlier in the design cycle, while hardware and software teams can still continue to use their familiar design methods.

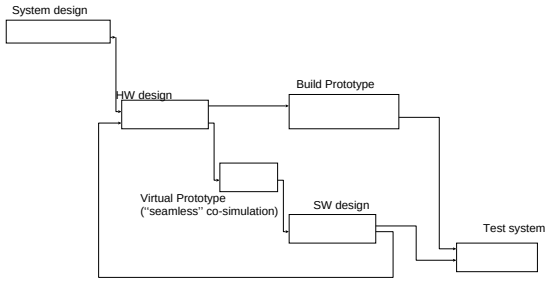

Figure 1: Virtual Prototype used for SW development

Several emerging codesign technologies, such as CoWare [5, 7] and Eagle [8], address the problem of constructing a virtual prototype of a system, where blocks described at different abstraction levels can be cosimulated. At the highest abstraction level, these tools have a very high simulation speed. At this level the tool is intended for the functional verification of the components and their interactions. However, timing behavior of the components can only be introduced on the cycle-true abstraction level, which means RT-level descriptions for hardware components or assembly code running on a instruction-set simulator for software components. These models, while providing the designer with accurate feedback on the timing of the system, result in low simulation speeds and require a low level of implementation detail, which makes them inconvenient to use in an early stage of the design process.

In our approach we overcome this limitation by specifying the timing behavior independently of the functional behavior of the components. A similar approach can be found in the PIA simulator [6]. Moreover the timing granularity can be chosen freely by the designer in function of the intended precision of the model in a particular stage of the design process. This will result in simulation speeds, comparable to the aforementioned untimed levels, but now timing information is included in the simulation. This offers the designer a flexible and fast model, that can be used e.g.

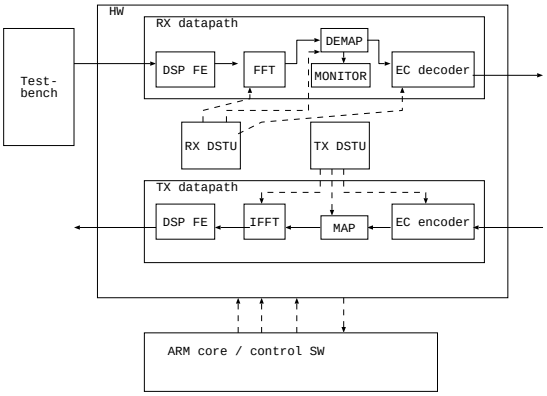

Figure 2: ADSL digital modem schematic

for evaluating alternative software implementations.

The remainder of this paper is organized as follows. Section 2 introduces the architecture of the ADSL modem, and proposes a design flow for codesign. Section 3 explains how the different hardware and software components are described, and cosimulated in model. Section 4 shows how this model can be used to analyze the timing constraints on the software. Finally Section 5 draws some conclusions.

2 Specification of the ADSL modem

We briefly present the architecture of the ADSL modem, and then motivate the design flow presented in the next sections.

2.1 ADSL System Architecture

To assess the viability of our approach, we have applied it to an industrial-strength ADSL modem design. This design implements the complete digital processing of the Discrete Multitone (DMT) modulation. This design consists of both hardware components and an important part of embedded software. The digital hardware is integrated into a single chip [1, 2] and consists of two parallel datapaths (for the receive and transmit parts), functionality for timing recovery and controllers. Additionally, an ARM core processor is integrated on the chip, which runs the software that executes the ADSL initialization sequence [4].

Figure 2 shows the architecture in more detail. The receive datapath consists of a chain of dedicated DSP processors : the DSP Front-End including decimation and time-domain equalization, the FFT module including a rotor and frequency domain equalization, the QAM demapper, the error and noise monitor, and a error-correction decoder. The transmit datapath has a similar structure. Dataflow communications are indicated by solid arrows in the diagram. Both datapaths have their own hardware timing controller (DSTU = DMT Symbol Timing Unit). This DSTU will activate the processors at the correct moments to do the appropriate processing of the DMT symbols. The control flow is indicated by dashed arrows in the diagram. All these hardware modules are implemented as programmable and configurable blocks, in order to execute different functions of the initialization procedure of the modem, and also in order to allow for changes in the ADSL standards.

Next to the hardware component, an ARM core processor runs the software that is responsible for programming

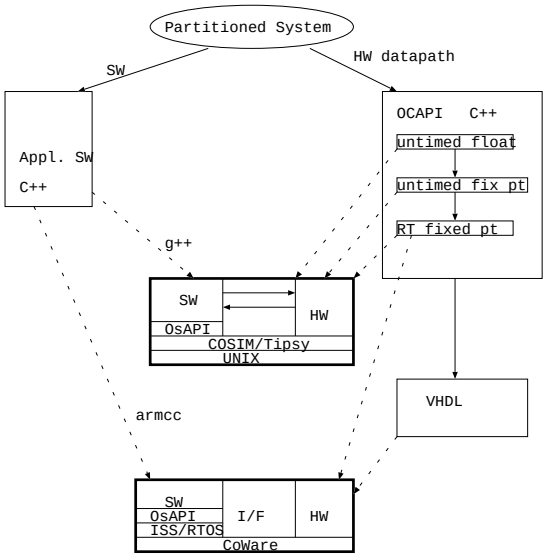

Figure 3: HW-SW codesign flow

and configuring the hardware. It has a control part, that configures the hardware to execute the different stages of the initialization sequence, and an algorithmic part to execute DSP functionality not implemented in hardware. The control part of the embedded software can be described as a reactive system, reacting on events generated by the monitor and other hardware modules, taking into account real-time constraints imposed by the ADSL standard.

For simulation purposes a testbench block is added, that generates the input sequences to the modem.

2.2 Design Flow

The inherent heterogeneity of the architecture requires the coexistence of different design flows. The proposed design flow starts after the system was partitioned in hardware and software modules. Every module will now follow a different design methodology. This modularity is the main incentive to use the object-oriented features of C++. By using a common cosimulation library, the different modules of the system can still be cosimulated at the various design stages. Figure 3 shows the two design flows, software to the left, and hardware datapaths to the right.

For the design of DSP datapaths a C++ based design flow, called OCAPI [9], is used. OCAPI is a dataflow-based object-oriented methodology, in which a Matlab-like untimed floating-point description is gradually refined into a cycle-true fixed-point description. Finally an automatic translation into synthesizable RT-level VHDL is provided.

The design of the embedded software will obviously use very different methodologies. Since in ADSL we are considering a very important software project (some 40k lines of C++ source), it is important that the software development team can continue to use its familiar development environment. Software development makes use of a object-oriented methodology for real-time systems Octopus [3].

In the next section we propose a cosimulation platform, where the different C++ levels of the OCAPI flow can be combined in a timed executable model on the workstation. This system model can be used by the software developer

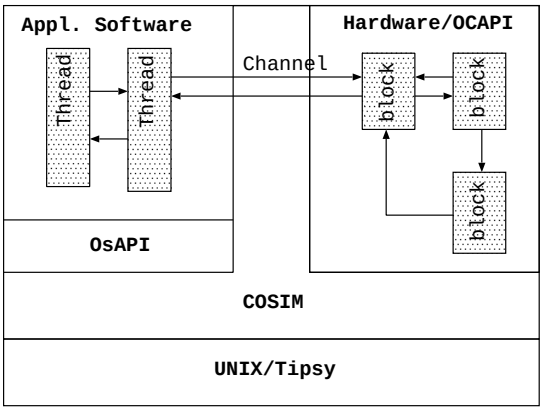

Figure 4: C++ based HW-SW Cosimulation using TIPSY

as a virtual prototype to test the software. It enables the software developer to develop and test the software on a UNIX workstation in the early stages of the design process. The cosimulation platform makes use of the libraries COSIM and TIPSY, explained in Section 3. Because the embedded software relies on a real-time operating system (RTOS), a library OsAPI is added, that implements the RTOS functionality on top of COSIM.

Since the system is not only characterized by the functional behavior of its components, but also by its real-time behavior, a model that combines these very different computational models, must specify the timing relationships between the components, in order to give meaningful feedback to the designer. It is important to notice that in this model functionality and timing are two independent features. This means that a model can be functionally very abstract, but still have very accurate precision on the timing of external interactions, or vice versa. Of course at lower abstraction levels functionality and timing aspects tend to be more closely linked. In this way a flexible environment is created, in which both hardware and software design teams can gradually refine various aspects of their implementation, and cosimulate the system.

An important aspect is, that the system level model described above, is integrated in a complete path towards implementation. For the hardware blocks it was already demonstrated how the OCAPI design flow offers a complete flow, that fits with VHDL implementation tools. Now we show how the CoWare tool can be used to implement the software on an embedded core processor, and implement the hardware-software interface automatically.

Both the RT-level C++ code and the VHDL code from OCAPI can be imported in the CoWare cosimulation. For the software part CoWare integrates an Instruction Set Simulator in the cycle-accurate cosimulation. The unmodified embedded software can be compiled for the core processor running a specific RTOS, by using an appropriate implementation of the OsAPI shielding library.

3 Cosimulation

In the subsequent sections, we explain how the hardware and the software have been described and cosimulated.

Figure 4 shows the cosimulation environment. The basis is a multithreading library TIPSY [10] running on the

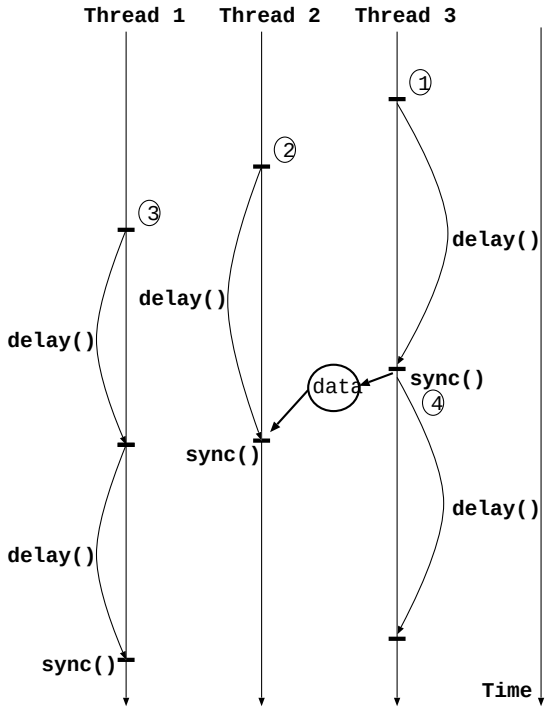

Figure 5: Thread Scheduling with TIPSY library

workstation, used for scheduling the threads. On top of TIPSY, the COSIM library provides classes for describing the components of the system, and the communication channels between components. A message-passing communication scheme is used as basic communication primitive here. The different hardware components are described in OCAPI, and communicate to the other modules via COSIM communication channels. For the embedded software an OsAPI library is used : this library offers most common RTOS features, implemented on top of the COSIM library. Because it shields the application code from the underlying RTOS, it allows cross-platform software development. This way the software developer can verify the actual application code in the virtual prototype. In the next sections each module will be further elaborated.

3.1 Scheduling threads with TIPSY Library

In the executable model of the system, there are multiple threads representing the different hardware and software blocks. In our approach we have chosen C++ as specification language, extended with a multithreading library TIPSY [10]. It schedules the threads, taking into account the simulated real time behavior of the threads. Below we give a short overview of the scheduling in TIPSY.

In TIPSY every thread keeps its local simulated time. The simulated time in a thread is augmented with the time step t by calling the library function *delay(t)*.

The scheduling of the threads is done based on the above mentioned simulated time, somewhat akin to the scheduling of processes in VHDL. The basic idea is that threads only need to be synchronized at the points where they communicate, i.e. when accessing shared data. This is expressed by the library function *sync()*, which has to be called before ev-

ery shared data access. The *sync()* function synchronizes the threads by invoking the thread scheduler; the scheduler always runs the thread with the lowest local time first. When the *sync()* function returns, all other threads are guaranteed to have a local time greater than or equal to the local time of the calling thread, guaranteeing the shared data to be accessed in the correct order.

TIPSY also supports the blocking of threads. In this case the execution of the thread is suspended until an event occurs in an other thread. When the other thread sends the event, the suspended thread resumes execution, and gets the same local time as the other thread. This allows for an elegant modeling of reactive software systems, without having to specify the timing estimations from the beginning, as will be demonstrated in section 3.5.

The scheduling is exemplified in figure 5. First *thread 3* is running, until it calls the *sync* function, before writing to shared data. At this point *thread 1* and *thread 2* have a smaller simulated time than *thread 3*, so they are now scheduled to run until they in turn call the *sync* function. Now *thread 3* can proceed, and write to *data*. This guarantees that when *thread 2* will read *data*, it is up-to-date.

3.2 Components and Communication

The components of the system are described at different abstraction levels and using different modeling techniques. Each component executes in a separate thread, and will use a common set of communication primitives to interact with the other components.

Message passing has been chosen as the basic communication primitive, because it can express the intertask communication primitives offered in RTOSes, and the dataflow queues used for describing the datapath. Basic message passing is sufficiently general, and more complicated communication schemes can be seen as a hierarchical composition of the basic message passing.

The communication primitive is implemented as a C++ class with methods for both blocking and non-blocking reads and writes. This class is implemented on top of the TIPSY synchronization functions (i.e. *sync()* and events) described above. In this way the designer can write the code without directly using the TIPSY calls. This has the advantage that the process is less error-prone, and that the designer can continue to use well-understood communication functions.

3.3 A Model for the Real-Time Operating System

In order to provide the software developer with a verification platform for the embedded software, the unmodified source code, as compiled on the embedded processor, must be used in the cosimulation. In most cases this means we should include support for a RTOS in the cosimulation.

The combined C++ models of the hardware blocks form a virtual prototype, on which the embedded software can be verified. Before we go to the actual software platform, i.c. an ARM core, we want to test the software on a UNIX platform. The software code is compiled onto the UNIX platform by using an interface library to RTOS, called OsAPI.

OsAPI is an intermediate library that shields the application code from the underlying RTOS, thus making the code independent of the specific RTOS. This library implements most common RTOS features on top of the COSIM/TIPSY library. This implies that the application code should not directly call RTOS-primitives, but the corresponding OsAPI calls. When the code is compiled for the core processor, the OsAPI library are translated into the actual RTOS calls.

3.4 OCAPI : Modeling the Hardware

The hardware of the modem consists of a receive and a transmit part. Both are built up of a datapath consisting of a chain of DSP blocks and of a real-time controller unit.

The OCAPI design flow [9], developed within IMEC, bridges the gap between system level specification (Matlab) and VHDL for data flow processors. It uses object-oriented C++ models on different abstraction levels : untimed and timed, floating point and fixed point. An automatic translation into VHDL is provided.

Since OCAPI is directed towards hardwired dataflow processors, extensions are needed to deal with programmable processors. This means to add the necessary interfaces towards the controller (core).

The time-granularity of the datapaths in the current model is at symbol rate (i.e. 4 kHz). This means that one execution cycle of the schedule implements 0.25 ms in real time. This is expressed by calling the TIPSY *delay(0.25)* function in the simulation loop of the datapath component.

Communications between hardware and software blocks use the communication primitives in the COSIM library, so that Tipsy can correctly schedule the threads.

In the OCAPI model, it is fairly easy to reorganize the same blocks to model the datapath on the sample rate (i.e. 2 MHz) level, using the same functional blocks. This will give a finer timing-accuracy of the model, at the cost of slower simulation, due to a higher number of context switches.

3.5 Modeling the Software

The application code for the embedded software describes a reactive system. The software responds to events (coming from hardware, from timers or from other control software) by executing threads that configure the hardware. The threads are implemented as tasks in the RTOS. Communications are mostly implemented as message queues in the RTOS with blocking reads. These are easily translated into the blocking COSIM communication primitives. Functionally the software threads are considered as running infinitely fast between two incoming events.

Figure 6 schematically represents how hardware and software models are synchronized. Thread HW1 is a periodical task, processing one DMT symbol per cycle. The *delay(sym)* statement introduces the given symbol rate. Sending event *ev1* activates the software thread *Th1*, which returns *ev2* to the hardware, and calls a RTOS-function to start a timer. The timer is modeled in the OsAPI library by a separate TIPSY thread, with a given delay, which returns the *ev3* to the software, and activates routine *Th2*. This could e.g. be used to count a number of symbols in software.

In the purely functional model of the software, the execution time is not taken into account. In the example the delays *delay(Th1)* and *delay(Th2)* are set equal to zero.

4 Analysis of the embedded software

In this section we explain how this model is used to verify the behavior of the embedded software under real-time constraints. The hard timing requirements in the ADSL system have two main origins : the dataflow sample rates in the RX

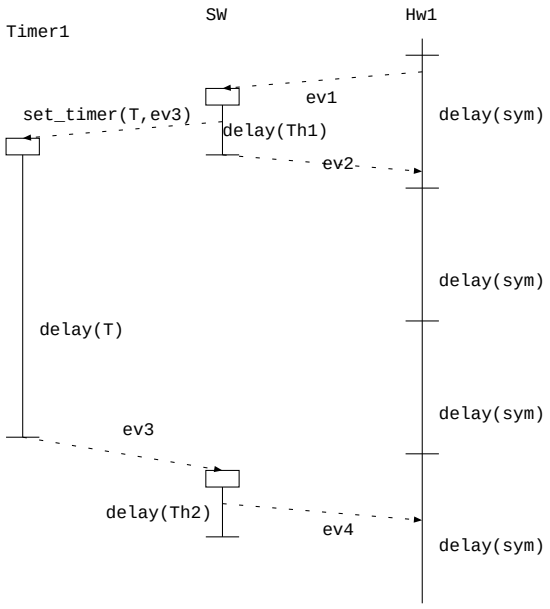

Figure 6: Timing analysis of SW tasks

and TX datapaths (e.g. the *delay(sym)* in Fig. 6), and timer operations in the embedded software (e.g. *delay(T)* in Fig. 6), which originate from timing constraints in the specification of the software. All these are, to the viewpoint of the software developer, functionally specified delays.

In a first step the software developer attempts to verify the functional correctness of the application software, without worrying about the performance aspects. This is accomplished in a very easy way, by running the code, as is, on the virtual prototype. Since in the software tasks the TIPSY 'delay' statements are left unspecified, these tasks are assumed to execute infinitely fast between two interthread communications. This means that time in the simulation progresses only due to the dataflow rate in the hardware datapaths and due to timers used in the RTOS.

Our model allows to verify the complete initialization sequence of the ADSL modem. The simulation time for 2 s in real-time was approx. 6 minutes on a HP K260 workstation. This is comparable to simulation speeds with an untimed C-model in e.g. CoWare, for the same level of functional detail. The datapath was described with a time step of 0.25 ms.

In a second step, the application software is annotated with estimated execution times, originating from profiling the code on the processor. These times are inserted in the code as TIPSY 'delay' statements (e.g. the *delay(Th1)* in Fig. 6). The behavior of the system could change if *delay(Th1)* becomes so large, that *ev2* arrives during the next symbol. The correctness of the system can again be tested by executing the model, and comparing the outputs.

A future extension is to provide in the TIPSY simulation environment accurate models of the RTOS, that enable a realistic simulation of the scheduling in the RTOS.

5 Conclusions

In this paper, we have shown how a complex digital system, such as the digital part of an ADSL modem, can be consistently described, using C++ as the common description language for all components of the system. We have shown how the timing aspects of the different components can already be incorporated in the model at a high abstraction level. Furthermore, by using the RTOS interface library, the unmodified application software code can be used in the virtual prototype. This makes our approach very interesting for embedded software developers, who can develop and test their final code on the virtual prototype, both with respect to functionality and timing.

Acknowledgements

This work was funded by the Flemish IWT in the Medea-xDSL project, and by the Flemish Government Impulse Program for Information Technology (IT-ISIS). The work was made possible thanks to the close collaboration with the system engineers in Alcatel Microelectronics.

References

[1] K. Adriaensen, F. Van Beylen, S. Van Hoogenbemt, H. Van De Weghe, J. De Laender, G. Verhenne, and P. Reusens. Single chip DMT-modem transceiver for ADSL. In *Proceedings Ninth Annual IEEE International ASIC Conference and Exhibit*, pages 123–126, 1996.

[2] Alcatel. MTK-20131 'DynaMiTe' Rate Adaptive Asymmetrical Digital Subscriber Line (ADSL) Modem Chipset, Product Brief . http://www.alcatel.com/mietec/datasht/xd245.htm.

[3] M. Awad, J. Kuusela, and J. Ziegler. *Object-Oriented Technology for Real-Time Systems : A Practical Approach Using OMT and Fusion.* Prentice Hall PTR, 1996. ISBN 0 13 227943 6.

[4] J. Bingham and F. Van der Putten. T1.413 issue 2. In *T1E1.4/98-007R3*, June 1998.

[5] I. Bolsens, H. De Man, B. Lin, K. Van Rompaey, S. Vercauteren, and D. Verkest. Hardware/Software Co-design of Digital Telecommunication Systems. *Proceedings of the IEEE*, 85(3):391–418, March 1997.

[6] K. Hines and G. Borriello. Dynamic Communication Models in Embedded System Co-Simulation. In *Proceedings of the 34th Design Automation Conference*, pages 395–400, June 1997.

[7] CoWare Inc. http://www.coware.com.

[8] Synopsys Inc. . http://www.synopsys.com/products/hwsw/hwsw.html.

[9] P. Schaumont, S. Vernalde, L. Rijnders, M. Engels, and I. Bolsens. A programming Environment for the Design of Complex High Speed ASICs. In *Proceedings of the 35th Design Automation Conference*, pages 315–320, June 1998.

[10] D. Verkest, J. Cockx, F. Potargent, G. de Jong, and H. De Man. On the use of C++ for system-on-chip design. IEEE Computer Society Workshop on VLSI, April 1999.