Assertion-Based Verification of RTOS Properties

Marcio F. S. Oliveira, Henning Zabel, Wolfgang Mueller
C-Lab, University of Paderborn
Paderborn, Germany
{marcio, henning, mueller}@c-lab.de

Abstract—Today, mobile and embedded real-time systems have to cope with the migration and allocation of multiple software tasks running on top of a real-time operating system (RTOS) residing on one or several processors. For scaling of each task set and processor configuration, instruction set simulation and worst case timing analysis are typically applied. This paper presents a complementary approach for the verification of RTOS properties based on an abstract RTOS-Model in SystemC. We apply IEEE P1850 PSL for which we present an approach and first experiences for the assertion-based verification of RTOS properties.

Keywords- real-time operating systems; verification; PSL

I. INTRODUCTION

Real-Time Operating Systems (RTOS) are required to provide a predictable platform for the execution of multiple software tasks on single microprocessors. In this context, operating system and task scheduling effects largely determine the overall software execution behavior, timing and quality metrics, such as performance or power. Therefore, each RTOS has to be accurately configured for the individual needs of the application and communication network. Today RTOS timing analysis is primarily performed by Worst Case Execution Time (WCET) and Response Time (WCRT) analysis [1] [2], by Instruction Set Simulation (ISS) [3] and by the simulation of an abstract canonical RTOS [4]. Due to the considerably high speed, SystemC simulation has turned out to become an attractive alternative.

In recent years, the principles of assertion-based verification and verification languages have been widely adopted for Register Transfer Level (RTL) verification such as SystemVerilog Assertions and IEEE P1850 PSL (Property Specification Language). In this paper, we demonstrate how to apply PSL for the verification of RTOS properties based on the simulation of our abstract canonical RTOS SystemC library aRTOS [4]. We introduce assertions to verify some basic RTOS properties which are selected to demonstrate basic principles in RTOS verification. We present different assertions such as, for the reachability of task states, arrival times, and scheduling orders. Our studies are based on the Mentor Graphics Questa [5] verification environment and give promising results in the context of abstract RTOS simulation.

The remainder of this paper is structured as follows. The next section discusses related works. Section III introduces basic principles of RTOS simulation and the applied RTOS library before the main approach is introduced in Section IV. Section V presents first evaluation results after which we close with a conclusion.

II. RELATED WORK

We are not aware of any approach which applies a verification language like SystemVerilog Assertions or PSL for the verification of RTOS properties. There are only very few approaches which apply PSL for general systems beyond hardware. Glazberg et al. [6] apply PSL for general system verification and Lahbib et al. [7] present an approach for system verification which is not triggered by a hardware clock. However, we can find multiple alternative approaches for RTOS simulation in SystemC to which our approach can be applied.

Desmet et al. [8] present an early approach for Operating System based Software Generation by means of an abstract Operating System API, i.e., SoCOS, on top of a C++ based System Level Design Environment. Gerstlauer et al. [9] introduce a canonical RTOS Model in SpecC. Their objective is to analyze real-time activities like preemption, interrupt handling and real time scheduling. Yu [10] presents a software synthesis for this model, which also takes hardware driver and Interrupt Service Routines into account. Recently, Schirner [11] increased the accuracy of that approach by means of Result Oriented Modeling (ROM).

Other works present RTOS Models in SystemC to evaluate timing behavior under consideration of scheduling strategies and task assignments to different processors [12][13][14].

Posadas et al. [15] published several articles on RTOS simulation. They introduce concepts in the context of their freely available SystemC RTOS library PERFidiX, which covers approximately 70% of the POSIX standard. Execution times are accumulated by overloading the operators of primitive data types with timing information.

Besides the analysis of real-time tasks, abstract RTOS - models support the software synthesis of embedded code from SystemC models [16] [17] [18]. This is mainly done by a step-wise replacement of communication with RTOS primitives and task assignments to processors.

Destro et al. [19] introduce a refinement for multiprocessor architectures in SystemC with a clear mapping from SystemC primitives to POSIX function calls. Starting from functional SystemC, first processor allocation and then hardware/software partitioning are performed. A final step maps SystemC to a co-simulation of hardware in SystemC and software running on top of an RTOS. After the mapping
hardware threads are executed by a specific SystemC compliant hardware scheduler. The mapping is basically done by the exchange of SystemC macros and definitions, i.e., replacing the settings in systemc.h by proprietary settings.

All models allow the analysis of task execution times and real-time properties, but they do not apply a dedicated hardware verification language. For this, we take PSL for assertion-based verification and combine it with our SystemC ARTOS library.

III. RTOS SIMULATION

Our abstract RTOS simulation at the task level is based on the partitioning of the application into hardware components and software tasks, including Interrupt Service Routines (ISR). Tasks and ISRs are divided into a sequence of time-annotated software segments. In order to accurately capture data-dependent delays, software segments are typically defined at the basic block level, though it is possible to partition them into more coarse-grain segments. Timing information of each segment is retrieved by back annotation from performance measurements or timing estimations through ISS or WCET analysis, respectively.

For abstract RTOS simulation our aRTOS library implements an abstract canonical RTOS model that is based on SystemC [20]. For RTOS task level simulation in SystemC, we model each software task and ISR as a SC_THREAD. The sequential part of each task and ISR is divided into timed segments. The time annotation of a segment is defined by a CONSUME_CPU_TIME function.

In general, the aRTOS library provides basic functions for software task and ISR (Interrupt Service Routine) synchronization, context switching, and scheduling. The actual RTOS context is defined and implemented by the sc_rtos_context. Each instance of the RTOS context represents a separate execution unit or processor core, respectively. It provides functions to register/deregister tasks and ISRs as well as managing their synchronization and performing RTOS context switches. Each RTOS context holds pointers to the task scheduler and the ISR scheduler, which select the next runnable task or ISR for sequential execution. The two schedulers are further coordinated by a main scheduling function of the RTOS context, which is sensitive to the schedule_event. The separation into two schedulers is necessary since scheduling strategies for tasks and interrupts may differ and thus have to be independent. Unlike software tasks, which are scheduled by the RTOS, ISRs are preemptively scheduled by the processor following the priorities of individual IRQs. Tasks and ISRs are implemented as SC_THREADS, which are spawned by the SC_RTOCCTOR constructor of an SC_RTOC_MODULE that extends the native SC_MODULE. Along the lines of the sc_module it is defined as an instance of the sc_rtos_module.

IV. VERIFICATION OF RTOS PROPERTIES

In the context of abstract RTOS simulation, we present a general architecture for assertion-based functional verification. The adopted solution applies PSL inheritance and provides an easy reuse of PSL formulae and specifications of the system timing information required for the assertions of RTOS properties. Fig. 1 illustrates the architecture.

The design is implemented in SystemC and makes use of the abstract RTOS API to define the tasks of the system. Considering a multiple processor system, one sc_rtos_context, e.g., sc_rtos_context_a and sc_rtos_context_b in Fig. 1, should be instantiated for each processor. The sc_rtos_context represents the context verified by PSL formulae and provides the information required by the vunits verification units through its ports for debugging and verification purpose: (i) the state of all tasks in the context; (ii) the task in execution; (iii) the scheduling event; and (iv) the pre-emption event. This information is observable by the verification units by the signals state, current_task, scheduling, preemption that are used in the definition of the PSL formulae.

In TLM verification, the communication between tasks performed through a bus structure, which can provide read and write events observable as signals reading and writing in its ports by the vunits. Additionally, it can provide specific information such as, the slot number of a TDMA-based (Time Division Multiple Access) FlexRay bus, for instance.

As an example, general PSL properties - namely check_context and check_channel in Fig. 1 - are specified in vunits, which are reused between multiple sc_rtos_context and sc_tdma_channel instances. Specific vunits - tdma_channel, context_a and context_b in Fig 1 - can be implemented, bound to the instances and inherited from the general vunits. Inside the implemented vunits and bound to the sc_rtos_context instances, two signals for each task representing the start time and finishing time can be specified and controlled in the PSL modeling layer. This article only shows some signals though more complex PSL formulae can be specified to verify additional properties, which may require other information. Signals and the task id are passed as parameters to the inherited properties in order to configure them for the specific assertion. Fig. 2 illustrates a sample vunit, i.e., context_a (Fig. 2, line 1), which inherits the general properties from check_context (Fig. 2, line 3), defines the procedural blocks that control the start_time and deadline signals (Fig. 2, lines 7-12), and asserts three properties start_time, finishing_time and unpreempted (Fig. 2, lines 15-19).

![Figure 1. Architecture for verification of abstract RTOS simulation using PSL formulae.](image)
The next four PSL properties define the verification of the RTOS context during the simulation. They are basic RTOS properties to demonstrate the basic concepts of RTOS verification.

The start_time property specifies when a schedulable task must start executing. The parameter t indicates which task must be verified. Herein, the boolean aevent is a signal, which indicates the specific start time event fired by the control in the vunit using the modelling layer.

```plaintext
start_time(const t; boolean aevent) = aevent -> (prev((state[t] == WAITING)) || prev((state[t] == READY))) && (current_task == t);
```

The finishing_time property defines the deadline until when a task must finish its execution. As in the previous formula, the parameter t indicates which task must be verified. Herein, the boolean aevent is a signal, which indicates the deadline event.

```plaintext
finishing_time(const task; boolean aevent) = (current_task == t) -> (prev(slot==s)) && (context==t) && writing;
```

The precedence property defines the dependence between executing tasks and can be used to check the proper scheduling order.

```plaintext
precedence(const task_1, task_2) = 
(\{ [*]; (state[t] == RUNNING); (current_task==v)[*]; (states[t] == WAITING)[*]; [ *]; (current_task == v)[*] \};
```

In some implementation there are functions which must be executed in atomic fashion, due to the correctness of an information or stringent time requirements. For this purpose the property unpreempted verifies if during a scheduling event (scheduling) the current execution of task t is not pre-empted by other tasks.

```plaintext
unpreempted (const task) = (scheduling && prev((current_task == t)) && (states[t] == READY)) -> (current_task==t) && !preemption;
```

During the implementation of embedded networked systems, such as multiple microcontroller units connected through a TDMA-based bus, the synchronization of RTOS scheduler and communication is a hard task. In order to help a designer to cope with this synchronization and to verify the correctness of the timing in the communication at abstract levels, additional PSL formulae can be specified as following.

```plaintext
no reading_in_slot(const t, s; boolean context) = (slot == s) -> !(context == t) && reading;
no writing_in_slot(const t, s; boolean context) = (slot == s) -> !(context == t) && writing;
```

Those two basic properties assert that one task t must not read/write in the time slot s reserved for the task to communicate in a TDMA-based bus. The signals reading and writing represents the reading and writing events fired by the bus.

```plaintext
reading_after_slot(const t, s; Boolean context) = ((reading)&&(context==t)->prev(slot==s);
```

In order to ensure the maximal synchronization between the scheduler and the communication, the reading_after_slot property defines that a task t must read just after the time reserved slot s finish, so that the task can read the most updated information available in the buffer.

```plaintext
writing_before_slot(const t,s; boolean context) = (prev(context==t) && (writing))->(slot==s);
```

Similarly, the writing_before_slot defines that task t must write the information before the time slot that is reserved for its communication, ensuring that the task will not waste a communication cycle, which may have a significant impact on communication performance.

V. EVALUATION

The present work is in progress and extends the application of PSL beyond RTL at task level RTOS simulation based on an abstract RTOS. Preliminary results were obtained adopting Mentor Graphics Questa 6.2, an integrated multi language simulator. In this tool the support of PSL assertions is limited to SystemVerilog/Verilog and VHDL, so that we have to bind vunits to SystemVerilog modules, providing the same interface as the SystemC modules to be verified. For this, SystemVerilog modules must be instantiated in the SystemC top module and connected to the original SystemC ones. However, it should be noted that the implementation of the presented approach may differ due to the applied tool.

Although the PSL standard does not delineate the timing in the clock operator, this construct is supported by EDA tools carried with a RTL semantic. In the context of abstract RTOS simulation, this semantic is not adequate and requires a new synchronization policy. Therefore, the defined PSL properties have to adapt PSL’s default clock to the scheduler event fired by sc_rtos_context and to reading and writing events fired by the sc_tdma_channel.

Considering different synchronization policies for our verification approach, two types of models were implemented. The first model uses a RTL clock which frequency is higher as the fastest event in the simulation in
order to allow the verification of all events during the simulation. The other one uses scheduling, reading, and writing events as a synchronization signal. In order to measure the overhead of the PSL verification in the abstract RTOS simulation and the gain of a more abstract synchronization policy, simulation of both types with and without assertion were performed.

The results indicate a time overhead of 41.80% when PSL assertions are performed in the simulation, which employs the RTL clock. A higher overhead of 45% is observed in the simulations without RTL clock. Moreover, following the expectation, the simulation without RTL clock is 37.65% faster than the simulation with the RTL clock as default synchronization policy. It is worth to note that the simulation with RTL clock and verified with PSL formulae is 56.33% slower than the simulation which replaces the RTL clock by scheduling, reading and writing events and verified with PSL too. Although the overhead of using PSL in abstract simulation is greater than the simulation of low level models, this approach focuses on the relevant information and exploits the benefits of the abstraction RTOS simulation as the final result is an improved and faster verification.

VI. CONCLUSIONS

In this paper, we presented an approach to apply IEEE P1850 PSL for the specification of RTOS properties for abstract RTOS simulation by our aRTOS SystemC library. We introduced some assertions to verify a specific set of RTOS properties. First studies show promising results so that we can keep the fast simulation times for abstract RTOS analysis. Current work focuses on the extension of rules for the verification of a wider set of RTOS properties and for the definition of a more generic set, which can be applied to different operating system libraries in order to arrive at more generic library elements, e.g., covering arbitrary OS states. Additional effort is currently spend on the investigation of adequate and efficient triggers for PSL synchronization.

ACKNOWLEDGEMENTS

The work described in this chapter was partly funded by the German Ministry of Education and Research (BMBF) in the context of the ITEA2 project TIMMO (ID01IS07002) and the ICT project COCONUT (ICT-217069).

REFERENCES