In *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen*.
by Christoph Scholl, Sefan Disch (Eds.). GI/ITG/GMM-Workshop, Shaker Verlag, Aachen, March 3-5, 2008

Semi-Automatic Generation of mixed Hardware/Software Prototypes from Simulink Models

Martin Streubühr, Michael Jäntsch, Christian Haubelt, Jürgen Teich
Hardware/Software Co–Design, Department of Computer Science
University of Erlangen–Nuremberg, Germany
{streubuehr, jaentsch, haubelt, teich}@codesign.informatik.uni-erlangen.de

Axel Schneider
Alcatel-Lucent Deutschland AG, Nuremberg
aschneider@alcatel-lucent.com

Abstract. *We present a semi-automatic design flow from Simulink models to prototypes of mixed hardware/software implementations of these models. Our work consists of three key contributions: (1) transformation of a functional model given in MATLAB/Simulink to the well-defined synchronous reactive model of computation (SR MoC), (2) an automatic SystemC code generation from Simulink models using the SR MoC and (3) a semi-automatic prototype generator for heterogeneous hardware/software systems implementing the* chaotic iteration *scheduling for SR models.*

1 Introduction

Simulink [17] is a well established modeling approach in many application domains and is today the de-facto standard for model-based design in the automotive area. Simulink's dominant role can be seen in the area of algorithm optimization. Furthermore, using Real Time Workshop [16], it is possible to automatically generate C-Code for single processor systems or VHDL RTL code for hardware implementations. However, there is currently neither support to generate mixed hardware/software systems from Simulink models nor to model architectural effects like resource contention in Simulink.

On the other hand, mixed hardware/software implementations become prominent in nearly all domains of embedded systems. Here, SystemC [9] can be seen today as one of the best suited languages to model architectural effects such as cost, performance, or power consumption as well as allowing a seamless design flow to hardware/software implementations.

In this paper, we will combine the strengths of Simulink and SystemC, and present a semi-automatic design flow from Simulink to hardware/software implementations. An overview of this design flow is shown in Figure 1. The contributions are (1) mapping of discrete Simulink models to the well known synchronous reactive model of computation (SR MoC) [7], (2) an automatic SystemC code generator from Simulink models using the results from (1), and (3) a semi-automatic prototype generator for mixed hardware/software systems by implementing the so called *chaotic iteration* scheduling for SR systems.

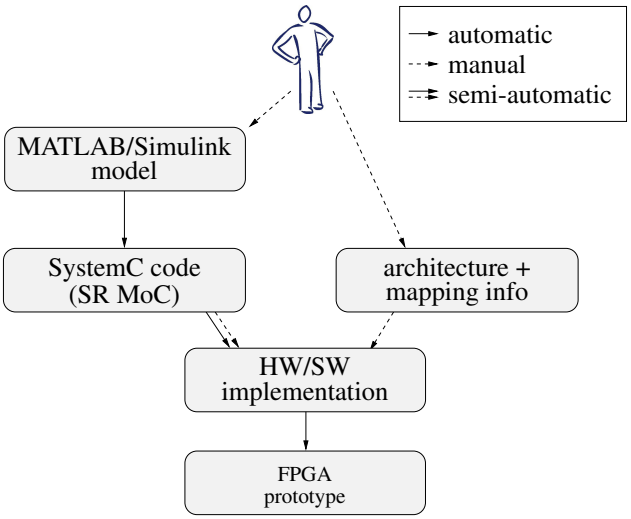

Figure 1: Functional Simulink models are mapped to SystemC models using the SR model of computation. Using architecture and mapping information, the SR model is transformed to a mixed hardware/software prototype implemented on an FPGA.

The paper is structured as follows: An overview on related work in given in Section 2. Then, we summarize some details on MATLAB/Simulink in Section 3 and we present the execution semantics for the synchronous reactive model of computation in Section 4. In Section 5, we present the mapping of Simulink models to SR models, and we show how to implement mixed hardware/software designs from SR models in Section 6.

2 Related Work

Huang et al. [12] presented a case study of a Simulink-based MPSoC design flow. An initial functional Simulink model is composed to a *combined algorithm and architecture model* (CAAM), which is derived by hand using hierarchical grouping of functional blocks. Here, the hierarchical structure represents the partitioning into CPU subsystems and thread subsystems as well as inter- and intra subsystem communication blocks. Afterwards, the Simulink CAAM model may be implemented at different abstraction levels (Virtual Architecture, Transaction-accurate Model and Virtual Prototype) using a *Multithreaded Code generator*. A very similar approach is reported by Atat and Zergainoh [2]. The functional Simulink model is refined to a Simulink transactional model. Using tool support, the transactional model is transformed to the more detailed *Macroarchitecture Model* or the even more detailed *Microarchitecture Model*. Simulation allows for verification at each abstraction level. In contrast to our work, partitioning into mixed hardware/software systems is done within the Simulink model, by grouping functional blocks and inserting special interface blocks. Our proposed design flow maps Simulink models to the well-defined SR MoC. The mixed hardware/software implementation and a self-scheduling scheme are derived from that model of computation.

Caspi et al. [6] proposed a layered approach quite similar to our own, while translating Simulink models to *SCADE/Lustre* [8], a synchronous language, and afterwards implementing it on a time triggered architecture. Their focus lies on designing safety critical software for the automotive and avionic industry. This is achieved by using *SCADE/Lustre*, which features a level-A certified automatic software code generator. Our approach also uses a synchronous in-

termediate representation for Simulink models in order to create prototype implementations. In contrast, our design flow is not limited to distributed software architecture, but supports mixed hardware/software implementations, too.

Baleani et al. [3] presented formal transformations between MATLAB/Simulink and ASCET using the synchronous reactive model of computation as intermediate layer. In particular, Belani et al. presented transformations for ASCET to SR, SR to ASCET, as well as, SR to Simulink and Simulink to SR. Our approach also uses a similar transformation from Simulink to the well-defined SR MoC, in order to derive mixed hardware/software prototype implementations from SR and thus from Simulink.

An approach to the transformation of Simulink models to the *System Property Intervals* language is reported by Jersak et al. [13]. This approach transforms the time-driven model of computation into a data-driven model, by introducing virtual FIFO-queues for synchronization between different rates. We propose to use the synchronous reactive model of computation [7] which is more suited to represent the reactive nature of Simulink models. Beyond, our design flow supports mixed hardware/software prototype implementations.

Stefanov et al. [15] presented a design flow for implementing MATLAB code on a target platform utilizing a microprocessor and an FPGA. A nested loop program specified in MATLAB code is transformed to a *Kahn Process Network*, which can be mapped to the target platform. Our approach also targets mixed hardware/software implementations, but uses Simulink models instead of nested loop MATLAB programs as input for the design flow.

Another platform-based design flow for optimized hardware/software FPGA implementation is reported in [11]. A SystemC-based language for modeling data-flow is used as input for automatic design space exploration. Optimized results from exploration may be used to create FPGA prototypes automatically. The results of this work could be extended in order to set up a similar platform-based design flow using Simulink models or SR models as input.

In our work, we focus on (1) mapping a Simulink functional model to the well-defined SR MoC, (2) automatically generating SystemC code from the Simulink models using an SR MoC implemented in SystemC and (3) generating hardware/software prototypes. Thus, our approach paves the way for an automated design flow without need for error-prone rewriting or redesigning of the initial functional model.

3 MATLAB/Simulink

MATLAB/Simulink is a toolbox for MATLAB, developed by MathWorks [17], that can be used to graphically model and simulate hierarchical systems. Another toolbox for MATLAB is the Real Time Workshop, which offers a code generator, to generate highly optimized C-Code from a Simulink model. MATLAB/Simulink provides an interactive block-diagram based graphical environment. Libraries offer a broad variety of predefined blocks that can be used along with user-defined functions. These can either be described by embedded MATLAB code or as so called S-Functions [17]. Data-flow between the blocks is realized using directed connections represented by arrows. Blocks are evaluated at a certain rate, the sample rate, which can be set globally or for each block individually.

Figure 2 presents an example of a Simulink model consisting of seven blocks. We assume each block has the same sample rate in this example. Hence, each block can be evaluated one after another at each sample time. Here the MATLAB/Simulink simulation would compute the output value of the SRC block from the value of the environmental input SRC-IN. Only when the SRC block has been evaluated completely, the computation of block T_1 and afterwards of the block Substract starts. In this case the feedback loop is broken by the Memory block, because

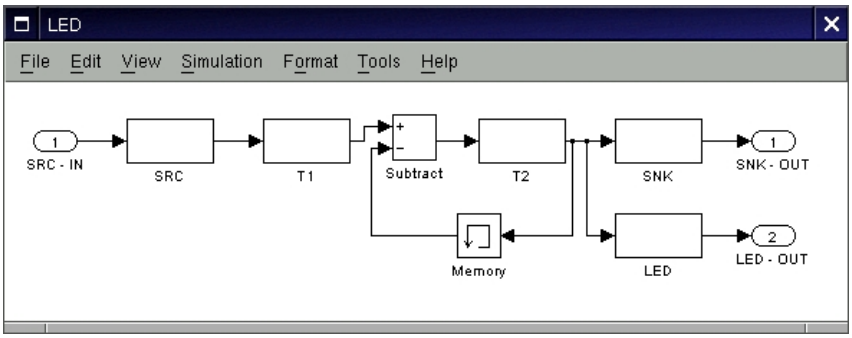

Figure 2: An example of a Simulink model. A *SRC* block receives data from the environment. The blocks T_1 and T_2 prepare and process the data from the *SRC* block, respectively. Blocks *SNK* and *LED* edit the results from block T_2 for different kinds of outputs to the environment.

the Substract block can use the output of the Memory block for computation, before the input of the Memory block is known. If the block Subtract has been executed, the evaluation of block T_2 starts. Note that the SNK and the LED block as well as the state update for the Memory block depend on the results from block T_2 and have no cross dependencies. Therefore, they could be evaluated at the same time in the Simulink model, right after block T_2.

MATLAB/Simulink is often used to simulate and verify systems in early stages of a design, because it allows for rapid design, simulation and generation of real-time C-Code. The option of hierarchical model development simplifies the design and reuse of systems.

4 The Synchronous Reactive Model

Modeling of reactive systems often relies on the *synchrony hypothesis* [4], with the ideal assumption of a system producing outputs synchronously to changes of the inputs. In order to fulfill this assumption, execution of a reactive system needs to be infinitely fast. Several examples of synchronous languages exist, such as *Esterel* [5], *LUSTRE* [10], *Quartz* [14] or *SR* [7].

The *synchronous reactive* (SR) model of computation is presented in [7]. There, a block-diagram language for describing synchronous software systems is used. An SR system is composed of blocks as depicted in Figure 3. Each block owns a set of inputs I and a set of outputs O. Channels interconnect an output with possibly several inputs using multicast semantics, depicted by arrows. Dangling channels represent interactions with the environment. Blocks can have a state, while outputs are functions of this state and the inputs or combinatorial functions of the inputs. SR uses a logical model of time by means of a sequence of *instants*. In SR, a channel can either contain a certain value or be *undefined*, while *undefined* is represented by the $\perp$ value.

The execution of an SR system is now given by the computation of a series of instants. Execution of an instant requires computing the fixed point for all channels. Blocks may be executed several times per instant if the SR system contains combinatorial feedback loops. Otherwise fixed point calculation reduces to causal computation, e. g., block execution in topological order. In the context of instantaneous feedback, Edwards suggest the *chaotic iteration* scheduling [7] for synchronous models: All channel connected to outputs are set to *undefined*, only environmental input channels may carry a certain value. Initially a block execution order is selected randomly. The entire SR system is executed repeatedly using this order, until the fixed

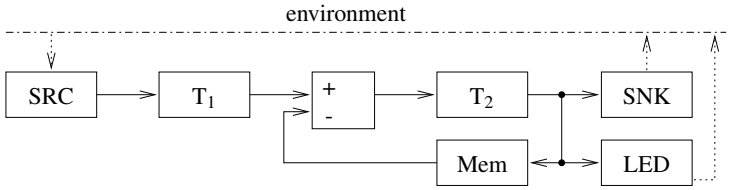

Figure 3: An example of a synchronous reactive system. Due to the *synchrony hypothesis*, the SR system reacts infinitely fast to changes of the inputs from the environment.

point is reached. A fixed point is found, when the next execution of the system leads to the same result as the prior execution. Edwards also suggest optimized static schedules especially for the simulation of SR models, i. e., single processor schedules. Note that one possible fixed point may be to evaluate each output to *undefined*, what may happen in case of feedback loops, e. g., a feedback from the output to the input of a block computing the identity function results in the obvious solution *undefined* for the output.

The fact that each block may execute several times within one instant leads to two different types of blocks: *strict* blocks and *non-strict* blocks.[1] *Strict* blocks can compute their outputs only if all of the inputs are defined. *Non-strict blocks* may compute parts of the outputs if only parts of the inputs are defined. A typical example for a *non-strict* block is the Boolean AND function. If any of the inputs is *false*, the output evaluates to *false*, as well. This kind of "short-circuit" evaluation may be used to break combinatorial feedback loops, where otherwise each output has to be evaluated to *undefined* as not enough inputs are defined. A Moore finite state machine is a *non-strict* block, as the outputs depend only on the actual state and can be computed independently from the inputs. Note that state updates are only allowed once per instant for *non-strict* actors. Therefore, we divide the functionality of *non-strict* blocks into two functions `go()` and `tick()` as suggested by Edwards [7]. The `go()` function is used to compute outputs, while state updates are implemented in the `tick()` function. As a consequence, the `go()` function can be executed several times per instant in order to generate *defined* outputs, while the `tick()` function is called at the end of an instant, carrying out the internal state update.

5 Mapping Simulink to SystemC

By transforming Simulink models to SystemC code we combine the strength of both languages. MATLAB/Simulink is a state-of-the-art tool for functional modeling, simulation, testing, and algorithm optimization. SystemC supports modeling and simulation of complex heterogenous embedded systems at different abstraction levels. Beyond, SystemC is often used to set up test-benches and to couple different simulation environments for co-simulation.

The proposed method to generate SystemC code from Simulink models is based on the Real-Time-Workshop (RTW) [16] toolbox. RTW is able to generate highly optimized C code from Simulink models, while code generation can be customized by the Target Language Compiler (TLC) using so called .tlc files (refer to [16]). Real Time Workshop works by creating a .rtw file, which is an XML type description of the entire model. This file contains all information about the system, including blocks, parameters and connections and is used to generate C code. Note that we refer to Simulink blocks and SystemC modules as blocks and modules, respectively.

[1] *Chaotic iteration* works identical for both, *strict* and *non-strict* blocks.

5.1 Code generation

We created a framework that automatically generates a SystemC module for each Simulink block, while using the RTW code generator to define the functionality. By mainly using Target Language Compiler directives, we can use most of the inherent flexibility, e. g., the automatic generation of code for user-specific blocks.

The SystemC modules created by our customized Target Language Compiler have a static template which is then augmented with functional C code, generated by RTW. Each module declares input and output ports and uses constructor code for initialization according to the given block's functionality. For discrete Simulink models, blocks may have internal discrete states (DSTATES), e. g., a memory block is able to store the last input. Blocks with discrete states may be used to break combinatorial feedback loops that otherwise could not be supported for C code generation. In this case, two functions `update()` and `output()` are created by the original RTW. By calling first the `output()` function writing the output, and later, if sufficient data is available, the `update()` function, feedback-loops are broken. Obviously, this behavior represents a Moore finite state machine, where outputs only depends on the actual state and may be produced independently from inputs. We cope with discrete state blocks by using *non-strict* modules as described in Section 4. Using the two functions `go()` and `tick()`, the *non-strict* behavior allows modeling modules, representing such Moore machines. Here, `go()` and `tick()` functions represent the `output()` and `update()` functions used for writing outputs depending on the internal state and updating the internal state depending on the inputs, respectively. Note that the `go()` function may be called several times per instant, but is guaranteed to produce the very same output, because state update occurs only at end of an instant by calling the `tick()` function once (monotonic behavior). Therefore, the input and the output signals are decoupled and feedback loops are executed in the same manner as in the original Simulink model.

5.2 Supported Simulink features

Currently, we only support *discrete* Simulink models. Therefore, we only support the fixed step, discrete state solver, while Simulink offers a couple of different types of solvers for simulation. In particular, we do not support any *continuous blocks* in the model. Using continuous solvers would make the model-of-computation ambiguous and therefore not appropriate for designing mixed hardware/software systems. Moreover, we do not support multidimensional or complex signals.

All blocks that are stateless, e. g. mathematical operations or any type of source, as well as all blocks from the discrete library, e. g. a *Discrete Transfer Fcn*, will translate to a working SystemC model. We support user-defined functions representing analytical functions (e. g. $y = x^2 + sinx$) by means of the simple *Fcn* block. Other kinds of user-defined functions are not implemented yet.

In Simulink, it is possible to have blocks running at different sample rates in the same model. For code generation using RTW, they must be separated by so called *Rate Transitions*. A *Rate Transition* in MATLAB/Simulink holds the signal from the predecessor block until it changes, similar to a register. So, at whatever rate the successor is running, it always gets a valid input signal. Naturally, the SR model of computation does not support execution at different rates, so we assume the entire SR model is executed at a minimum rate given by the greatest common divisor of all rates. To achieve the individual rate of blocks, we use counters to guard the execution of the functionality inside modules.

6 Generation of Mixed Hardware/Software Prototypes

The generation of an FPGA-based prototype allows for early estimations of mixed hardware/-software implementations, in terms of power, performance, etc. Yet, the mapping steps described in the following are not automated. But they provide a general approach to create mixed hardware/software implementations from SR models, not limited to a certain application. By automating these mapping steps, we could easily set up an automatic platform-based design flow. Furthermore, optimized commercial high-level tools could be integrated to automate the implementation of individual SystemC modules.

The implementation of SR models as mixed hardware/software systems requires the implementation of each block as a hardware module or a software function. Additionally, channels between blocks need to be mapped accordingly to a communication infrastructure supporting mixed hardware/software designs. In particular, we distinguish four different kinds of channel implementations to exchange data between blocks in hardware and software: We have channels for: (1) software to software, (2) hardware to hardware, (3) hardware to software and (4) software to hardware communication. As all channels in an SR model may carry the *undefined* symbol $\perp$, there is the need to represent this *undefined* symbol for all kinds of channel implementations. For this purpose, we introduce an additional Boolean signal associated with each channel, indicating if the value stored in this channel is *defined* or *undefined*. Thus, a software to software channel consists of a pair of variables, the data and the control variable. The hardware implementation of a channel needs a vector of signals, representing the Boolean control value and the data value. The hardware/software interface implementation combines both kinds of channel implementations via memory mapped registers assigned to the processor bus.

One way to generate a mixed hardware/software implementation from a given synchronous specification can be achieved by implementing clusters of the specification as hardware accelerators wrapped by corresponding software drivers. For this purpose, a synchronous system has to meet the following requirements: (1) A cluster evocation can be executed in a limited number of clock cycles. (2) The worst case execution time (*WCET*), i. e., the number of clock cycles until all outputs of the cluster are computed and set to *defined*, is known a priori or can be computed at compile-time. (3) Each cluster behaves *strict*, requiring all inputs to be defined before any output can be computed. (4) All outputs of the hardware cluster are guaranteed to become *defined* or *undefined* independently from the value of these cluster's inputs.[2] If these requirements are fulfilled, each evocation of a hardware cluster can be treated like a function called from a software wrapper, by writing all inputs to the cluster, awaiting the *WCET* and reading the results, subsequently.[3] In this case a simple single processor software scheduling approach can be used to schedule the entire system.

A more general approach to the implementation of a synchronous system as a mixed hardware/software system uses the *chaotic iteration* scheduling. This kind of scheduling can be combined with a high-level synthesis approach for single blocks easily. Beside the enhanced flexibility, *chaotic iteration* scheduling may decrease execution times in the average case in contrast to WCET-based scheduling. Such a mixed hardware/software system derived from the SR application given in Figure 3 is depicted in Figure 4. This implementation is more flexible and allows for concurrent execution of hardware blocks, in parallel to the software blocks. But this flexibility requires more control overhead, as hardware blocks can compute in parallel and need to be synchronized. Determining the end of an *instant*, i. e., the entire system reaction,

[2] Inputs that are guaranteed to be *undefined* don't need any computation.

[3] In the extreme case that the *WCET* is smaller than the latency between writing the last input and reading the first output via the bus, no additional waiting operations in software are required.

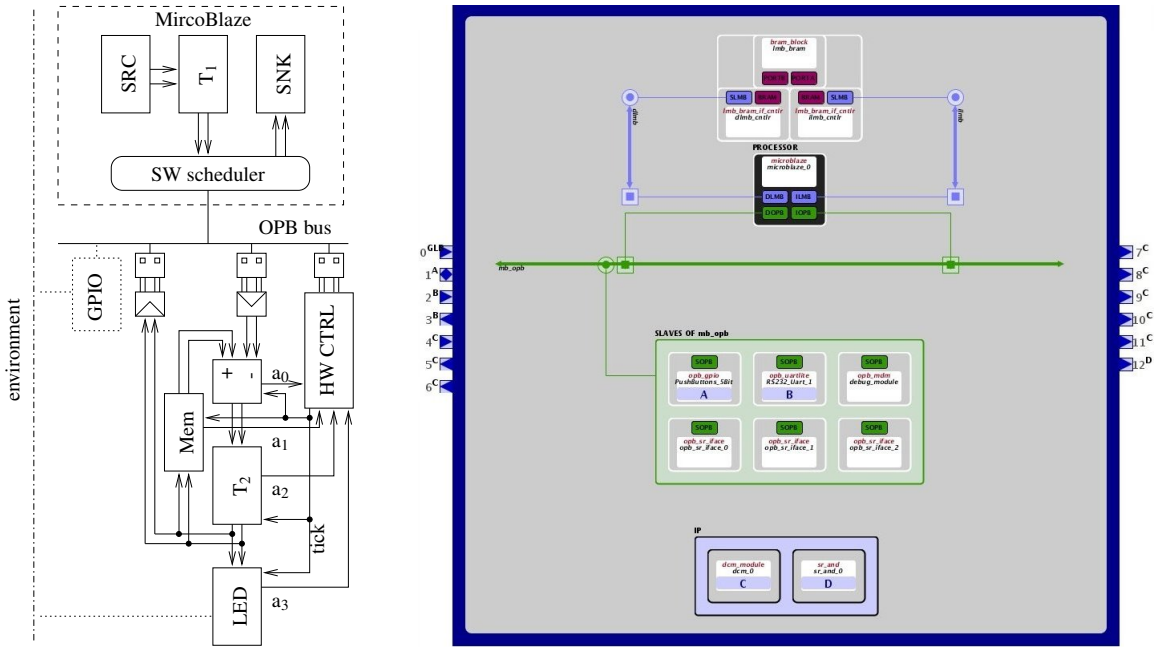

Figure 4: A synchronous reactive system implemented as a mixed hardware/software prototype. A software scheduler and a hardware controller ensure deterministic execution of the entire distributed application. Left hand side shows schematic view and right hand side shows block diagram captured from Xilinx Platform Studio [18].

synchronous to the inputs, is now a distributed issue. We solve this problem by generating a hardware controller which is polled periodically from a software scheduler. This way we can construct a self-scheduled mixed hardware/software system in a similar manner to *chaotic iteration*. The scheduler iterates over the software blocks wrapped by software functions. Each function returns if any output was updated and set to *defined* successfully. If so, the values from output variables are written to the hardware signals, using a memory-mapped hardware/-software interface by the scheduler. Each hardware block checks self-reliantly for sufficient *defined* inputs in order to generate *defined* outputs. During computation, a block sets an outgoing `active` signal to *true* and otherwise to *false*, e. g., a_0 in Figure 4. The hardware controller generates a `global active` signal by computing the logical OR on all incoming `active` signals. That way, the `global active` signal tells us if any hardware block is executed. If the `global active` signal becomes true, an `update` signal is set to true, indicating that any hardware block may have updated output signals. The software scheduler polls and resets the `update` signal and the `global active` signal. Thus, the software scheduler notices if software or hardware needs any update via the hardware/software interfaces. Any update of an output variable in software is forwarded to the hardware signal by the scheduler. If any output in hardware was updated (signal `update` is set) and the hardware is not longer active (`global active` is not set), then the hardware signals need to be forwarded to the software variables via the hardware to software interface. Otherwise, if there are no updates, no activity in hardware and no software functions produces any update, execution of an instant has reached a *fixed point*, i. e., the synchronous reaction of the entire system is fixed. Any other case means hardware blocks need more execution time or software blocks need further execution iterations. Finally, each block can implement an internal state update and each channel is set to *undefined* in order to start the execution of the next *instant*. In software, this requires calling a `tick()` function for each block. For hardware, this results in setting the `tick` signal to *true*.

6.1 Environmental Interaction

Execution of synchronous reactive systems needs special care about handling the interaction with the environment. To guarantee a monotonic behavior of the system, we must prevent the environment from interfering with the execution of an instant. From the specifications point of view, the execution of the synchronous system is assumed to be infinitely fast, so each action from the environment leads to an instantaneous reaction. But from the implementations point of view, there are computation delays and thus reaction needs some time. To fulfill the *synchrony hypothesis*, the system must react faster than new environmental actions occur. A typical implementation (e. g., [1]) to guarantee this constraint uses sampling of the inputs and outputs. When starting execution of an instant, all environmental inputs are sampled and buffered. After reaching a *fixed point*, the outputs to the environment will be updated. Using this decoupling scheme, we prevent interference of environmental actions and can guarantee monotonic behavior. Indeed, to guarantee the *synchrony hypothesis*, the overall execution time of a mixed hardware/software system needs to be faster than the environment. Note that any implementation using either hardware or software needs to react faster than the environment, though.

6.2 Example Implementation

A prototype of the application shown in Figure 2 and Figure 3 is implemented on an FPGA board with a Xilinx Virtex II Pro (XC2VP30) using the *chaotic iteration* scheduling approach. Figure 4 depicts the schematic view and the block diagram captured from Xilinx Platform Studio of the prototype implementation. The blocks *SRC*, T_1, and *SNK* are mapped to software, running on a Xilinx MicroBlaze soft-core processor, while T_2 and *LED* are implemented in hardware. A software scheduler and a hardware controller are used for controlling and monitoring the execution of blocks.

The hardware controller forwards the `global active` signal and an `update` signal to memory mapped registers read from the software scheduler. If requested by the software scheduler, the `tick` signal is set indicating the end of an instant. A set `update` signal causes the scheduler to update the hardware to software interfaces. Afterwards, the `update` signal is reset by the software scheduler in order to notice any further updates.

The main loop of the software scheduler implements the synchronous reactive execution of mixed hardware/software systems. A certain fixed point needs (1) updating the software part via the hardware to software interface, (2) executing the software block in *chaotic* order, (3) updating the hardware to software interfaces, and (4) testing if any hardware block is either still running or has already updated some channels to *defined*. Steps (1)-(4) are repeated while any update occurs in software or hardware. Otherwise, all channels are reset to *undefined*, the environment is sampled, and the very next fixed point calculation is started.

7 Conclusions

The SR model of computation [7] complements the modeling front-end of our platform-based design flow [11], supporting the modeling of reactive systems. We presented a basic mechanism to implement SR models on heterogenous platforms that can be automated and integrated to the design flow. A transformation step for Simulink models to SR models allows for automatic implementation of mixed hardware/software designs from functional Simulink models.

In the future, we will integrate the presented mapping steps into a platform-based design flow [11] enabling also automatic design space exploration of different implementations of

given Simulink models, including a search for optimized hardware/software architectures and prototype implementations on FPGA platforms.

References

[1] C. André, F. Boulanger, and A. Girault. Software Implementation of Synchronous Programs. In *IEEE International Conference on Application of Concurrency to System Design*, pages 133–142, Newcastle upon Tyne, UK, June 2001. IEEE Computer Society.

[2] Y. Atat and N.-E. Zergainoh. Simulink-based MPSoC Design: New Approach to Bridge the Gap between Algorithm and Architecture Design. In *Proceedings of the IEEE Computer Society Annual Symposium on VLSI*, pages 9–14, Porto Alegre, Brazil, 2007. IEEE Computer Society.

[3] M. Baleani, A. Ferrari, L. Mangeruca, A. L. Sangiovanni-Vincentelli, U. Freund, E. Schlenker, and H.-J. Wolff. Correct-by-construction transformations across design environments for model-based embedded software development. In *Proceedings of Design, Automation and Test in Europe*, pages 1044–1049. IEEE Computer Society, 2005.

[4] A. Benveniste and G. Berry. The Synchronous Approach to Reactive and Real-Time Systems. In *Proceedings of the IEEE*, volume 79, pages 1270–1282, 1991.

[5] G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[6] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert. From Simulink to SCADE/Lustre to TTA: a Layered Approach for Distributed Embedded Applications. *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 153–162, 2003.

[7] S. A. Edwards. *The Specification and Execution of Heterogeneous Synchronous Reactive Systems*. PhD thesis, EECS Department, University of California, Berkeley, 1997.

[8] Esterel Technologies. SCADE Suite^{TM}, 2007. www.esterel-technologies.com.

[9] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Kluwer Academic Publishers, 2002.

[10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[11] C. Haubelt, J. Falk, J. Keinert, T. Schlichter, M. Streubühr, A. Deyhle, A. Hadert, and J. Teich. A SystemC-based Design Methodology for Digital Signal Processing Systems. *EURASIP Journal on Embedded Systems, Special Issue on Embedded Digital Signal Processing Systems*, 2007:Article ID 47580, 22 pages, 2007. doi:10.1155/2007/47580.

[12] K. Huang, S. I. Han, K. Popovici, L. Brisolara, X. Guerin, L. Li, X. Yan, S.-I. Chae, L. Carro, and A. A. Jerraya. Simulink-based MPSoC design flow: case study of Motion-JPEG and H.264. In *Proceedings of the 44th Design Automation Conference*, pages 39–42, San Diego, California, 2007. ACM.

[13] M. Jersak, Y. Cai, D. Ziegenbein, and R. Ernst. A Transformational Approach to Constraint Relaxation of a Time-driven Simulation Model. In *Proceedings of the 13th international symposium on System synthesis*, pages 137–142, Madrid, Spain, 2000. IEEE Computer Society.

[14] K. Schneider and T. Schuele. Averest: Specification, Verification, and Implementation of Reactive Systems. In J. Desel and Y. Watanabe, editors, *Conference on Application of Concurrency to System Design*, St. Malo, France, 2005. IEEE Computer Society.

[15] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. F. Deprettere. System Design Using Kahn Process Networks: The Compaan/Laura Approach. In *Proceedings of Design Automation and Test in Europe*, pages 340–345, 2004.

[16] The MathWorks, Inc. Real-Time Workshop 6: Target Language compiler, 2007. www.mathworks.com.

[17] The MathWorks, Inc. Simulink - Using Simulink, 2007. www.mathworks.com.

[18] XILINX. *Embedded SystemTools Reference Manual - Embedded Development Kit EDK 8.1i*, October 2005.