A Transformational Approach to Constraint Relaxation of a Time-driven Simulation Model

Marek Jersak, Ying Cai, Dirk Ziegenbein, Rolf Ernst
Institut für Datenverarbeitungsanlagen, Technische Universität Braunschweig, Germany
$\{jersak|cai|ziegenbein|ernst\}$@ida.ing.tu-bs.de

Abstract

Time-driven simulation models typically model timing in an idealized way that is over-constrained and cannot be directly implemented. In this paper we present a transformation to relax the constraints imposed by a time-driven simulation model, thus creating a larger design space. We transform the system into SPI, a common intermediate representation for heterogeneously specified embedded systems. At the SPI level critical timing constraints are (re)introduced, resulting in a representation that is well suited for global system analysis, optimization and synthesis.

1. Introduction

Most high-level simulation models either do not model timing, or model timing in an idealized way that cannot be implemented. In neither case, specification of timing constraints and analysis of their satisfiability is possible. However, timing analysis is necessary for the implementation of embedded real-time systems.

Since system models supporting timing analysis are not available, timing analysis is not done today until late in the design process, close to implementation. Because of the high level of detail involved (ISA and HDL simulators) and the ever increasing system complexity, this approach is very time-consuming. Even worse, there is usually no guarantee that all the critical corner cases will be found.

System analysis is even more difficult for the common case of embedded systems specified using several languages or tools with different models of computation. Each of these models has properties which can be exploited for analysis and optimized implementation, but these properties are different for each model, thus inhibiting analysis and optimization across language boundaries.

The SPI model is a novel internal system-level representation that enables global analysis, in particular of timing, early in the design flow, as well as optimization and synthesis of heterogeneously specified embedded systems [14, 15]. Since systems are not modeled directly in SPI, methods to translate different languages or models of computation into SPI are needed.

In this paper, we present the translation of a time-driven simulation model into SPI, using *Simulink* [11] as an example. The translation relaxes the over-constrained exact timing assumed for all Simulink components, thereby yielding a larger design space. Timing constraints are (re)inserted afterwards and only where critical for correct real-time implementation.

We introduce the main aspects of SPI in section 2. In section 3 we describe the Simulink model of computation. In section 4 we discuss which aspects of the Simulink model of computation have to be preserved and which should be abstracted. Based on those findings, we formulate a set of translation rules. We show how timing-constraints are (re)introduced, and present our design flow and implementation. In section 5 we present an example before concluding in section 6.

1.1. Related work

Real-Time Workshop [10] is the standard software codegenerator for Simulink. It can be used for prototyping or as a basis for production code, but lacks the ability to guarantee timing. Timing is also a problem for other codegenerators, e.g. [7]. Ptolemy II [5] could serve as an alternative target for the translation from Simulink. However, SPI is the more suitable model for timing analysis, as it abstracts process function into externally visible parameters. RATAN [1] analyzes satisfiability of timing constraints for an event-driven process model. This is not applicable here, since Simulink uses a time-driven execution model. Once we have transformed the Simulink execution model into activation by relative execution rates, it becomes possible to take a similar approach. In [12, 14] it has been shown how to translate several models of computation into SPI, including Kahn graphs, event driven models, and periodic process systems.

2. The SPI model

SPI (**S**ystem **P**roperty **I**ntervals) [14, 15] is an internal representation that facilitates the safe integration of parts of a heterogeneously specified system and enables system analysis across language boundaries. In this section, only the basic concepts of SPI, which are necessary to understand this paper, are introduced informally.

Computational elements in SPI are *processes* that communicate via two different channel types, *FIFO-queues* and *registers*. The activation of SPI processes is implicit and based on data availability, i. e. a process *may* start if there is sufficient data on its input queues to support one execution.

The SPI model elements are not characterized by their exact behavior but by a *set of parameters* such as activation function, latency time and data rates. Data is produced on and consumed from all connected FIFO-queues simultaneously at the end of the execution of a process (atomic buffer update).

Parameters may be specified as *value intervals*. For example, a data rate interval limits the number of produced or consumed data, capturing data-dependent communication. Using the concept of process modes, such conditional process behavior depending on internal states or input data can also be modeled explicitly, i. e. a process is refined to have different behaviors (parameter sets) that are modeled as modes.

Virtual processes and channels are used to model the system environment and to represent additional information for synthesis. Since virtual elements are not part of the system functionality, they do not need to be implemented.

Timing constraints in SPI are specified using latency path constraints that limit the time for causal process executions along a certain path. Other types of timing constraints can be modeled by latency path constraints over virtual elements.

3. Simulink

Simulink [11] is a block diagram oriented industry standard tool for simulating mixed reactive/transformative, nonlinear dynamic systems that builds on the MATLAB environment for technical computing. It uses a time-driven simulator that supports continuous-time, discrete-time (also multi-rate) or a hybrid of the two. The basic, time-driven execution model is extended by additional semantics, such as triggered and enabled subsystems.

3.1. Application and limits

A typical application of Simulink is the modeling and simulation of a physical system (e.g. from the automotive domain) together with associated controlling and signal processing functionality. While the model of the physical system is eventually replaced by the actual system, the signal processing and controller design is often implemented as an embedded system.

Real-Time Workshop (RTW, [10]) generates C code from Simulink block diagrams for a variety of host and real-time platforms. The generated code can be used for prototyping and as a basis for production code.

The main weakness of RTW and other code generators (e.g. [7]) is their inability to guarantee timing, since they lack a method to specify timing constraints and do not easily support analysis of value intervals for execution latencies [13]. In case of RTW, parts of the system that have the same execution rate are clustered into individual processes, which are triggered by periodic interrupts [10]. There is no guarantee that a process is not interrupted prematurely, if its execution time is longer than the available time slot. This is particularly hard to detect during testing for the common case of data-dependent execution times.

RTW also does not allow to control process granularity. Additionally, to our knowledge there do not exist partitioning tools or hardware generators for Simulink, and hence neither co-design nor multi-processor implementation are supported.

3.2. Model of computation

A Simulink block-diagram represents a set of differential equations. Directed edges between the blocks are used to communicate values. These edges have *register semantics* (non-destructive read, destructive write). Consequently, in multi-rate designs a value on an edge can be read multiple times (if the reading block is executed at a faster rate than the writing block), or it can be overwritten before having been read (if the reading block is executed at a slower rate than the writing block).

A number of fixed-step and variable-step solvers are available to solve the set of differential equations at certain points in time. Variable-step solvers are important in certain situations to model a physical system accurately (e.g. to find zero crossings), but are of little significance for the design of control and signal-processing embedded systems. These applications are generally modeled using *fixed-step solvers* to enable code generation into periodic processes.

Simulink uses an idealized timing model for block execution and communication. Both happen *infinitely fast at exact points in simulated time*. Thereafter, simulated time is advanced by exact time steps. All values on edges are constant in between time steps.

While this timing model is appropriate for numerical solutions of differential equations, it obviously cannot be implemented in an embedded system. This fact and the lack of means to specify implementable timing constraints in Simulink explain the problems current code generators have with timing (section 3.1).

4. Translation

In this section we show how to translate the semantics of Simulink into a system of execution dependencies in SPI, thereby replacing the restrictive absolute periodic timing by a relative timing. This approach increases the design space for optimization, supports combination with other models of computation and allows to introduce global timing constraints.

The following aspects of the Simulink model of computation (section 3.2) have to be preserved: 1) causality and the resulting partial ordering of Simulink blocks, 2) relative execution rates between Simulink blocks, and 3) read and write access sequence on each edge to model the register semantics used by Simulink for communication.

Timing has to be relaxed, since 1) it is fundamentally impossible to implement blocks or channels with zero execution time and 2) activation of all blocks at exact points in time is unnecessarily restrictive.

Instead, there are usually few processes where timing is critical, typically I/O processes which have to satisfy external requirements. Additionally, *latency path constraints*, typically between inputs and outputs or along a cycle, may have to be specified to guarantee timely completion [6, 14, 15].

4.1. Basic translation rules

Our translator uses the following basic rules when mapping a Simulink block-diagram into SPI elements.

1. Each Simulink block is mapped into one SPI process. (Clustering is considered in section 4.3.)

2. Each Simulink port is mapped into one virtual SPI process that models the environment.

3. Each Simulink edge is mapped into one SPI register channel to maintain Simulink destructive write, non-destructive read semantics. One token is written (read) on each register channel per activation of the writing (reading) process.

4. A pair of virtual FIFO-queues is generated between every two processes that communicate over a register channel. Activation of the generated SPI processes is enabled by availability of tokens on those virtual FIFO-queues. The time-driven Simulink model of computation is thus transformed into a data-driven model which is supported by SPI.

5. Relative execution rates and partial ordering between Simulink blocks are maintained by writing (reading) the appropriate number of tokens to (from) each virtual queue, and by the number of initial tokens on each virtual queue, as specified in the following equations.

$$
\begin{aligned}
r_{virt}(P_i) &= t_s(B_i) \\
n_{C_j(P_{wr} \rightarrow P_{rd})} &= r_{virt}(P_{rd}) - 1 \\
n_{C_j(P_{rd} \rightarrow P_{wr})} &= r_{virt}(P_{wr})
\end{aligned}
$$

$r_{virt}(P_i)$ is the number of tokens written and read by process (P_i) per execution on each of its virtual channels. $t_s(B_i)$ is the sample time[1] of block i. n_{C_j} is the number of initial tokens on virtual queue C_j. The direction of queue C_j is indicated by indices P_{wr} and P_{rd}, which refer to the writing and reading processes of the corresponding register channel.

We show the application of those rules for the simple system in figure 1. It consists of four Simulink blocks, one input and two output ports. The respective sample times (in ms) are annotated to each block.

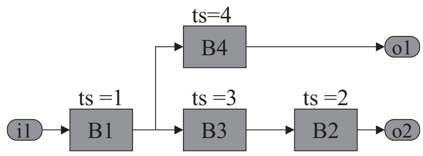

Figure 1. A simple Simulink block diagram

The SPI representation of the system is shown in figure 2. Large circles indicate processes with the number of tokens written or read per execution annotated to each process port. Small circles indicate FIFO-queues with the number of initial tokens annotated. Squares indicate register channels. Dotted elements are virtual.

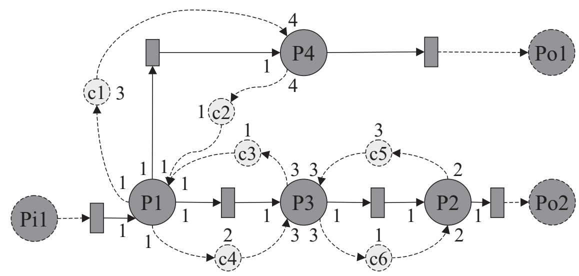

Figure 2. Generated SPI graph

For example, each execution of process P_3 consumes 3 tokens on virtual channels C_4 and C_5, produces 3 tokens on virtual channels C_3 and C_6, and reads and writes the connected registers.

Figure 3 shows one possible execution sequence of the generated SPI representation. For each channel, the number of initial tokens, and the number of tokens after a process

[1] If sample times are not integer values, they have to be multiplied by an appropriate factor to obtain integer values for r_{virt} for every process.

has been executed is shown. Clearly, the exact timing of processes is not constrained. For comparison, the vertical dotted bars indicate advancement of the original *Simulink sample time* by $1ms$.

	P1	P4	P3	P2	P1	P1	P2	P1	P3	P1	P2	P4	...
c1 3	4	0			1	2		3		4		0	
c2 1	0	4			3	2		1		0		4	
c3 1	0		3		2	1		0	3	2			...
c4 2	3		0		1	2		3	0	1			
c5 3			0	2			4		1		3		
c6 1			4	2			0		3		1		

Figure 3. Valid execution sequence

4.2. (Re)introducing timing constraints

Once the design has been translated to SPI, timing constraints can be specified exactly as needed, also across input language boundaries. For example, the absolute execution rate for a single SPI process generated from Simulink can be constrained, e.g. by requiring exact periodic activation or relaxed periodic activation. This automatically produces the maximum possible execution time intervals for *all processes* coupled to this process through pairs of virtual queues (see below). If any of these processes is not executed during its execution time interval, the constraint specified by the designer is violated. The remaining design space is available for exploration.

Exact periodic activation An exact latency constraint on a virtual "self"-channel of a process forces an exact periodic activation of this process. This is shown in figure 4, where process P_1 has to be activated exactly every $1ms$ (presumably because exact sampling of input i_1 is critical) to satisfy the latency constraint on channel C_s.

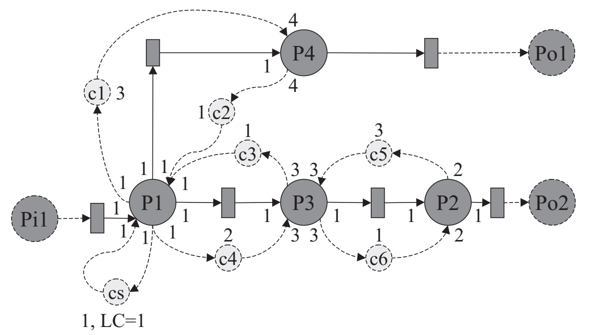

Figure 4. Exact periodic activation of P_1

"Exact" timing implies process instantiation accuracy at timer or clock resolution. This usually requires hardware implementation of the process, or activation by a highest priority interrupt which is generated by a timer, and which is guaranteed immediate handling.

Relaxed periodic activation If the activation time of a periodic process does not have to be exact, a relaxed periodic activation can be used instead. An additional *virtual process* P_{Clk} with an exact periodic activation can be used for this purpose. This is shown in figure 5 for process P_1 (the rest of the system is the same as in figure 4).

Process P_1 is connected to P_{Clk} via virtual channel C_7 with a latency constraint interval that specifies the earliest and latest valid completion times of process P_1 relative to process P_{Clk}.

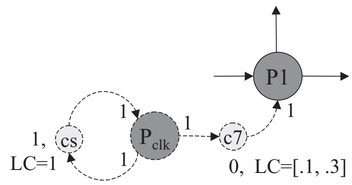

Figure 5. Relaxed periodic activation of P_1

Latency path constraints A periodic timing constraint for a *single* process result in maximum execution time intervals for, and hence *implicit* latency path constraints between any pair of processes that are coupled to this process through relative execution rates. This is shown in the timing diagram in figure 6, which is a valid single-processor implementation of the constrained SPI graph in figure 4 without process P_4. Process P_1 is executed exactly every $1ms$ because of its periodic activation constraint. Processes P_2 and P_3 have been scheduled as late as possible without violating the timing constraint for process P_1. The arrows indicate the resulting longest possible latency for a change of value at output o_2 because of a change of value at input i_1. Stricter latency path constraints between inputs and outputs can be specified *explicitly* if necessary.

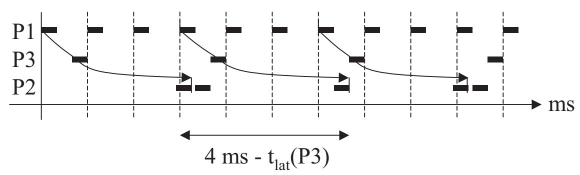

Figure 6. Implicit latency path constraint

4.3. Advanced translation issues

In this section, we present several advanced translation issues. Detailed information can be found in [9].

Triggered Subsystems A triggered subsystem has a trigger condition which is a function of the value at a special trigger input. Every time the block driving the trigger port is executed, the trigger condition is evaluated. The subsystem is only executed if the trigger condition is satisfied. A triggered subsystem can be modeled as a SPI process with two modes [15], one modeling the system's behavior if the trigger condition is satisfied, and the other modeling the case that the trigger condition is not satisfied. The SPI representation of a triggered subsystem is shown in figure 9, process *Cur*.

Continuous sample time Even if blocks with continuous sample time are present, Simulink has to solve a system in discrete time-steps. Therefore, additionally to signal and state values, also their derivatives are used to approximate continuous time. Since inner function is abstracted in SPI, and since the basic Simulink concept of periodic execution is maintained for time-continuous blocks, they can be mapped to SPI in the same way as blocks with a discrete sample time.

Granularity The task of finding a good process granularity is important for an efficient implementation [2]. As an extension to rule 1 in section 4.1, we thus allow the designer to specify which Simulink blocks should be clustered into a single SPI process, independent of subsystems used in Simulink to build hierarchical block diagrams. This independence is useful, since subsystems in Simulink are a structuring concept to facilitate comprehension and navigation of the design. However, this kind of structuring may not be the best solution when it comes to implementation.

4.4. Design Flow and Implementation

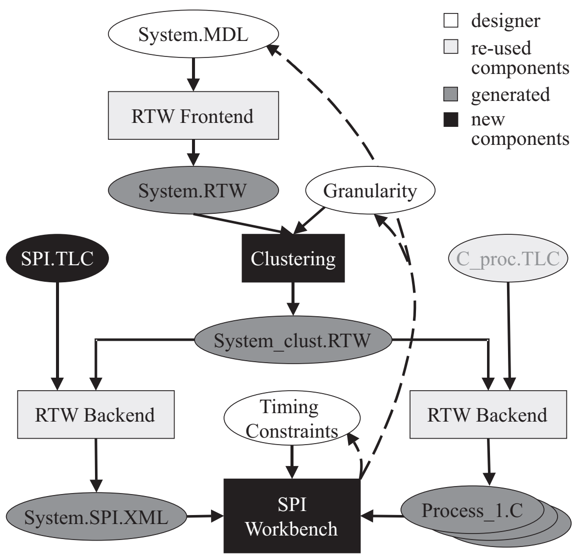

Figure 7. Design flow

Our design flow is shown in figure 7. To facilitate implementation, we have reused as much as possible from *Real-Time Workshop (RTW)* [10].

1. The designer starts by specifying and simulating a system in Simulink (*System.MDL*).
2. The designer writes a granularity file to specify the clustering of Simulink blocks into SPI processes.
3. Using the standard RTW front-end and an additional clustering stage, the Simulink block-diagram is translated into an intermediate format (*System_clust.RTW*), a slight extension of the RTW intermediate format.
4. The intermediate format is translated into an XML [8] representation of SPI processes, channels, and process and channel attributes (*System.SPI.XML*), which can be validated by our *SPI.DTD* [2]. The XML representation is used as an exchange format in the SPI workbench [4]. This step is performed by the standard RTW code-generator, using a set of SPI XML code-generation rules (*SPI.TLC*).
5. The standard RTW C-code generator is modified to generate a single C function for every SPI process generated in step 4. This separates process function from scheduling, thus opening the design space for exploration.
6. The designer specifies timing constraints as necessary.

At this point, a SPI model of the Simulink design is available, with timing constraints and references to an executable process model in C. Tools that hook into the *SPI workbench* [4] can now be applied. Behavioral intervals for process and communication timing can be analyzed [13] and the results annotated to SPI processes and channels. The SPI model generated from the Simulink design can also be integrated with SPI models generated from system parts specified in other languages, thus allowing global analysis and optimization. Analysis results can be fed back to the Simulink designer to change the system, granularity, or timing constraints. Once satisfied, the system can be implemented.

5. Example

We now apply our approach to an induction motor controller [3] (figure 8) modeled in Simulink. It uses pulse width modulation (PWM) with a frequency of $16kHz$.

Additional knowledge about real-time requirements and constraints is necessary, which cannot be represented in the Simulink model. In the reference implementation [3], block *Measure* is triggered by an external interrupt in the middle of each PWM period to make sure that the PWM current is not measured at a PWM edge. Block *Speed / Flux* is triggered by a timer interrupt to get an absolute time reference.

A SPI representation of the controller including timing constraints is shown in figure 9.

[2] A DTD is a grammar file used to define XML-tags and their properties.

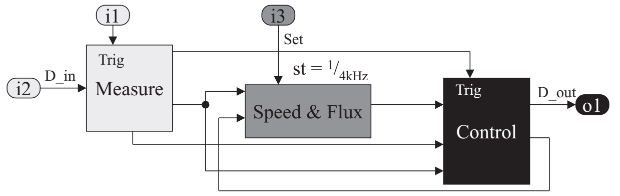

Figure 8. Induction motor controller

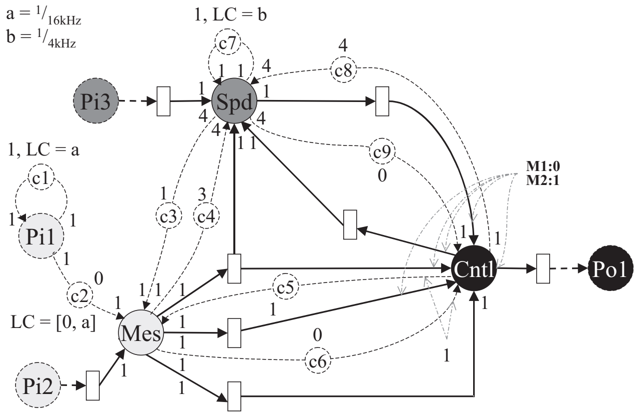

Figure 9. SPI representation of the controller

1. Exact timing has been replaced by activation using relative execution rates. This leaves a larger design space.
2. The only process with an exact periodic activation constraint is P_{Spd} as required by the design document [3].
3. P_{Mes} has a relaxed periodic activation constraint, since based on the design document the trigger signal for Simulink block *Measure* arrives periodically, with a certain jitter (we have assumed a full period).
4. Processes P_{Cur} is modeled using two modes (section 4.3). In both modes, the same numbers of tokens on virtual channels are produced and consumed to maintain causality and relative execution rates, but only in mode M_2 data is consumed and produced on the register channels.

At this point, the maximum design space for the Simulink model that satisfies all timing constraints has been obtained. Execution time intervals can be obtained using static timing analysis methods [13]. With all information in one place, efficient design space exploration can now be performed.

6. Conclusion

We have demonstrated how relaxation of timing constrains increases the design space for real-time implementation of a time-driven simulation model using Simulink as an example. We translate Simulink into the SPI model, and in this process capture the essential aspects of the Simulink model of computation, while allowing to abstract those aspects that cannot be implemented or that severely limit the design space. Timing constraints that cannot be captured in Simulink can then be annotated on the SPI level. This approach enables global system-level analysis, in particular of timing, as well as optimization and synthesis of heterogeneously specified embedded systems.

References

[1] A. Dasdan, A. Mathur, and R. K. Gupta. RATAN: A tool for rate analysis and rate constraint debugging for embedded systems. In *Proceedings ED&TC '97*, 1997.

[2] B. P. Dave, G. Lakshminarayana, and N. K. Jha. COSYN: Hardware-software co-synthesis of embedded systems. In *Proceedings 34th Design Automation Conference (DAC '97)*, Annaheim, USA, June 1997.

[3] dSPACE. *Squirrel Cage Induction Motor Control with DS1103 Controller Board*, Sept. 1999.

[4] R. Ernst, D. Ziegenbein, K. Richter, L. Thiele, and J. Teich. Hardware/software codesign of embedded systems - the SPI workbench. In *Proceedings IEEE Workshop on VLSI*, Orlando, USA, June 1999.

[5] A. Girault, B. Lee, and E. A. Lee. Hierarchical finite state machines with multiple concurrency models. *IEEE Trans. CAD*, June 1999.

[6] R. K. Gupta and G. D. Micheli. Specification and analysis of timing constraints for embedded systems. *IEEE Trans. CAD*, Mar. 1997.

[7] H. Hanselmann, U. Kiffmeier, L. Köster, and M. Meyer. Automatic generation of production quality code for ECUs. Technical report, dSPACE GmbH, Mar. 1999. Distributed during Embedded Intelligence, Nürnberg, Germany.

[8] E. R. Harold. *XML Bible*. IDG Books Worldwide, 1999.

[9] M. Jersak. Translating Simulink to SPI. Technical report, IDA, TU Braunschweig, Feb. 2000.

[10] The MathWorks, Inc. *Real-Time Workshop User's Guide, Version 3*, Jan. 1999.

[11] The MathWorks, Inc. *Using Simulink, Version 3*, Jan. 1999.

[12] K. Richter. Developing a general model for scheduling of mixed transformative/reactive systems. Master's thesis, Technical University of Braunschweig, Jan. 1998.

[13] F. Wolf and R. Ernst. Intervals in software execution cost analysis. In *Proceedings ISSS '00*, Madrid, Spain, Sept. 2000.

[14] D. Ziegenbein, R. Ernst, K. Richter, J. Teich, and L. Thiele. Combining multiple models of computation for scheduling and allocation. In *Proceedings Sixth International Workshop on Hardware/Software Co-Design (Codes/CASHE '98)*, Seattle, USA, Mar. 1998.

[15] D. Ziegenbein, K. Richter, R. Ernst, J. Teich, and L. Thiele. Representation of process mode correlation for scheduling. In *Proceedings International Conference on Computer-Aided Design (ICCAD '98)*, San Jose, USA, Nov. 1998.