Verification of Real-Time Properties for Hardware-Dependent Software

Wolfgang Mueller, Marcio F. da S. Oliveira, Henning Zabel, Markus Becker
University of Paderborn/C-LAB, Paderborn, Germany
{wolfgang, marcio, henning, markus.becker}@c-lab.de

Abstract—Seamless HW/SW codesign flows support early verification of hardware and Hardware-dependent Software (HdS) like drivers, operating systems, and firmware. For early estimation and verification, the application of SystemC in combination with Instruction Set Simulators and Software Emulators like QEMU is widely accepted. In this article, we present an advanced design flow for HW, (RT)OS and HdS refinement and verification with focus on the transition from abstract RTOS verification to full system RTOS/HdS emulation. In the context of assertion-based verification, we introduce a set of generic real-time properties which can be reused and verified at different abstraction levels and discuss their application. The properties are presented by the means of IEEE standard PSL assertions which are applied for mixed SystemC/HdS models.

Keywords- Verification; PSL; SystemC; Real-Time Systems

I. INTRODUCTION

With increasing design complexity, the early consideration of software became a crucial factor in electronic systems design [12]. In this context, the configuration and scaling of the Hardware-dependent Software (HdS) as an important interface between application software and hardware became the focus of multiple investigations throughout the last years.

In order to deal with consideration of mixed HW/SW systems, additional methodologies and abstraction levels were introduced beyond Register Transfer Level (RTL). In 2003, [8] introduced the notion of Transaction Level Modeling (TLM). They address the abstraction of cycle-timed RTL via approximatively-timed TLM (bus arbitration) abstraction to untimed system specifications with respect to computation and communication. In parallel, [10][14] introduced clockless TLM abstractions to interface embedded SW and HW to capture logic operations on busses for master and slave communication. For the smooth transition from timed to untimed models, the levels of Programmer’s View with and without timing were introduced (PV & PV-T) which finally contributed to the OSCI TLM 1.0 standard [27]. Thereafter, PV-T was revised by the OSCI and refined to loosely- and approximately-timed coding styles as they can be found in TLM 2.0 [28]. As such, TLM provides a bus-centric abstraction for mixed HW/SW systems as they are sketched in Fig. 1. Models of different abstractions are connected to a TLM-based communication by so-called transactors, which convert the individual interfaces and protocols to the respective TLM protocol.

Embedded software is typically analyzed by formal timing analysis or by means of a virtual prototype. For static formal timing analysis, Worst Case Execution/Response Time (WCET/WCRT) analysis is applied on a frequent basis. WCRT analysis is typically based on an event stream abstraction, where the individual tasks in the model are activated by events [9][33]. The event stream abstraction takes composable event FIFOs at the components’ inputs and outputs for performance analysis. The underlying theory defines a workload for the individual tasks within a specific time interval, so that minimal and maximal distances between events can be determined, e.g., by the means of the sliding window technique [24]. This allows the analysis of over and under sampling effects in distributed systems, for instance.

For the implementation of a virtual system prototype, a system level language - today mainly SystemC - is applied in combination with an abstract canonical RTOS [13] and/or an Instruction Set Simulator [25]. Abstract RTOS models have the advantage that they provide a significantly faster simulation speed for time-based simulations on the costs of less accuracy in the simulation results. As such, significant speed-ups of up to 40,000x compared to ISS with simulation errors of less than 2% have been reported [31][36].

In the context of virtual prototypes, several approaches for different abstraction and refinement levels have been introduced through the last years. A layered approach was presented in [31], incrementally describing processor modeling with essential features of task mapping, dynamic scheduling, interrupt handler, low-level firmware and hardware interrupt handling. At the highest level, the application is running natively on the simulation host. At Task Level, an abstract RTOS model is introduced and processes are refined into tasks. In the proces-
A dedicated approach for the refinement from functional SystemC PV models to SystemC/ISS cosimulation is introduced in [11]. It supports smooth HW/SW partitioning by replacing SystemC threads with POSIX threads without code modifications with the help of their SC2OS library. However, the approach is limited to the functional evaluation of SW not considering (RTOS) properties. Therefore, it was extended in [2] to a four-level RTOS-aware TLM 2.0-based refinement that starts from functional untimed SystemC PV models. In a first step, the methodology applies SC2OS before models are ported to a canonical abstract RTOS, namely, aRTOS, which is finally replaced by the target (RT)OS running in the full system mode of the QEMU software emulator as given in Fig. 3. In the third step, aRTOS system calls for abstract RTOS simulation are introduced [35], which are replaced by system calls of the target (RT)OS in the final step.

In this paper, we first introduce the basic principles of QEMU and abstract RTOS simulation. Thereafter, based on the work in [2], we extend the previous approach and show how the transition from the abstract to the target RTOS can be smoothened even further before we discuss the specification and verification of real-time properties. The later is based on our work in [26] which introduced a basic set of PSL assertions for abstract RTOS simulation.

QEMU can operate in two modes: (i) user mode and (ii) full system mode. The user mode supports an efficient user space simulation of a single process on top of one process on the simulation host. In user mode, a system call raises an exception, which in turn invokes a corresponding Linux system call. The user mode maps the target memory directly to the host memory, so that it abstracts from any specific memory model. The QEMU full system mode includes an entire target platform with full I/O and kernel space access for operating system and driver simulation so that system calls can be directly executed.

### II. INSTRUCTION SET SIMULATION WITH QEMU

QEMU is a software emulator, which provides an instruction and register accurate CPU abstraction with the support of multiple platform, e.g., x86, ARM and PowerPC [4]. QEMU comes with an efficient dynamic binary translation based on a fetch/decode/execute cycle. The cycle applies an online compilation of instructions into a host Instruction Set Architecture (ISA). For this, target code is considered on Basic Block (BB) level, i.e., linear code segments until a final branch instruction. QEMU uses a dynamic code generator to translate BBs at runtime by concatenating precompiled host code segments, i.e., Translated Basic Blocks (TB). For faster execution, TBs are stored in a TB cache. Then, the major translation effort is just chaining TBs from the cache and patching instruction operands. Meanwhile several approaches like [2] apply QEMU for efficient Instruction Set Simulation in combination with SystemC. [16] additionally presents a QEMU extension for QEMU/SystemC cosimulation for time and power estimation also considering caching effects.

#### III. ABSTRACT RTOS SIMULATION

Several approaches for efficient RTOS simulation were introduced in the last years for early HW/SW estimation and verification. They can be roughly divided by the integration of a real (RTOS) API, like the POSIX-based PERFidiX library [30] and u-ITRON-based RTK-Spectron system [17], and approaches based on abstract RTOS models like [13]. The latter was originally developed in SpecC and reimplemented in SystemC [35] enhanced by a separate interrupt management and preemptive wait statements.

In SystemC, for HW/SW cosimulation, the application software and interrupt service routines are typically executed in SC_THREADS and divided into non-preemptive time/power-annotated code segments with interaction points between them. Interaction points are for the synchronization with the simulation kernel, e.g., for advancing the simulation time, for system calls, and for various communications. Time and power information are backannotated from ISS or WCET analysis which can be done at source code level [13][31][37] or at compiled/intermediate code level [18]. Considering the impact of compiler optimization, [37] also investigates automatic source code annotations and their traceability in the compiled code.

It is important to note, that the number of interaction points has an impact on the simulation speed as each interaction point implies a synchronization with the simulation kernel. For abstract RTOS simulation, typically, one interaction point is
assigned for each execution path. However, more coarse- or fine-grained segmentations are possible, if necessary.

As the advance of the simulation time between the interaction points is not preemptive, less interaction points may result in a lower accuracy of the simulation when just applying native (SystemC) wait statements. The processing of an interrupt by the simulator, for instance, always has to be postponed to the next wait statement in such a case. To significantly increase the accuracy of the simulation, several approaches were developed to replace the direct synchronization with the kernel by higher level constructs or an overloaded wait statement [30][32][35].

In our simulations, we apply our aRTOS SystemC library [35] which implements an abstract canonical RTOS model based on SystemC 2.1. The library provides basic functions for SW task and ISR (Interrupt Service Routine) synchronization, context switching, and scheduling. The actual RTOS context is defined and implemented by a sc_rtos_context module. Each instance of the RTOS context corresponds to a separate execution unit, i.e., processor core. It provides functions to register/deregister tasks and ISRs as well as managing their synchronization and performing RTOS context switches. SW tasks and ISRs are implemented as SC_THREADS. For a smooth synchronization and performing RTOS context switches, SW tasks and ISRs have to be implemented as SC_THREADS. For a smooth refinement with user mode QEMU emulation, we can apply Translated Blocks (TBs) segmentation with corresponding interaction points for time annotation. SW tasks are SC_THREADS executed in an SC_RRTOS_MODULE. As shown in Fig. 4, aRTOS implements an explicit state model with three basic states: waiting, ready, and running [7]. We later also refer to waiting as blocked and to ready as preempted to simplify our PSL assertions. Entering the blocked state means that the task was running and is waiting for a blocked resource for synchronization. A task enters preempted, when it was preempted and needs to get running again to complete.

![Figure 4. Different aRTOS States](image)

IV. ASSERTION-BASED VERIFICATION FOR HDS

The notion of Assertion-based verification (ABV) [5] is based on the declarative definition of systems properties by the means of functional assertions. Once defined, the formalized properties can be applied to different verification technologies, which are provided by the underlying verification tools like formal verification or simulation. Assertions are defined by a Hardware Verification Language (HVL) like the IEEE standards SystemVerilog [19], e [21], and PSL [20]. Meanwhile, large assertion libraries and methodological guidelines became available to supplement the tooling and standardization efforts, such as the Verification Methodology Manual (VMM) [6] and the Open Verification Methodology (OVM) [29]. As assertions typically define properties at the interface of a component, their main benefit is their massive reuse at different levels of abstractions. This makes them also potentially applicable for standard SW components like device drivers or general real-time properties as we will show later in this section.

A. Abstraction Levels for Assertion-based Verification

We apply our RTOS assertions along different refinement levels starting with the abstract RTOS simulation and ending with the full system mode QEMU software emulation as a final step. This approach is based on the flow introduced in Fig. 3 [2] and extended by user mode QEMU emulations for a more efficient transition between both levels. In this refinement, we take system calls as a clear interface between the application software (limited to user space access) and system operations with kernel space access for the execution of privileged instructions. As such, we can identify three RTOS refinement levels:

- **Abstract RTOS Model.** At this level, the SW has to be divided into segments separated by interaction points. An interaction point is either determined by (i) an abstract RTOS system call, (ii) a communication operation (e.g., bus access, I/O), and (iii) a synchronization with the simulation kernel for advancing the simulation time, where (ii) and (iii) should be implemented as specific system calls. For (iii) we can take QEMU Translated Blocks (TBs) for segmentation.

- **User Mode QEMU.** At this level, we can keep the SW code without modification as long as each interaction point can be implemented as a system call. Otherwise, remaining interaction points have to be replaced by a system calls or removed. For this, we have modified the QEMU kernel by retargeting the system call exception handling routine of QEMU to calls of the SystemC aRTOS library.

- **Full System Mode QEMU.** At this level, each abstract RTOS system call has to be refined to a native RTOS system call. In contrast to the previous refinements, this may require considerable effort to port system calls to the target operating system.

If adequately defined, we may apply the same assertions to all three levels under the condition that it is possible to insert explicit state and scheduling information into the code of the abstract and target operating system which just requires little modifications which at least presumes the availability of the (RT)OS source code, like it is the case with the different Linux variants. The refinements come s at the costs of increasing simulation times. Our experiments showed that, in the first step, User Mode QEMU approximately doubles the simulation time compared to aRTOS simulation. This little increase is due to the efficient of QEMU’s binary translation. From user mode to full system mode, we measured an increase between 5x - 150x, dependent on the number of synchronizations between SystemC and QEMU [3]. With less processor utilization, the difference between those levels can even be higher.

The next subsection gives a short introduction to PSL before we introduce the application of PSL for the verification of real-time properties in the context of the previous abstraction levels.
B. The Property Specification Language (PSL)

The IEEE P1850 Property Specification Language (PSL) [20] is composed of several layers which support the reuse of PSL statements. At the Boolean layer, Boolean expressions describe system state properties at a point in time. Temporal operators and sequence expressions (Temporal layer) are used to define the relationship between those states over the time. Regardless to the respective abstraction level, an EDA tool typically requires that the evaluation of assertions has to be triggered by a specific clock. As such, PSL properties must refer to the clock operator @ in order to define a Boolean clock expression. Therefore, the system state must be sampled in the clock cycle, where the clock expression is true.

At the Modeling layer, input stimuli, complex behavior and auxiliary logic can be defined by different hardware description languages (flavors) like Verilog and VHDL. At the Verification layer, so-called verification directives define instructions for individual verification tools, e.g., assert, assume, cover. On top of this, PSL properties are grouped into separate verification units (vmodule, vprop and vunits) which are bound to the specific component to be verified.

C. RTOS Verification Library (RVL)

For real-time property verification, we implemented an RTOS Verification Library (RVL) based on the principles of the Open Verification Library (OVL) [1]. The RVL provides so-called checkers with well-defined interfaces and PSL vunits bound to those checkers. Fig. 5 illustrates the general RVL verification architecture, the binding of PSL properties to checker modules and their instantiation in the virtual prototype. For this, checker modules are instantiated and connected via ports and signals to the given SystemC model under test. As an example, in Fig. 5, Context A Checker is an instance of a RVL checker module which verifies RTOS Context A.

The RVL is implemented along the basic concepts of [7] which identifies a set of basic properties for real-time tasks: arrival time, deadline, start time, finishing time, tardiness, tardiness and laxity.\(^1\) We have defined PSL assertions for those properties to be applied in the checker modules. Fig. 6 gives an overview of the corresponding checkers with their interface definitions.

Checkers are implemented for the verification of real-time tasks which are identified by the individual taskid. The first four ports define standard signals where pulse refers to a high level asynchronous event which triggers the evaluation of the

\(^1\) We ignore other properties like criticalness and value as we consider them as either redundant or as parameters for the scheduler.

property (see also next subsection). The other three signals refer to signals providing RTOS internal information: preemption, scheduling and state (of the tasks in the task set). assert_arrival_time verifies the point in time when a task becomes ready for execution. assert_start_time checks when a schedulable task must start executing, i.e., to become running. Here, the Boolean signal required_start_time indicates the specific start time event. assert_absolute_deadline (finishing time) defines the time a task must finish its execution. The assert_relative_deadline checker verifies the time window until a task must finish its execution. Here, the parameters required_start_time and deadline are signals which define the execution window. In order to support fault tolerance analysis and the design of soft real-time tasks, assert_max_tardiness can be used to verify if the time a task remains running after its deadline not exceeding a specific limit. Complementary to [7], assert_reach_state verifies if a task or an ISR reaches a specific state (rstate). The assert_precedence checker defines the dependence between two executing tasks and is used to verify a proper scheduling order. Some implementations require the atomic execution of two or more functions due to strict timing requirements. For this, the assert_no_preemption checker also verifies that the current execution of a task is not preempted by another one.

In real-time systems, time-triggered busses, like FlexRay, connect different communicating nodes and reserve communication slot for each individual node (or task). In this context, we can also define real-time checkers, which verify bus scheduling properties as given in Fig. 7.

---

Figure 5. Application of the RVL.

---

Figure 6. RVL Checkers

---

Figure 7. RVL Checkers for Time-Triggered Busses

assert_no_reading_in_slot#(taskId,slotId) (reading);  
assert_no_writing_in_slot#(taskId,slotId) (writing);  
assert_no_writing_before_slot#(taskId,slotId) (reading);

assert_writing_before_slot#(taskId,slotId) (writing);
available in the buffer. Similarly, \texttt{assert\_writing\_before\_slot} defines that task \texttt{taskId} must write into a time slot time slot before \texttt{slotId}.

To finally give one example for an assertion, Fig. 8 shows the definition of the \texttt{start\_time} property that is applied by the \texttt{assert\_start\_time} checker. This property verifies when a schedulable task must start executing. Parameter \texttt{t} indicates which task is verified. The Boolean \texttt{restart\_time} is a signal, which indicates the specific start time (rising edge) event. The property states that when \texttt{restart\_time} holds, it implies that the state of task \texttt{t.state[t]} must be \texttt{WAITING} or \texttt{READY} in the previous cycle (\texttt{prev}) and \texttt{RUNNING} in the current cycle (the cycle which the property is triggered).

\begin{verbatim}
start_time(const t; boolean restart_time) = 
restart_time -> (prev((state[t] == WAITING)) || 
prev((states[t]==READY)))&&(state[t]==RUNNING)
\end{verbatim}

Figure 8. PSL start\_time Property

D. Synchronization of RTOS Assertions

Assertions are typically applied in the context of cycle-accurate simulations and are presumed to be triggered by a low level high frequency clock. However, our application is specified for clockless TLM-based simulation, so that we need a dedicated asynchronous trigger, i.e., the \texttt{pulse}, for the evaluation of the checkers. The pulse is a combination of several events. For RTOS related properties, these are mainly (i) the state changes of tasks, (ii) scheduling events indicating when a task is scheduled by the (RT)OS scheduler, (iii) a preemption event indicating that one task has been preempted, (iv) additional user defined timers for the inspection of specific periodic or aperiodic points in time. Additionally, for real-time property verification of busses, at least \texttt{reading} and \texttt{writing} events have to be available.

For the impact of different triggers on the evaluation of the properties, let us consider the scheduling example of the task set in Fig. 9. That figure applies a RTL clock as a high-frequency clock. It can be easily seen by this example that an adequate pulse corrects sensor faults.

\begin{verbatim}
assert_start_time(const t; boolean rstart_time) = 
assert_start_time -> (prev((state[t] == WAITING)) || 
prev((states[t]==READY)))&&(state[t]==RUNNING)
\end{verbatim}

Figure 9. Three synchronization policies for PSL assertions

It can be easily seen by this example that an adequate pulse is mandatory for a correct and efficient evaluation of the assertions. This basically requires the emission of events from the RTOS task scheduler. However, as long as the (RT)OS source code is available, this comes at low costs as it only needs a few local modifications in the code of the abstract RTOS and the target RTOS.

V. APPLICATION RESULTS

In order to measure the detailed impact of the granularity of the pulse, we evaluated the presented approach by the verification of a realistic fuel injection module of an automotive case study with Mentor Graphics Quest 6.2. Fig. 10 presents the architecture with two SystemC modules implemented on top of the RTOS model: the \texttt{Engine} and \texttt{Fuel Controller}. The engine model is executed in a periodic 5ms task and provides sensor signals like \texttt{throttleAngle}, (engine) \texttt{speed}, manifold absolute pressure (map), and exhaust gas oxygen (ego). The fuel controller model is implemented by two 10ms periodic tasks. The first one calculates the fuel rate and the second one detects and corrects sensor faults.

We applied 3 different synchronization policies for this example: (i) an aperiodic high level pulse as it was introduced in the previous subsection, (ii) a low frequency clock determined by the minimum common divisor of the periodic task with a period of 1ms, and (iii) a high-frequency clock defined as 50% faster as the fastest simulation event with a period of 5us. The last two policies basically resemble two different alternatives of a clock based RTL synchronization.

Our studies have shown that the PSL checkers for TLM-based multi level verification adequately matches all verification points with a simulation overhead of not more than 40%. The simulation with the low frequency clock of the presented example was 19.72% faster but missed a significant amount of verification points. The high frequency clock increases simulation time of the example by 32.54% compared to the aperiodic pulse, and even then it was not able to match all verification points. We finally applied the clock of the target processor running at 25 MHz with a 4ns period, which guarantees that all verification points are matched. This simulation was 20.000x slower than the previous simulation runs. This clearly indicates that the latter is not a real alternative to reach a fast high level simulation.

VI. CONCLUSIONS

In this paper, we presented the basic principles of SystemC based simulation of Hds and an approach for RTOS refinement from an abstract RTOS model to full system mode QEMU emulation. Based on this refinement, we applied IEEE P1850 PSL to define a general set of real-time properties for RTOS tasks and time-triggered busses. In general, our results show that assertion-based simulation is well applicable for Hds
components and we can overcome the inefficient synchronization of current EDA tools which presumes a high frequency RTL clock. However, our results also show that the synchronization policy has to be carefully selected to run meaningful simulations. This and the definition of standard interfaces for assertion-based simulation at multiple levels certainly require more studies.

ACKNOWLEDGEMENTS

The work described herein was partly funded by EU through the COCONUT project (FP7-ICT-3217069), by the DFG through the SFB 614 (Self-Optimizing Systems for Mechanical Engineering), and by the German Ministry of Education and Research through the SANITAS (01IM088) and the VERDE projects (01S09012).

REFERENCES