Operating System based Software Generation for Systems-on-Chip

Dirk Desmet
IMEC,Kapeldreef 75
3001 Leuven,Belgium
desmetd@imec.be

D. Verkest
IMEC,Kapeldreef 75
3001 Leuven,Belgium

Hugo De Man
IMEC,Kapeldreef 75
3001 Leuven,Belgium
also Professor at
Katholieke Univ. Leuven

ABSTRACT
In this paper we propose a system-level design environment, aimed at System-on-Chip (SOC) designs, including real-time embedded software. While many SOC modeling languages originate from hardware description languages, and thus tend to describe statical architectures, we observe that embedded software makes SOC designs essentially dynamic, and so a SOC modeling environment must include dynamic behavior. Such behavior is analogous to the services an Operating System offers in the software world, hence the term System-on-Chip Operating System (SoCOS).

1. INTRODUCTION
Various system-level design languages have been proposed in the recent past, to bridge the productivity gap between silicon integration capability and design productivity. There is a growing consensus in the design community that a general programming language like C++, already familiar to software designers, can be accommodated to be used also in hardware design. For this purpose libraries were added to the basic C++ language, rather than extending the language itself. (SystemC [7], CynApps [2], OCAPI [12]).

Most of these aforementioned libraries are based on typical hardware design requirements, and thus add to the C++ language concepts as signals, clocks, registers, parallel synchronous and asynchronous processes (concepts known from hardware description languages like VHDL and Verilog). However these libraries tend to offer little support for the embedded system designer, who wants to include the dynamic real-time behavior in the system model. This dynamic behavior comes into the system both by the embedded software and by reconfigurable hardware. Also the real-time aspects, with which the designer deals at the early phases of system design, are not addressed by the clocked synchronous processes of these libraries. In system-level design various components of the system will be described at very different timing abstractions, which can not be efficiently synchronized by a common system clock.

At the system level a design team will use many different computational models, for different components at the different stages of refinement. It is not our intention to elaborate on the library support needed for every one of these models. Equally important is a consistent way of instantiating and interconnecting these component models into an unambiguous system model. The required services are : run-time component instantiation and termination, run-time allocation and deallocation of resources (CPU/hardware, memory, communication channels) and access to these system resources. It can be observed that these are the type of services that are offered by (Real-time) Operating Systems in software development. Hence the idea to start system level design using a similar type of library (further called SoCOS).

SoCOS, as a system-level design environment, is used for modeling and simulating the system, analysis of the system and implementation through gradual refinement. An existing C++ based hardware design flow [12] is integrated in SoCOS. In this paper we will show more in detail how SoCOS can be used in embedded software design. The emphasis will be on the task concurrency issues. Memory management is an equally important design aspect in SoCOS, which is already extensively covered in other tools [14], and will not be further discussed here. It is important to notice that the major difference with an RTOS is, that SoCOS is used for simulation and analysis of the system, including its real-time behavior, on a workstation, while an RTOS is an implementation library running on the target platform.

Various design methodologies exist for embedded real-time software design. Many of these start from OMT/UML [11] (Rational [10], Octopus [3]). In our approach the OMT/UML design flow can be followed up to an intermediate level, where an executable version of the embedded software is made, so that the remaining refinement steps (task scheduling, resource allocation, inter-task communication) can be verified on the executable model. This offers a major advantage over a theoretical analysis. An automated synthesis also allows a quick evaluation of alternative implementations.

The remainder of this paper is organized as follows. Section 2 explains the basic principles of our system level modeling, and section 3 makes a comparison with other existing approaches. Section 4 elaborates on the approach for embedded software support. Section 5 shows how our approach was applied to an industrial design, i.c. an ADSL modem.

2. SOCOS PRINCIPLES
2.1 Communicating processes

DAC 2000, Los Angeles, California
(c) 2000 ACM 1-58113-188-7/00/0006..$5.00

A system model in SoCOS consists of communicating processes. Each process is executed in a separate thread in the OS. Processes can be statically or dynamically created. Statical threads are instantiated before the simulation scheduler starts running, and are destroyed only after the simulation has ended.

2.2 Communication

Processes have communication ports. Communication ports are declared as members of the process object, and can have type input, output or in/out. Ports are connected to communication channels. Also communication channels can be statically or dynamically created, and can have following attributes :

- event/data : whether the channel transports only an event (token is present or not), or also data
- data type : floating point, fixed point
- buffer size : (default = 1)
- blocking and non-blocking : blocking communications can cause the calling thread to wait.

2.3 Timing

Real-time behavior of the different processes is expressed by the simulated local time of each process. The simulated time is expressed by the statement *soc_delay(t)*, where t is the time increment. It is very important to notice the difference between this *soc_delay(t)* statement in SoCOS and a similar timer-call in a RTOS (something like *os_wake_after(t)*). The SoCOS delay is merely a modeling feature, that expresses the estimated time execution will take in the final implementation, and which SoCOS will use to schedule the processes in the system model. The scheduling mechanism is explained in more detail in [13]. A similar scheduling library can be found in [5].

2.4 How is it implemented

Since the support for analysis and refinement is an essential feature of the SoCOS library, an internal object-oriented datastructure is built at run-time, which stores the system configuration that is created using the API calls described above. This datastructure is used for simulation, but also supports analysis and synthesis (through code generation). Analysis routines operate on this data structure. Examples are communication profiling (how many accesses to communication channels, and at which time points), analysis of deadlocks (caused by blocking communications).

Since SoCOS is meant for simulation, analysis and refinement, the main focus is not on efficiency (code size, execution time), but rather on observability. In the final step towards the embedded software code, an efficient RTOS-based implementation is generated through code generation, where all overhead, caused by the SoCOS datastructure, is eliminated.

2.5 Computational models

Several computational models will coexist in a system level model. Library support has to be provided to offer these models to the system designer. This section demonstrates how SoCOS can be used as a basic layer to build such models.

```
class Soc_async_generator :
  public Soc_object {
  const int out_port;

  Soc_async_generator(const char* name,
       const Soc_channel& _out
  ) {
    int tid = soc_thread_create(
      this, Soc_async_generator::run, 0);
    out_port = soc_port_create(
      "out", tid, OUTPUT);
    soc_port_connect( out_port, _out);
    soc_thread_start(tid);
  }

  void Soc_async_generator::run( int arg)
  {
    int token;
    while(1) {
      token = calculate_token();
      soc_delay(T);       // est exec time
      soc_port_write(
        out_port, NOWAIT, token);
    }
  }
};
```

Figure 1: Example of an asynchronous model

2.5.1 Asynchronous models

The full strength of the SoCOS approach becomes clear in asynchronous models. Asynchronous models communicate with the rest of the system without any relationship to a system-wide clock. The execution time of an asynchronous model is determined by the *soc_delay(t)* statements it contains. In addition to *soc_delay*, an asynchronous model can have blocking input and output port accesses. This waiting time will add to the execution time of the process.

The code example in Figure 1 shows an asynchronous process that generates and infinite stream of output tokens. The time between the tokens is only determined by the *soc_delay* statement. In this simple example there are no blocking communications (NOWAIT), so this process is synchronized with its environment only through the *soc_delay* statement. The delay is essential here, since otherwise an endless loop would be created.

2.5.2 Reactive models

Reactive models contain processes that are triggered by an event on a communication channel. In that respect they are very similar to interrupt routines in software. These processes are dynamic by nature. SoCOS provides a system-call *soc_set_event_handler* that defines a reactive process.

Reactive processes can contain 'delay' statements, or not. When they do not, a zero-execution-time model is built. These will generally be used at the highest abstraction levels, where execution time is not yet relevant. Such models are also useful to model testbenches, probes, etc...

Reactive models can be used both for hardware and software blocks.

```
class Soc_react : public Soc_object {
  const int in_port;
  const int out_port;

  Soc_react(const char* name,
    const Soc_channel& _in,
    const Soc_channel& _out
  ) {
    int tid = soc_thread_create(
          this, Soc_react::run, 0);
    in_port = soc_port_create(
          "in", tid, INPUT);
    soc_port_connect( in_port, _in);
    out_port = soc_port_create(
          "out", tid, OUTPUT);
    soc_port_connect( out_port, _out);
    soc_set_event_handler(in_port, tid);
  }

  void Soc_react::run( int arg)
  {
     int in_token = soc_port_read(
          in_port, NOWAIT);
     int token = calculate_token(in_token);
     soc_delay(T);       // est exec time
     soc_port_write(
          out_port, NOWAIT, token);
  }
};
```

Figure 2: Example of a reactive model

In software blocks these correspond obviously to interrupt routines. In hardware models they can be used at the highest abstraction levels in combination with synchronous and asynchronous models.

Figure 2 shows a reactive model, where the *Soc_react::run()* function is attached as an event handler routine to *in* port, so it is triggered by an incoming event on the *in* port. It reacts after time *T* by producing an output. By setting *T* equal to zero (or omitting the statement), a zero-execution time model is obtained.

2.5.3 Synchronous models

Synchronous models communicate with the rest of the system on the edges of a clock. This concept is well known to hardware designers, and is used to build register-transfer level descriptions. It can as well be used at a higher level to make e.g. sample-rate dataflow models.

The execution time of a synchronous process is fully determined by the clock, and hence it cannot contain asynchronous *soc_delay* statements nor blocking input/output port accesses. The clock itself however is modeled as an asynchronous process. A master clock can be easily modeled in SoCOS by an asynchronous process, consisting of an infinite loop with a *soc_delay(clock_period)* All synchronous processes sensitive to this clock are triggered in this loop. Derived clocks can be modeled as synchronous processes sensitive to a master clock.

3. COMPARISON TO PREVIOUS WORK

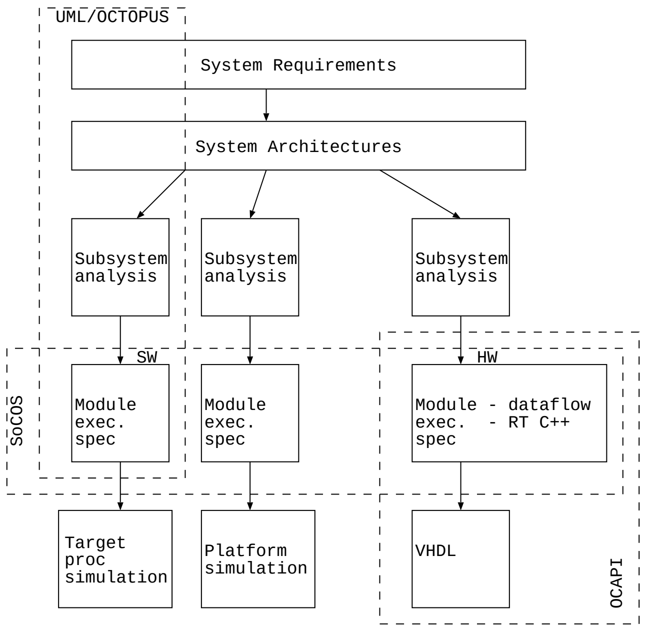

Figure 3: System level design flow

Other C++ based system-level models have been proposed recently. SystemC is one of the most comprehensive of these, and it covers also many of the models discussed above. The timing model used is however fundamentally different from SoCOS. In SystemC one system-wide master clock controls the simulation. As a consequence asynchronous processes have inherently a zero-execution time. This is very well suited to model relaxation-simulation in combination with a synchronous model. The SoCOS approach is however more general, and becomes especially useful when combining several models, described at very different abstraction levels.

Other system level models, like CoWare [6], also accommodate the mix of many different abstraction levels in one system model. The main difference here is however the dynamic character of the SoCOS model, as opposed to the statical (compile-time) model composition in CoWare. The dynamical character makes the model more suited to model the embedded software, as will be demonstrated in section 4. Also the simulated time notion is absent in CoWare, which makes models either clocked (synchronous) or untimed.

Another environment, where multiple computational models can be cosimulated, is Ptolemy [4, 9]. In SoCOS all models are synchronized by the common notion of the real-time behavior (simulated time). In Ptolemy not all models have to be time-aware, so it offers a more general way of synchronizing different domains. Dynamic process creation is also possible in Ptolemy. However Ptolemy is mainly intended as a simulation environment, while support for implementation through refinement is an essential part of the SoCOS approach.

4. DEVELOPMENT OF EMBEDDED SOFTWARE

4.1 Design flow

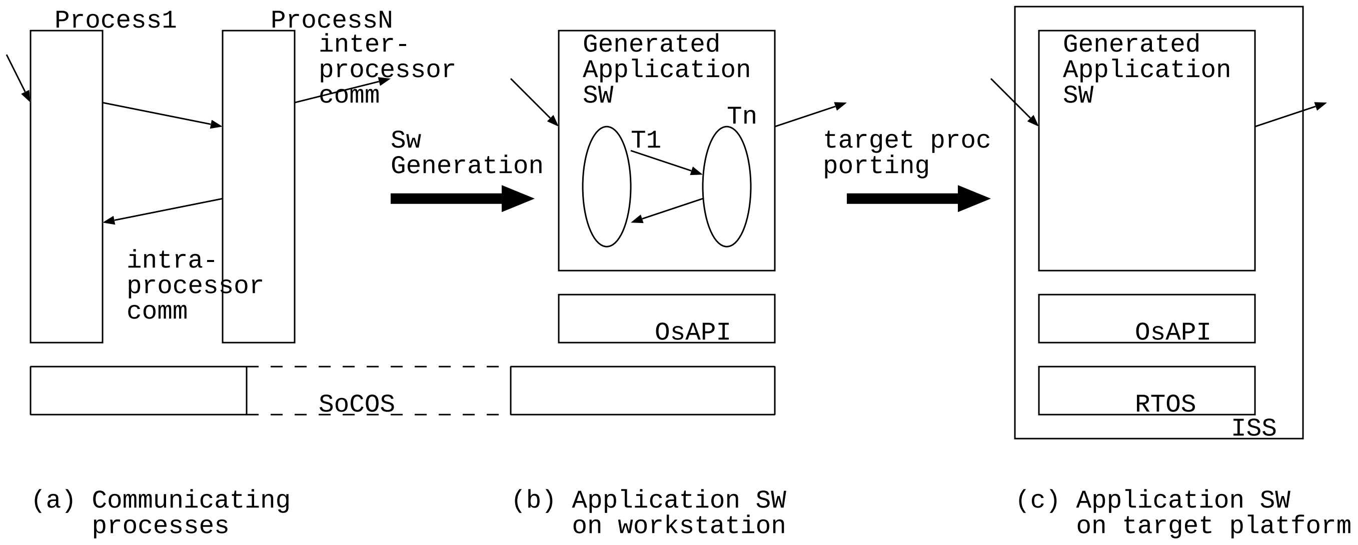

Figure 4: Embedded Sw Generation

This section explains the refinement of the software part of the system level model. Contrarily to the hardware implementation path, which has well defined steps in going from a high-level model, over a RT-level refinement, to synthesizable HDL (see e.g. [12]) the steps taken in real-time software implementation are less clearly defined.

Figure 3 shows the positioning of the SoCOS environment in a global system design flow. System design starts with a system requirements specification phase, followed by a system architecture phase. As a result, the system is divided in subsystems, and for every subsystem an analysis is made. The above mentioned design phases are covered in OMT/UML based methodologies for software design, such as OCTOPUS [3]. The next step is subsystem design for every subsystem. Here hardware and software design of course follow very different design flows, however for both, C++ based design environments exist, which result in executable specifications in SoCOS.

Our design flow is intended to be complimentary to existing software design flows. The starting point in SoCOS is the most abstract, yet executable, level of description, which is called further uncommitted parallel processes. Uncommitted means: having maximum parallelism (all threads run virtually in parallel on different processors), having zero execution time, and unlimited communication resources (i.e. a dedicated channel for every communication). This uncommitted parallel processes model is the closest we can get using a sequential language like C++ to a purely functional specification. Although functional languages, such as ML [8], might be ideal for modeling parallel processes at a purely functional level, we believe that the C++ model is preferable due to the large acceptance of the language in the design community.

In the implementation refinement process the goal is to arrive at a fully committed description, which means :

- execution time: all processes are annotated with realistic execution times. These will have to be extracted from profiling the code on the target processor. It has to be remarked that not giving realistic execution times will not prevent the use of our design flow to generate a software implementation, but will of course make it impossible to evaluate the scheduling upfront.
- processor scheduling: processes are allocated to processors, and priorities are defined.
- every inter-process communication is allocated to a communication resource. Here we must make a distinction between inter-processor and intra-processor communications. Intra-processor communications will be implemented in the design flow discussed below towards a RTOS-based implementation. Inter-processor communications require hardware-software interfaces.

No particular order in the refinement is imposed. After every refinement step, the result will have to be verified by simulation. Profiling results can be compared to previous executions.

4.2 Embedded Software Generation

For every software processor, application source code is generated from the fully committed communicating process model. In general, application software will be based on an RTOS. In this step every SoCOS system call, present in the communicating process model of Figure 4.a, is replaced by a corresponding piece of code based on the RTOS library in Figure 4.b. In this translation the behavior is guaranteed to be kept consistent, while the implementation overhead of SoCOS is replaced by an efficient RTOS implementation.

4.3 Embedded Software Co-simulation

In our environment the final embedded software code is cosimulated in the system model using an OSAPI library (see Fig. 4). The OSAPI library provides the functionality of a typical RTOS to the application code. This functionality is implemented on top of the underlying SoCOS simulation environment. This approach can be used, either to cosimulate existing software code, or generated code, with the rest of the system model. This simulation will run

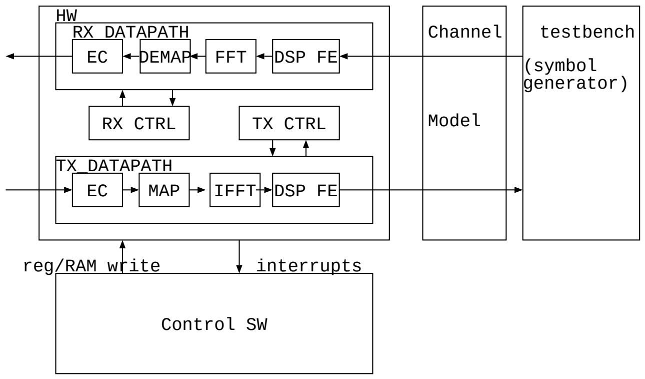

Figure 5: ADSL Application

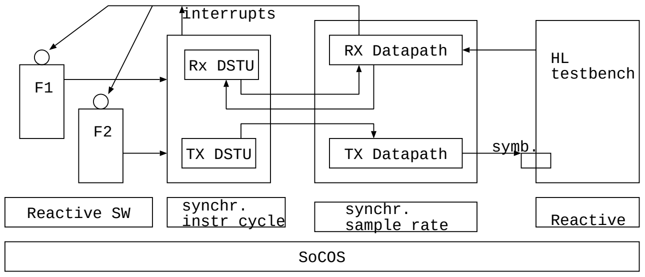

Figure 6: System level model of ADSL

orders of magnitude faster than a simulation on an Instruction Set Simulator (at the cost of less timing accuracy) and is useful as a last verification of the embedded software functionality and as a reference before transferring the software to the target processor.

Because the timing accuracy is always limited by execution-time estimates, all assumptions must still be verified on the target platform (Fig 4.c). However a quick design path exists to generate alternative implementations from the same communicating process level description.

5. DESIGN EXAMPLE

The SoCOS design library was successfully applied to the design of an ADSL modem. This design (see Fig. 5) consists of two datapaths, each datapath controlled by a real-time programmable controller, and embedded control software (running on an ARM core) that controls the initialization and retraining of the modem. More details about the design can be found in [1].

5.1 System level model of the modem

The system level model of the ADSL modem, shown in Fig. 6, is built using SoCOS as a combination of various computational models each most appropriate for a specific component of the system. Both DSP datapaths were modeled as sample rate data flow using synchronous processes, controlled by a sample rate clock (0.5 - 2 MHz). The real-time control units (DSTU) were modeled as simple instruction-set simulators, running as synchronous processes at a instruction-cycle clock (here 8 MHz). The high-level testbench (which implements the behavior of the other modem) is modeled as a reactive process, triggered by a token at its input port.

The various hardware blocks have outputs, sending interrupt events to the control software (e.g. the RX datapath will produce a *symbol_detected* event when a given symbol was received). At this level the software is described as a number of reactive processes, triggered by the interrupt events. While the hardware components described above are statically created (the configuration is unchanged during all the simulation), in the software the dynamical features of SoCOS are used. Suppose the thread F1 in Fig. 6 corresponds to the software routine handling the detection of the ACK-symbol. Upon successful detection of the ACK, the system is reconfigured to detect REVERB, and a new process is connected to the *symbol_detected* interrupt event.

5.2 Refinement of the SW model

Up to this point the SW was modeled without any resource constraints, i.e. every thread runs in a parallel process with zero-execution time. This section explains the steps taken to refine this uncommitted model towards a software implementation using an RTOS on an embedded core processor.

First estimated execution times are added to the model for time-consuming calculations, e.g. the calculation of the equalizer coefficients in thread detectREVERB. Since it was decided to run all control software on one ARM core, all SW threads will run on the same processor. At this point an implementation strategy for the interrupt handling must be chosen. In our case (see Fig. 7) a main routine was added, containing an infinite loop that waits for an incoming event, and dispatches it according to the origin of the interrupt.

Next, implementation decisions have to be taken concerning the task concurrency. Every event on a communication channel, to which a reactive task is attached as an event handler, can be implemented either synchronously or asynchronously. In this context synchronous has the meaning of a function call, and the body of the called process is executed in the calling process; while asynchronous means the called process is executed in a separate thread. At this point the model can be simulated to verify the decisions made so far. In our case the second detect-routine was made asynchronous, because the coefficient calculation is very time consuming, and would block the handling of other interrupts if it was merged into the main process. Figure 7 shows that the software now consists of 2 processes.

In a next step more detailed implementation decisions have to be taken concerning an RTOS based implementation (Fig. 8). ISRs (Interrupt Service Routines) are added to the model, which translate the incoming hardware interrupt into an RTOS event. Next all inter-process communications (within the same processor) are allocated to RTOS resources (e.g. message queues, semaphores, shared memory). These are added as attributes to the ports of the processes and to the channels. Again at this point the simulation can be run to verify all refinements.

5.3 Embedded Software Generation

The fully committed software model described above contains all information to generate the RTOS-based application source code. Basically every SoCOS call is replaced by a library-based implementation, based on the attributes given in the previous steps, using RTOS system calls.

The generated software can again be simulated on the workstation, using an OSAPI library, implemented on top of SoCOS. The OS-

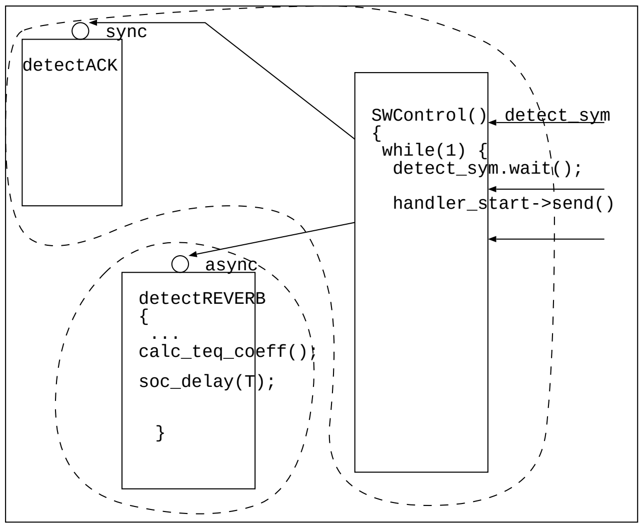

Figure 7: SW after processor and process allocation

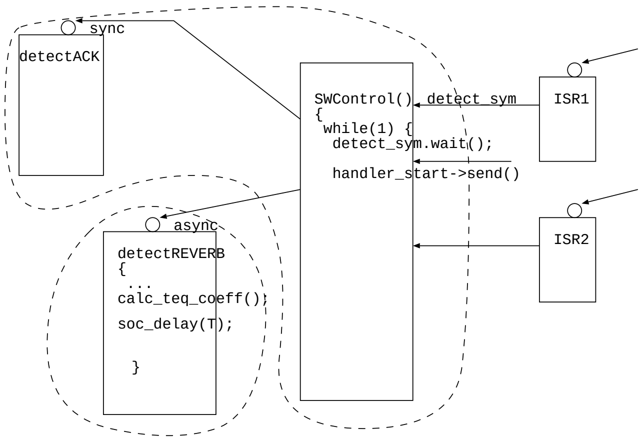

Figure 8: Fully refined SW

API library implements all common RTOS system calls. The same generated application software can be compiled for the target processor, using an OSAPI version for a specific RTOS (OSAPI acts as a simple translation between OSAPI calls and RTOS calls).

5.4 Results

The complete ADSL model consists of the sample-rate dataflow model of the datapaths (8700 lines of C++ code) and the simple instruction set simulator for the real-time controllers (980 lines of C++ code). The embedded software is described at the communicating processes level by approx. 5000 lines of code, which expand to 22640 lines of C++ code after refinement. The simulation of the entire model on a 366 MHz Pentium with Linux takes approx. 25 minutes for simulating 10s of real-time behavior.

The SoCOS library was extended with Tcl/Tk Graphical User Interface, through which the simulation is controlled. Design object like registers, RAMs, communication channels, etc. are observable in the GUI.

6. CONCLUSION

This paper proposed a C++ library for system level design, that offers the designer with services analogous to an operating system in software design. Software can be functionally tested in combination with hardware. Real-time aspects can be gradually introduced, without rewriting the code (by adding implementation attributes). We have elaborated the refinement steps that lead from the system level model to embedded real-time software source code. This approach was demonstrated on a digital DMT ADSL modem.

7. 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-IRMUT). The work was made possible thanks to the close collaboration with the system engineers in Alcatel Microelectronics.

8. 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 (Cat. No.96TH8186). IEEE, New York, NY, USA; 1996; xvii+326 pp. p.123-6*, 1996.

[2] C. . C. Applications. http://www.cynapps.com.

[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. B. et al. PTOLEMY: A framework for simulating and prototyping heterogeneous systems. *International Journal on Computer Simulation*, January 1994.

[5] 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.

[6] C. Inc. http://www.coware.com.

[7] T. O. S. Initiative. http://www.systemc.org.

[8] L. C. Paulson. *ML for the working programmer.* Cambridge University Press, 1991.

[9] PtolemyII. http://ptolemy.eecs.berkeley.edu/ptolemyII.

[10] Rational. http://www.rational.com/uml/index.html.

[11] J. Rumbaugh. Omt : The development process. *Journal of Object Oriented Programming,* May 1995.

[12] 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.

[13] D. Verkest, J. Cockx, F. Potargent, H. De Man, and G. de Jong. On the use of C++ for system-on-chip design. In *Proceedings of the IEEE Workshop on VLSI*, pages 42–47. Orlando,Florida, April 1999.

[14] D. Verkest, J. da Silva, C. Ykman, K. Croes, M. Miranda, S. Wuytack, G. de Jong, F. Catthoor, and H. De Man. Matisse: A system-on-chip design methodology emphasizing dynamic memory management. *Journal of VLSI Signal Processing*, 21(3):277–291, July 1999.