Efficient Modelling and Simulation of Embedded Software Multi-Tasking using SystemC and OSSS

Philipp A. Hartmann, Henning Kleen
OFFIS Institute for Information Technology
Oldenburg, Germany

Philipp Reinkemeier, Wolfgang Nebel
Carl von Ossietzky University
Oldenburg, Germany

Abstract

Since the software part in today's designs is increasingly important, the impact of platform decisions with respect to the hardware and *the software infrastructure (OS, scheduler, priorities, mapping) has to be explored in early design phases.*

In this paper, we present an extension of the existing SystemC™-based OSSS design flow regarding software multi-tasking in system models. The simulation of the OSSS software run-time model supports different scheduling policies, as well as efficient timing annotations, and deadlines. Inter-task communication is modelled via user-defined Shared Objects*. The impact of timing annotation granularity on the achievable simulation performance is studied. As a result, a* lazy synchronisation *scheme is proposed, that is based on omitting SystemC time synchronisations, that do not have observable effects on the application model.*

1. Introduction

The increasing pressure on time-to-market and cost of today's embedded systems requires ever increasing design productivity. As a result, embedded software becomes more and more important, since the required design effort for a software function is usually expected to be lower than for the respective hardware parts. Additionally, an increased flexibility and the possibility of changes late in the design process are an advantage. On the other hand, the choice of the "correct" software architecture such as the chosen RTOS, task priorities, scheduling policies, mapping of tasks on processors is by no means a simple task.

To help developers during this phase of the design space exploration, efficient and fast modelling of different architecture alternatives has to be supported by the chosen design flow. Apart from considering the underlying hardware platform, this includes the early analysis of software/RTOS effects on the system's overall performance, which is important especially if multiple tasks are sharing a single processor. Real-time capabilities have to be explored by choosing e.g. the scheduler, and task priorities to fulfil a given set of requirements like deadlines or other application specific constraints.

Not only since its IEEE standardisation [12] is SystemC™ [16] a very popular modelling language for system-level modelling of complex hardware/software systems. Since the modelling of real-time software specific aspects is not directly supported by SystemC itself, several extensions to SystemC have been developed, to enable the early exploration of real-time software properties, some of which will be compared briefly in Section 2.

The approach to software multi-tasking presented in this paper is based on OSSS – the *Oldenburg System Synthesis Subset*, an extension of the SystemC synthesisable subset [19] with object-oriented features. An introduction to the accompanying OSSS design flow is given in Section 3. The OSSS methodology is characterised by a layered approach and its partly automated path to an implementation. Up to now, the support for software modelling in OSSS has been limited to a single task per processor. In this work, we extend the modelling capabilities of OSSS with support for simulating multiple tasks on a processor running a given RTOS.

Due to the object-oriented approach, communication between different components is modelled by abstract method calls in OSSS to so called *Shared Objects*. This concept is reused for the modelling of inter-task communication in the new software parts. The main advantage of this approach is the abstraction of difficult RTOS synchronisation primitives and therefore a more robust modelling environment. In Section 4, we present the new software multi-tasking features of OSSS, like tasks and their properties, timing abstraction and *Software Shared Objects*.

An important property of abstract software models is their simulation performance. The synchronisation overhead between several tasks (i.e. SystemC processes) and the underlying simulation kernel is one of the limiting factors for this. As shown in Section 5.1, the granularity of the timing annotations within the *Software Tasks* has a direct impact on the overall simulation performance. In Section 5.2, we show that this impact can be significantly reduced by applying our *lazy synchronisation* scheme, which reduces the SystemC overhead without changing the visible system behaviour and correctness. The paper concludes in Section 6 with a summary and directions of future work.

2. Related Work

Many different approaches to modelling embedded software in the context of SystemC have been proposed. Some of them, like the SPACE framework [2] rely on the

co-simulation with an external RTOS simulator or even an instruction set simulator. Although these approaches may provide higher simulation accuracy, they usually lack the required simulation performance for early platform exploration and are therefore out of the scope of this paper.

Abstract RTOS models, like the one presented for SpecC in [5] are better suited for early comparison of different scheduling and priority alternatives. The timing accuracy and therefore the simulation performance of this approach is limited by the fixed minimal resolution of discrete time advances.

Several approaches based on abstract task graphs [11, 14, 18] have been proposed as well. In this case, a pure functional SystemC model is mapped onto an architecture model including an abstract RTOS. The mapping requires an abstract task graph of the model, where estimated execution times can be annotated on a per-task basis only, ignoring control-flow dependent durations. This reduces the achievable accuracy.

A single-source approach for the generation of embedded software from SystemC-based descriptions has been proposed in [3, 10, 17]. Starting from untimed, heterogeneous models in HetSC, a POSIX conformant description can automatically be generated by the SWgen methodology. The performance analysis of the resulting model with respect to an underlying RTOS model can be evaluated with the PERFidiX library, that augments the generated source via operator overloading with estimated execution times. Due to the fine-grained timing annotations, the model achieves a good accuracy but relatively weak simulation performance. This interesting approach might significantly benefit from our proposed *lazy synchronisation*, as presented in Section 5.

An early proposal of a generic RTOS model based on SystemC has been published in [13]. The presented abstract RTOS model achieves time-accurate task preemption via SystemC events and models time consumption via a `delay()` method. Additionally, the RTOS overhead can be modelled as well. Two different task scheduling schemes are studied: The first one uses a dedicated thread for the scheduler, while the second one is based on cooperative procedure calls, avoiding this overhead. Although in this approach explicit inter-task communication resources are required (message queue, ...), the simulation time advances simultaneously as the tasks consume their delays.

In [9], an RTOS modelling tool is presented. Its main purpose is to accurately model an existing RTOS on top of SystemC. It can not be directly used by a system designer. In this approach, the next RTOS "event" (like interrupt, scheduling event, ...) is predicted during runtime. This improves simulation speed, but requires deeper knowledge of the underlying system.

Just recently, another two approaches [15, 20] have been published in German. In [15], tasks are derived from a special base class, which augments the regular SystemC `wait()` method with synchronisation calls to an abstract RTOS model. An additional FIFO class, with a similar synchronisation scheme is included as a HW/SW communication primitive. Regular SystemC events can be used by the application for inter-task communication as well. In [20], the main focus lies on precise interrupt scheduling. For this purpose, a separate scheduler is introduced to handle incoming interrupt requests. This is similar to our ring-based scheduling approach (see Section 4.1). Timing annotations and synchronisation within user tasks is handled by a replacement of the SystemC `wait()`, similar to [15].

In this paper, we present an abstract model for software multi-tasking based on OSSS, which included some properties of the above mentioned approaches, especially concerning a simple RTOS model. The flexible integration of user-defined communication mechanisms via *Shared Objects* (Section 4.3) and the efficient handling of timing annotations (Section 5.2) is the main contribution of our approach.

3. The OSSS Design Flow

Based on an object-oriented hardware design approach [7], one of the main objectives of OSSS is to enable the usage of object-oriented features known from languages such as C++ in a synthesisable SystemC model. This includes concepts such as classes, inheritance, polymorphism and abstract communication based on method calls. OSSS extends the synthesisable subset of SystemC [19] by defining synthesis semantics for many of these features. Furthermore, new concepts specifically targeted to the modelling and design of embedded systems are introduced to raise the level of abstraction and increase the expressiveness of the language.

OSSS defines separate layers of abstraction for improving refinement support during the design process. The design entry point in OSSS is called the *Application Layer*. By manually applying a mapping of the system's components, the design can be refined from *Application Layer* to the *Virtual Target Architecture Layer*, which can be synthesised to a specific target platform in a separate step by the synthesis tool *Fossy* [4].

3.1. Application Layer

On this layer the hardware/software system is modelled as a set of communicating processes, representing hardware modules and software tasks. The *Application Layer* model abstracts from the details of the communication between the components of a model, such as the actual implementation of the communication channel, even across HW/SW boundaries.

One concept introduced by OSSS without a direct equivalent in C++ is the so called *Shared Object*, which equips user-defined classes with specific synchronisation facilities. Due to their well-defined synthesis semantics, *Shared Objects* can act as a replacement for some non-synthesisable features of SystemC such as hierarchical channels, mutexes and semaphores.

Synchronisation is performed by arbitrating concurrent accesses and a special feature called *Guarded Methods*, that can be used to block the execution of a method according to an user-defined condition. As a result, they are especially useful for modelling inter-process communication, between hardware and software processes. Communication between modules or tasks and *Shared Objects* is

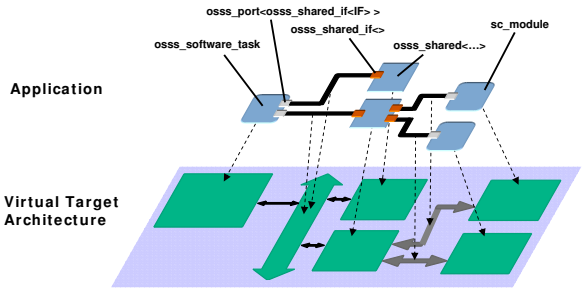

Figure 1. Mapping of Application Layer to Virtual Target Architecture Layer

performed by method calls through abstract communication links (binding).

A comprehensive description of the *Shared Object* concept, including several design examples, is part of the publicly available simulation library [6].

3.2. Virtual Target Architecture Layer

In a refinement step the *Application Layer* model is transformed to a so-called *Virtual Target Architecture*. This involves mapping software tasks to processor(s) and hardware modules to certain hardware blocks as shown in Figure 1. Moreover, the abstract communication links of the *Application Layer* model are then to be mapped to specific communication infrastructure, like buses or point-to-point channels.

To enable an easy mapping of the method-based communication on the *Application Layer* to a signal-based communication OSSS uses a concept known as *Remote Method Invocation* (RMI). A detailed description of this scheme is beyond the scope of this paper, but can be found in [8]. Basically, the implementation of the RMI concept allows the transport of method calls including their parameters and return values over arbitrary communication infrastructures and is used for HW/HW as well as HW/SW communication.

The *Virtual Target Architecture Layer* model of the design is then translated automatically by the synthesis tool *Fossy* [4], producing RTL SystemC or VHDL output which is used for further synthesis by vendor-specific implementation tools.

4. Modelling Software in OSSS

The approach to modelling multi-tasking software in OSSS is *not* meant to directly model existing real-time operating system (RTOS) primitives. The *Software Tasks* (see Section 4.2) in OSSS are meant to *run* on-top of a rather generic (but lightweight) run-time system, where the synchronisation and inter-task communication is modelled with *(Software) Shared Objects* (see Section 4.3), similar to the modelling with OSSS in the hardware domain. This enables a seamless specification environment, where the same concepts are used for both, hardware *and* software on the application layer (see Section 3.1). The flexible timing annotation mechanism (Section 4.4) enables high simulation performance, since the synchronisation overhead with the SystemC kernel can be minimised.

4.1. Abstraction of Run-time System

The basis of the OSSS software run-time simulation model is an OSSS RTOS abstraction class. This predefined library element handles the time-sharing of a single processor by several *Software Tasks* (see Section 4.2), which are bound to this OS instance. A specific scheduling policy can be bound to each set of tasks, grouped by the same *ring*, as depicted in Figure 2. Several frequently used scheduling policies are already provided by the library, most notably

- static priorities (preemptive and cooperative),
- time-slice based round-robin,
- earliest-deadline first,
- and rate monotonic.

Additionally, user-defined schedulers can be implemented through an abstract interface class. The RTOS overhead of context switches and execution times of scheduling decisions can be annotated as well.

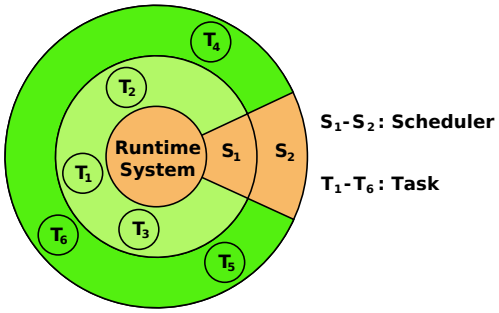

Figure 2. Ring-based task scheduling

Several *rings* can be specified by the designer. The rings are an additional priority layer, where every ring receives its own scheduling policy. The processor is assigned to a task from lower-priority rings only, if there is no task in `ready` state (see Section 4.2) in any higher priority ring. An example use case for this feature is to model (prioritised, non-preemptive) interrupt service routines with otherwise time-sliced round-robin scheduled user tasks.

With this set of basic elements, the behaviour of the real RTOS on the target platform can be modelled. Task synchronisation is not part of the modelling elements, since the inter-task communication is meant to be modelled by using *Software Shared Objects*. On the target platform, an implementation of this primitive will be provided by the means of the scheduling primitives of the architecture (see Section 4.3).

4.2. Software Tasks

SystemC processes, that are meant to be implemented in software are modelled as *Software Tasks* in OSSS. These tasks are derived from the common base class and

define their behaviour within a `main()` method. In the multi-tasking OSSS implementation, tasks are equipped with additional properties, like a priority, an initial startup time, optional periods and deadlines, an optional *task ring* (see Section 4.1).

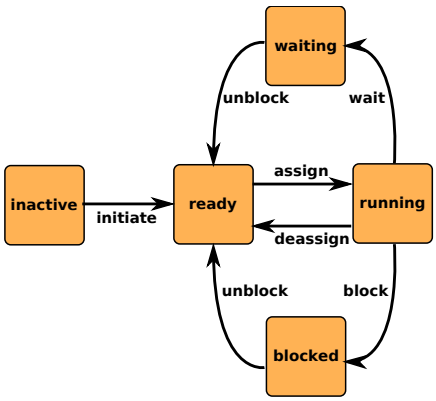

Figure 3. Task states and transitions (terminate edges omitted)

During simulation, the tasks can be in different states as shown in Figure 3. The distinction between `blocked` and `waiting` has been introduced to ease the detection of deadlocks. A task in `waiting` state will enter the `ready` state after a given amount of time, whereas a `blocked` task can only be de-blocked, once a certain condition is fulfilled (usually a *guard*, see below).

Technically, the *Software Tasks* are implemented with `SC_THREAD`s, that use internal synchronisation mechanisms with the RTOS abstraction during the simulation, to achieve the effect of only one "active" task at a time on a given processor. The task preemption is supported at arbitrary times, independently of the annotated execution times. If a task has been preempted, it continues to consume the remainder of the current time annotation, once it gets `running` again. Regular SystemC `wait()` calls are disabled within *Software Tasks*.

4.3. Software Shared Objects

The inter-task communication in an OSSS software model is specified like in an OSSS hardware model in terms of user-defined *Shared Objects* (see also Section 3), which are inspired by the protected objects known from Ada [1]. On the final platform, the software implementation of a shared object has to be integrated with the OSSS software run-time, since it usually requires RTOS primitives for the synchronisation.

As outlined in Section 3, *Shared Objects* provide mutual exclusive access and *Guarded Methods* to ensure deterministic behaviour across several concurrent tasks (see Listing 1). The guard mechanism resembles the well-known *monitor* concept and can directly be implemented e.g. on an underlying POSIX-compatible OS using condition variables (`pthread_cond_t`). The required locks for mutual exclusive access can be implemented using the existing locking mechanism of the underlying RTOS (e.g. mutexes).

```
// guard definition ( <name>, <condition> )
OSSS_GUARD( not_empty, cnt_items_ > 0 );

// guarded method declaration of int get()
// – blocking, until not_empty guard holds
OSSS_GUARDED_METHOD
  ( int, get, OSSS_PARAMS(0), not_empty );
```

Listing 1. Example of a guarded method.

Therefore, in OSSS the complexity of inter-task synchronisation primitives is hidden from the designer, which improves design productivity. It is planned to automatically generate the required run-time system from an OSSS software model and use cross-compilation techniques to translate *Software Tasks* and *Shared Objects* directly to the target machine code.

4.4. Software Execution Times

A proper modelling of software multi-tasking requires the consideration of time consumption of the modelled tasks. In OSSS, the **E**stimated **E**xecution **T**ime (EET) of a code block can be annotated within *Software Tasks* and inside methods of *Shared Objects* with the help of the `OSSS_EET()` macro (see also [6]).

The macro receives a duration as an `sc_time` argument, that estimates the execution time of the following code block. This enables a flexible and accurate annotation, depending on the required accuracy. Control structures can be efficiently annotated and during the simulation, the resulting execution time respects the (potentially data-dependent) control flow. Listing 2 exemplifies the syntax of these annotations. The simulation semantics of these annotations are discussed in Section 5.1.

`OSSS_EET()` blocks can not be nested and must not contain inter-task communication calls. As of today, these times are meant to be determined by profiling the cross-compiled code on the target processor. On the other hand, it is envisioned to extract and back-annotate these times automatically in the future.

In addition to the EETs, OSSS enables the designer to specify local deadlines for a specific code block. This is especially useful in combination with inter-task communication calls or preemptive scheduling policies.

The syntax follows the one for the EETs, meaning that a certain **R**equired **E**xecution **T**ime (RET) is specified by the `OSSS_RET()` macro, which guards the duration of the following code block. If required, RETs can be nested at arbitrary depth. The consistency of nested RETs is checked during the simulation like any other violation of the RETs, or the optional globally annotated deadline of a given task. If such a timing constraint can not be fulfilled during the simulation, it is reported by the library. Unmet RETs can arise from the choice of the scheduling policy, (additional) delays caused by blocking guard conditions, or simply unexpectedly long estimated execution times (e.g. `max_i` ≥ 9 in Listing 2).

```
// ...
while( some_condition )
  // the following block has to be finished within 1ms
  OSSS_RET( sc_time( 1000, SC_US ) )
{
  OSSS_EET( sc_time( 20, SC_US ) ) {
    // computation, that consumes 20µs
  }
  // estimate a data-dependent loop
  for( int i=0; i<max_i; ++i )
    OSSS_EET( sc_time( 100, SC_US ) ) {
      // loop body
  }
  if( my_condition ) {
    // communication only outside of EET blocks
    result = my_shared->get(); // see Listing 1
  }
} // end of RET block and loop
```

Listing 2. Example of estimated and required execution time annotations.

5. Simulation Results

An important factor for the feasibility of abstract software models is the simulation accuracy, they can achieve. Nonetheless, simulation performance is a critical factor during design space exploration as well. These goals are contradictory, as we will discuss in this section.

5.1. Accuracy and Performance

In order to ensure, that only one *Software Task* is active at a any time during the simulation, the different tasks have to be synchronised with the central RTOS abstraction, which then assigns the tasks according to its scheduling policy. Since the simulation time is usually handled by the SystemC kernel, this synchronisation requires an (at least implicit) call of `wait()`. Since this comes at the cost of an host system context switch (see Figure 5), the number of these synchronisations has to be reduced.

If the implementation of timing annotations immediately consumes the annotated delays, the granularity of these annotations must be kept quite coarse-grained to ensure good performance. As a result, control structures like data-dependent loops as shown in Listing 2 have to be estimated by their WCET, instead of taking the real number of iterations into account.

Another difficult situation arises from sporadic or conditional inter-task communication, as shown in Listing 2 as well. To annotate the surrounding basic blocks *and* to keep the number of annotations low, the annotations might

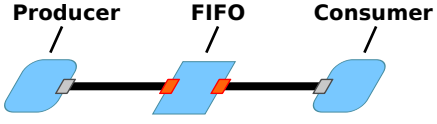

Figure 4. Producer/Consumer benchmark

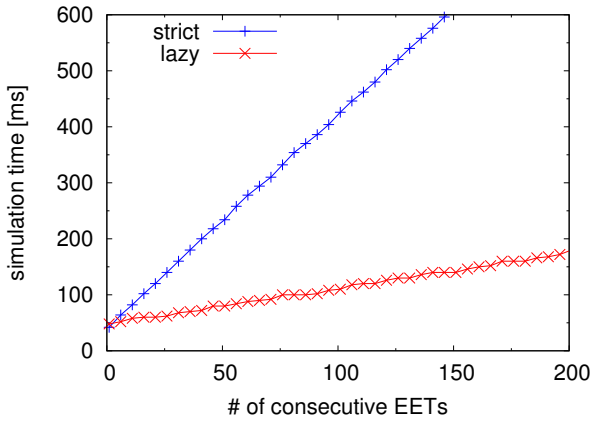

Figure 5. Impact of EET resolution on simulation time

have to be moved *entirely* before or after the communication primitive to keep the processor utilisation accurate. This results in a loss of accuracy with respect to access traces on the shared resources. Therefore, the trade-off between simulation time and modelling accuracy limits the observable effects, which might lead to wrong design decisions.

5.2. Lazy Synchronisation

If, on the other hand, the to-be-consumed processor time can be accumulated until a time synchronisation is explicitly *required*, a considerable speed-up is to be expected even in case of fine-grained annotations. Synchronisation between the abstract OS and the currently active task is required, whenever interaction with the OS or other components is requested. This especially includes inter-task communication and deadline validation.

Fortunately, communication in OSSS is expressed via *Shared Objects*, which easily enables the integration of such a execution time accumulation. We have implemented this alternative *lazy synchronisation* scheme as an optional feature of the current OSSS multi-tasking library model. Task synchronisation is delayed until a *Shared Object* call or the border of an `OSSS_RET()` block is encountered. By this, a task might logically pass multiple EET blocks without actually noticing any SystemC time advance. At the above mentioned synchronisation points, the accumulated delay is consumed at once. This accumulation of consecutive EET blocks without intermediate synchronisation is possible, since the observable behaviour – which only depends on the order and timestamp of inter-task communication – is not changed at all.

In Figure 5, these two scenarios have been compared. The benchmark scenario consists of a simple producer/-consumer scenario, where the producer pushes random numbers to the consumer through a FIFO *Shared Object* (see Figure 4). The tasks are scheduled with static priorities and overall constant estimated execution times, such that the FIFO channel is nearly empty (and thus blocks the consumer task) during the simulation. The `OSSS_EET()`

blocks in the producer are then increasingly split into small chunks, resulting in an increasing number of synchronisations in the "strict" scenario. The simulation consists of ten thousand FIFO calls (on both sides) and has been run on a Pentium D workstation with 2.8 GHz. Figure 5 shows, that an accumulated synchronisation leads to significantly faster execution with higher number of consecutive EETs.

6. Conclusion

In this paper, we presented an approach to modelling embedded software in OSSS. In comparison to the previously existing software modelling capabilities in OSSS, the current implementation introduces an abstract runtime system with support for multi-tasking and SW/SW inter-task communication. The modelling primitives like *Software Tasks* and *Shared Objects* are similar to the elements on the OSSS *Application Layer* (see Section 3) and abstract from the error-prone synchronisation primitives, the underlying RTOSs would provide.

The integrated RTOS abstraction includes different scheduling policies (preemptive *and* cooperative), periodic and continuous tasks, priorities, absolute and relative deadlines, without being targeted to a specific RTOS directly. As long as some locking primitive is available on the software target architecture, the OSSS software runtime can be mapped on this platform. A prototypical implementation on an existing RTOS will be published separately.

The HW/SW and SW/HW communication capabilities of OSSS (Section 3) are not yet fully integrated with the new software multi-tasking implementation. The communication refinement will follow the OSSS *Channel* approach, see [6, 8].

As we have shown in Section 5, the granularity of timing annotations and the resulting synchronisation overhead is an important factor for simulation speed. Therefore, the presented approach offers a flexible and simulation efficient way to specify estimated execution times. Moreover, the *lazy synchronisation* scheme further reduces the required SystemC kernel invocations by merging consecutive EETs between required synchronisations, which has been demonstrated with a benchmark in Section 5.2. This further improves the early exploration of software platform effects for systems modelled in OSSS.

References

[1] A. Burns and A. Wellings. *Concurrency in Ada*. Cambridge University Press, 1997.

[2] J. Chevalier, M. de Nanclas, L. Filion, O. Benny, M. Rondonneau, G. Bois, and E. M. Aboulhamid. A SystemC Refinement Methodology for Embedded Software. *IEEE Design & Test of Computers*, 23(2):148–158, Mar. 2006.

[3] V. Fernandez, F. Herrera, P. Sanchez, and E. Villar. *Embedded Software Generation from SystemC*, chapter 9, pages 247–272. Kluwer, Mar. 2003.

[4] Fossy – *Functional Oldenburg System Synthesiser*. `http://fossy.offis.de`.

[5] A. Gerstlauer, H. Yu, and D. Gajski. RTOS Modeling for System Level Design. In *Proceedings of Design, Automation and Test in Europe*, pages 47–58, 2003.

[6] C. Grabbe, K. Grüttner, H. Kleen, and T. Schubert. *OSSS - A Library for Synthesisable System Level Models in SystemC*, 2007. `http://icodes.offis.de`.

[7] E. Grimpe, W. Nebel, F. Oppenheimer, and T. Schubert. Object-Oriented Hardware Design and Synthesis based on SystemC 2.0. In *SystemC : Methodologies and Applications. - Boston u.a.: Kluwer*, pages 217–246, 2003.

[8] K. Grüttner, C. Grabbe, F. Oppenheimer, and W. Nebel. Object Oriented Design and Synthesis of Communication in Hardware-/Software Systems with OSSS. In *Proceedings of the SASIMI 2007*, Oct. 2007.

[9] Z. He, A. Mok, and C. Peng. Timed RTOS modeling for Embedded System Design. *Real Time and Embedded Technology and Applications Symposium, 2005. RTAS 2005. 11th IEEE*, pages 448–457, Mar. 2005.

[10] F. Herrera and E. Villar. A Framework for Embedded System Specification under Different Models of Computation in SystemC. In *Proceedings of the Design Automation Conference*, 2006.

[11] S. A. Huss and S. Klaus. Assessment of Real-Time Operating Systems Characteristics in Embedded Systems Design by SystemC models of RTOS Services. In *Proceedings of Design & Verification Conference and Exibition (DVCon'07)*, San Jose, USA, 2007.

[12] IEEE Standards Association ("IEEE-SA") Standards Board. *IEEE Std 1666-2005 Open SystemC Language Reference Manual*, 2005.

[13] R. Le Moigne, O. Pasquier, and J.-P. Calvez. A Generic RTOS Model for Real-time Systems Simulation with SystemC. *Design, Automation and Test in Europe Conference, 2004. Proceedings*, 3:82–87 Vol.3, 16-20 Feb. 2004.

[14] S. Mahadevan, M. Storgaard, J. Madsen, and K. Virk. ARTS: A System-Level Framework for Modeling MP-SoC Components and Analysis of their Causality. *13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 480–483, Sept. 2005.

[15] M. Müller, J. Gerlach, and W. Rosenstiel. Abstrakte Modellierung von Hardware/Software-Systemen unter Berücksichtigung von RTOS-Funktionalität (german). In *11. Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV'08)*, pages 21–30, Mar. 2008.

[16] Open SystemC Initiative. *SystemC™*. `http://www.systemc.org`.

[17] H. Posadas, F. Herrera, V. Fernandez, P. Sanchez, and E. Villar. Single Source Design Environment for Embedded Systems based on SystemC. *Transactions on Design Automation of Electronic Embedded Systems*, 9(4):293–312, Dec. 2004.

[18] M. Streubühr, J. Falk, C. Haubelt, J. Teich, R. Dorsch, and T. Schlipf. Task-Accurate Performance Modeling in SystemC for Real-Time Multi-Processor Architectures. In *Proceedings of the Design, Automation and Test in Europe Conference*, pages 480–481, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[19] Synthesis Working Group Members of Open SystemC Initiative. SystemC Synthesizable Subset, Draft 1.1.18. Whitepaper, Open SystemC Initiative (OSCI), Dec. 2004.

[20] H. Zabel and W. Müller. Präzises Interrupt Scheduling in abstrakten RTOS Modellen in SystemC (german). In *11. Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV'08)*, pages 31–39, Mar. 2008.