HeroeS: Virtual Platform Driven Integration of Heterogeneous Software Components for Multi-Core Real-Time Architectures

Markus Becker
University of Paderborn/C-LAB
Paderborn, Germany
Email: beckerm@c-lab.de

Ulrich Kiffmeier
dSPACE GmbH
Paderborn, Germany
Email: ukiffmeier@dspace.de

Wolfgang Mueller
University of Paderborn/C-LAB
Paderborn, Germany
Email: wolfgang@c-lab.de

Abstract—This article presents the HeroeS virtual platform driven methodology for embedded multi-core and real-time SW design. The methodology’s focus is on early integration, testing and performance estimation of heterogeneous SW stacks, i.e., SW components and layers at mixed abstraction levels and/or targeting different instruction sets. We take into account current system-level methodologies such as Transaction Level Modeling (TLM) and Real-Time Operating System (RTOS) modeling. For this, a SystemC virtual platform framework is presented combining state of the art simulation techniques according to the proposed methodology. This includes host-compiled target SW abstraction, abstract RTOS and Hardware Abstraction Layer (HAL) models in SystemC, extended QEMU user and system mode emulation and TLM 2.0 bus models. Efficient but Yet accurate performance estimates can be provided through static and dynamic annotation. We apply binary mutation testing, i.e., a test assessment and improvement approach for instruction level SW testing. Our approach was investigated by prototypical integration into a commercial AUTOSAR environment. Experimental results were obtained by an automotive case study: a fault-tolerant fuel injection control system, which is part of an in-car network.

I. INTRODUCTION

Today’s embedded systems increasingly include networks of multi-core CPUs running more and more SW with real-time constraints. In some embedded application domains, such as mobile computing or automotive industry, SW has already replaced HW as the key driver for innovation. However, HW greatly impacts SW in terms of performance. Thus, SW development is tightly coupled with the HW especially at lower levels of the software stack, which are referred to as Hardware-dependent Software (HdS). Virtual platforms enable early SW development by providing execution environments based on more or less abstract models before the actual HW is available. In order to manage the ever increasing platform complexity TLM methodologies were introduced raising the level of abstraction, thereby, elevating HW/SW co-design and verification to Electronic System-Level (ESL). However, current approaches such as OSCI TLM 2.0 are still too much focused on HW. Though, the advent of RTOS modeling approaches in SystemC partially closed the SW methodology gap, there remains a lack of guidance for applying the different methodologies and libraries in a unified design framework providing smooth HW/SW corefinement.

Embedded SW engineering has mainly become layer and component oriented enabling reuse across projects and companies, e.g., in Commercial Off-The-Shelf (COTS) driven deployment flows. Such components are typically subject to intellectual property protection. Thus, source code is not available at the integration stage. Moreover, hardware platforms are often shipped with SW components, e.g., standard libraries, device drivers and boot firmware. Such SW is typically provided in formats targeting a certain Instruction Set Architecture (ISA), e.g., assembler code, linkable object code and binary images. Thus, engineers are frequently confronted with the integration of heterogeneous SW components. For this, practical design environments are required addressing more flexibility in terms of supporting early integration and estimation of such SW stacks.

In this article we propose the HeroeS methodology focusing on advanced models for heterogeneous SW in the context of current system-level methodologies such as TLM and RTOS modeling. Our methodology is supported by a virtual platform framework enabling integration of heterogeneous SW components at different abstraction levels. We consider early performance and real-time estimation techniques addressing speed vs. accuracy trade-off. Integration testing is covered by advanced mutation analysis techniques which apply to both source level and instruction level SW. The implemented virtual platform framework is based on SystemC [1] utilizing state of the art techniques for simulation acceleration. This includes techniques such as host-compiled target SW abstraction, extended QEMU user mode and full system emulation, performance modeling through static and dynamic annotation of SW segments, abstract RTOS and HAL models and TLM 2.0 based system bus abstraction.

We investigate the applicability of the HeroeS methodology in the context of automotive SW design flows. For this, a prototypical integration into a commercial AUTOSAR tool chain was achieved. Industrial example application: a distributed motor management SW which is part of an in-car network.

The remainder of the article is organized as follows. In Section II we give an overview to current system level methodologies. In Section III we introduce the HeroeS methodology. In Section IV we describe a virtual platform framework according to the presented methodology. In Section V we consider AUTOSAR as a case study for applying the HeroeS approach to automotive domain. In Section VI we discuss related work in the field of system level SW modeling and analysis. Finally, Section VII concludes our paper giving an outlook to future work.
II. ELECTRONIC SYSTEM LEVEL DESIGN

In order to deal with consideration of complex mixed HW/SW systems, system-level methodologies and abstraction levels were introduced beyond Register Transfer Level (RTL). In 2003, [2] introduced the notion of Transaction Level Modeling (TLM). They address the abstraction of cycle-timed RTL via approximately-timed TLM (bus arbitration) abstraction to untimed system specifications with respect to computation and communication. In parallel, [3][4] introduced clockless TLM abstractions to interface embedded SW and HW to capture logic operations on buses for master and slave communication. For the smooth transition from timed to untimed models, the levels of Programmers View with and without timing were introduced (PV & PV-T) which finally contributed to the OSCI TLM 1.0 standard [5]. 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 [6]. Thus, the term TLM is somehow ambiguous as different definitions were given by individual research groups. We refer to TLM as the general concept of starting system design from a combined HW/SW model of processes communicating via read and write primitives accessing channels such as FIFOs. Whenever we use the term TLM 2.0 we refer to the more specific approach provided by the Open SystemC Initiative (OSCI) [1] that focuses mainly on abstract modeling of memory mapped system buses. For this, they provide a library and coding styles based on concepts such as target and initiator ports, blocking and non-blocking interfaces, the quantum keeper, the generic payload and Direct Memory Interface (DMI). In order to connect with models of different abstraction levels so-called transactors are required, which convert the individual interfaces and protocols to the respective TLM protocol.

A. Processor and SW modeling

In former times of HW/SW codesign unified modeling stopped after the HW/SW partitioning step by continuing SW development in cosimulated Instruction Set Simulators (ISS). In the context of system-level design, additional abstraction levels for processor and SW modeling have been introduced through recent years referred to as (abstract) RTOS modeling. Such approaches are typically based on the introduction of SW schedulers and RTOS APIs in a system-level design language—today mainly in SystemC.

Schirner et al. presented a layered approach in [8], 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 processor models at the Firmware (FW) and TLM refinement steps, hardware abstraction (HAL) and processor hardware layers are introduced. The FW and TLM level add models of the external bus communication and the interrupt handling chain on the software and hardware side, respectively. Finally, a Bus-Functional Model (BFM) of the processor includes pin- and cycle-accurate models of the bus interfaces and the protocol state machines driving and sampling the external wires.

III. HETEROGENEOUS SW-CENTRIC SYSTEM MODELING

Fig. 1 shows the common organization of a layered software stack as it can be found in embedded systems. Such SW stacks are divided into layers of the application software, middleware and the Hardware-dependent Software (HdS) which can be further divided into a Hardware Abstraction Layer (HAL) and the portable part of the system software, i.e., the RTOS, communication protocols, boot firmware and device drivers. We propose the advanced HeroeS design methodology in order to deal with the integration of such embedded SW stacks which are highly heterogeneous by nature.

The starting point of our methodology (see Fig. 2) is an Application SW model (I. ASM), i.e., a SW partition of a system specification model. We assume the application to be defined by source code, e.g., C/C++ code either written by hand or automatically derived from higher level models such as state flow models in MATLAB/Simulink or UML. For heterogeneous SW modeling, we propose four intermediate refinement models decoupling code refinement from interface refinement, namely, Source Level Task Model (II. STM), Target Level Task Model (III. TTM), Source Level System Software Model (IV. SSM) and Target Level System Software Model (V. TSM). For this, we employ abstract RTOS modeling further separating task modeling and system software modeling steps which can be modeled at both the source code level and the target instruction level. By decoupling also the refinement of individual CPUs or RTOS tasks a high degree of modeling flexibility can be achieved. Finally, the target specific SW Implementation Model (VI. SIM) contains all SW layers for running on real hardware.

Boxes in the background of Fig. 2 depict the relationship of our methodology to transaction-level modeling as introduced in Section II. Our focus is on SW modeling and refinement. However, comodeling of HW and SW is essential to our methodology. For this, we consider TLM as the HW/SW interface throughout the refinement process. Though, some address related information can be partially derived by models L-V., such as the frequency and locality of host or virtual memory accesses, full benefits of OSCI TLM 2.0 can be exploited only with the SIM which includes register accurate access to physical addresses through the definition of a HAL.

![Common organization of layered software stacks by Ecker et al. [7].](image-url)
We do not consider performance modeling as explicit part of the SW refinement since performance is estimated by the virtual platform. Nevertheless, as indicated by the z-axis in Fig. 2 we identified four general modeling techniques which can be applied for SW performance estimation depending on the SW model and the available details of the HW. More details on individual methods will follow in subsequent sections.

A. Code and Interface Refinement

Code and interface refinement consists of four major steps (a.-d.). Each refinement step comprises several smaller design decisions. In order to provide more flexibility in terms of integrating instruction level target SW we consider code refinement and interface refinement as orthogonal steps. Thus, (b.) system software modeling and (c.) target instruction set architecture modeling can be carried out independently. The four refinement steps are as follows:

a. Task modeling requires mapping of SW processes to tasks and assigning tasks to single-core or multi-core scheduling contexts. Each task has a set of assigned properties such as periodic vs. aperiodic activation and preemptive vs. non-preemptive execution. Each scheduling context can have a user-defined scheduling policy such as fixed priorities, dynamic priorities and/or time slicing. Primitive calls such as read/write or send/receive have to be refined to RTOS and communication APIs. This can be automated/abstracted by the generation of a middleware/adapter layer such as the Run-Time Environment (RTE) layer approach in AUTOSAR [9]. The introduction of APIs can be stepwise starting from a canonic RTOS API refining towards standard APIs, such as POSIX [10] or OSEK/AUTOSAR. Multi-core programming support can be introduced on top of dedicated middleware APIs such as OpenMP [11] or MCAPI [12]. RTOS and communication models are abstract at this point as the actual system services implementation remains undefined.

b. System software modeling requires the definition of system services which are typically based on a HAL API. Such services include the portable part of the RTOS, higher layers of the communication stacks and second level device drivers. At this point the abstract RTOS model must be refined to an actual RTOS implementation, e.g., the task activation mechanisms and task schedulers. For this, HW interrupts such as timers or I/O requests need to be introduced and according Interrupt Service Routines (ISR) must be defined. Communication APIs need to be refined according to the system topology, e.g., shared memory, inter-process communication, I/O or network communication. As the system SW layer interfaces with the HW through APIs the SW can be still defined in high level languages such as C. Thus, there is no dependence to modeling details of the target architecture and the physical memory layout.

c. Target instruction set architecture modeling requires the definition of the target ISA, e.g., ARM or PowerPC, and compilation of the source code to processor specific object and binary code. This implicates the selection of a compiler tool chain and compiler flags such as the specific instruction set and code optimization flags. Moreover, application binary interface (ABI) agreements are decided, i.e., calling conventions, byte ordering, padding and alignment. As such, user space memory layout is defined by the compiler/linker introducing virtual address space.

d. Hardware abstraction layer modeling requires the definition of the HAL according to the actual physical memory layout of the system/memory bus. This includes the definition of the low level assembler routines such as HW initializations, the RTOS context switch and other routines accessing HW registers directly, e.g., clock, timer, interrupt and power configuration. Resource allocation might require SW refinements of the higher layers according to the refined HW platform model. A binary image of the software stack can be derived in order to move to cycle-accurate ISS/RTL models or real hardware for more precise performance and real-time verification. However, all design decisions concerning the SW stack design have been taken at this point. Thus, the result can be considered as SW implementation model targeting a real HW platform.

B. Performance Modeling

SW performance can be modeled at different accuracy levels according to the available details of the HW model. In
general, we distinguish four approaches for SW performance modeling: (i) untimed (sometimes referred to as zero-timed), (ii) static execution time annotation, (iii) dynamic execution time estimation and (iv) bus transaction modeling. The most abstract approach is untimed or zero-timed modeling which is be applied when there are no details of the HW platform at all. Here, we assume an idealized processor executing software with infinite speed. Virtual time is approaching through discrete events used to block/trigger SW processes. However, the SW execution itself does not have any execution time (i.e., zero-time). This method can be used to investigate the effects of causal relationship between SW and time-stamped events but not for estimating performance of the SW being executed on a certain HW.

A simple method for performance estimation is to annotate software segments, such as instructions, compiler basic blocks, functions or tasks, by static execution delays. Such delays can be derived by the specification in terms of worst-case considerations or actual delays can be analyzed in advance by some static model of the HW platform using analysis tools like AbsInt aiT [13]. However, the accuracy of static annotation might be insufficient due to dynamic execution delays which cannot be resolved off-line. Here, dynamic estimation can provide more accuracy through abstract models of the processor core, e.g., an ALU timing model for data-dependent delays. Depending on the system complexity performance is heavily influenced by the memory subsystem resulting in more or less deterministic SW timing. Modeling the memory delay as a constant overhead per access can be sufficient for deterministic platforms such as used for hard real-time applications. However, modern mobile platforms are highly dynamic since they are equipped with shared memories, hierarchical caches, pipelines and branch predictors. In such a case, accurate estimations require more sophisticated models simulating the sequences of memory transactions. In order to avoid the overhead of cycle-accurate bus modeling TLM 2.0 coding styles Loosely-Timed (LT) or Approximately-Timed (AT) are used to abstract from pin-accurate models and to make synchronization among the bus components less tight. This provides a good accuracy vs. speed trade-off for early virtual platforms. Performance modeling techniques apply to the proposed HeroeS models as depicted by Table I.

<table>
<thead>
<tr>
<th>Untimed (zero-timed)</th>
<th>ASM</th>
<th>STM</th>
<th>SSM</th>
<th>TLM</th>
<th>TSM</th>
<th>SIM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static execution time annotation</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Dynamic execution time estimation</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>TLM 2.0</td>
<td>Loosely-timed</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>bus</td>
<td>Approximately-timed</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### C. Component Integration Testing

The integration of components to a system often reveals unforeseeable failures which cannot be avoided by thorough unit and component testing alone. This can be for instance due to incompatible interface specification. Thus, additional testing is required at the integration stage which can be a very challenging task when dealing with heterogeneous SW stacks. In order to reach sufficient test coverage integration testing might require sophisticated test patterns which have to be different compared to component level testing. Such test patterns must be able to propagate test stimuli from the interfaces of the integrated Design Under Test (DUT) through interconnected components and stacked SW layers. Here, mutation based testing approaches combined with Automatic Test Pattern Generation (ATPG) approaches turned out to be valuable tools for providing high quality test environments. The basic principle of mutation based testing is to insert small failures into the DUT which are likely to be coupled with real faults. Such faults are typically applied to source code and mutants are derived by compilation. Each mutant is executed with the available set of test cases. A mutant is said to be killed in case a deviation of the output w.r.t. the original DUT was detected. A mutation score is computed in order to provide a quality metric for the current test environment. The more mutants were killed the better the test environment is rated. ATPG methods, such as random and/or constraint based approaches, can be applied in case the test environment reveals an insufficient quality.

While white box testing methods apply well to source level SW they are not suitable for instruction level SW such as binaries. In contrast, black box testing is limited w.r.t. ATPG as there is no insight into the DUT. We provide a method for mutation testing at the instruction level utilizing target specific fault models. Our method can be applied to the unmodified binary under test without assuming source code access or any knowledge of the binary’s behavior. Instruction level mutation testing comprises four major steps: (i) binary analysis, (ii) automatic test pattern generation, (iii) binary mutation testing and (iv) evaluation. We apply static control flow and data flow analysis in order to derive a constraint annotated control flow graph (CFG) which serves as input model for a SAT solving. Mutation tables are derived by a target-specific fault model and the annotated CFGs. The actual application of the mutation table is performed at run-time. For this, we extended the QEMU binary translator flow to inject and detect faults while the binary under test is executed. Testing results are back annotated to the CFG model in order to compute additional test cases. A combined formal/heuristic approach is implemented by randomly solving path constraints in order to increase probability of code and path coverage which correlates also with the mutation score. In case the achieved mutation score indicates an insufficient test quality steps (ii)-(iv) can be repeated. More details on this method can be found in [14][15]. SW mutation testing applies to HeroeS models as depicted by Table II.

### IV. Virtual Platform Framework

In this section we present a virtual platform framework according to the presented HeroeS design methodology. The platform’s backplane is a SystemC kernel for event driven simulation. We utilize different SystemC libraries to achieve efficient execution environments for each of the proposed SW models. For ASM simulation, the application SW is mapped to plain SystemC. In STM, SSM, TTM and TSM, lower layers of the SW stack are abstracted by RTOS and HAL models in SystemC which are replaced stepwise through source level and instruction level production code. Source level target SW is abstracted in ASM, STMs and SSMs through
host-compilation and native execution. Here, performance can be modeled through static binary code analysis and back annotation to source code. We extended QEMU to support mutation and dynamic performance estimation for instruction level SW which we refer to as XEMU [15]. Wrappers for XEMU were developed for interfacing with SystemC in order to simulate TTM, TSMs and SIMs. TLM 2.0 bus models can be connected to the memory interface of XEMU system mode for SIM simulation. Fig. 3 shows the mapping of the Heroes methodology to the SystemC virtual platform framework.

A. SystemC RTOS and HAL Models

In SystemC, concurrency modeling is achieved by means of a cooperative scheduler implementing pseudo parallelization. In order to yield control to the simulation kernel a SystemC thread (SC_THREAD) has to explicitly invoke wait blocking for an event or an amount of virtual time. The semantics of a native SC_THREAD do not apply to SW modeling since SW threads are inherently preemptive due to CPU interrupts. Thus, our abstract RTOS and HAL models in SystemC provide an additional synchronization layer on top of the SystemC scheduler for modeling of preemptive SW processes. For this, we introduce two special SystemC module types SC_RTLS_MODULE and SC_HAL_MODULE. These modules define APIs for abstract RTOS and HAL services. A dedicated function CONSUME_CPU_TIME is provided in order to account a SW segment’s execution time by implementing an interruptible wait statement.

Tasks and cores are modeled by connection of SC_THREADS to RTOS contexts. Each RTOS context can be assigned to a user-defined or common scheduling policy such as fixed priorities, EDF or Round-Robin. In addition to a generic RTOS API the model provides standard OS interfaces such as POSIX or OSEK/AUTOSAR. The SC_HAL_MODULE provides system service modeling on top of a HAL API with primitives for process creation, context switching and interrupt management. An ISR is modeled by an SC_THREAD which must be connected to a HW event and an RTOS context. Dedicated ISR schedulers can be defined according to the interrupt policy of the HW platform. More details on this method can be found in [16].

B. Host-Compiled Target SW Abstraction

For the native execution of host-compiled target SW we provide a wrapper for linking shared object code to the SystemC model. In order to abstract from the wrapping mechanism we defined a common interface which is shared by source level and instruction level wrappers. The wrapped SW can be triggered by SystemC events through control API functions such as run(MyRoutine). Callbacks coming from wrapped SW are forwarded to SystemC APIs. For this, a function call is represented by a structure syscall_t which wraps the callback data in a target independent format. Callbacks are also used for piggybacking accumulated execution delay in order to synchronize locally decoupled time with the global SystemC time through calling the CONSUME_CPU_TIME primitive.

Static Execution Time Annotation: At the source level target SW execution time can be modeled by instrumenting the host-compiled SW. For this, code has to be segmented into time annotated behaviors such as tasks or functions. We apply segmentation to the level of source code branches in order to accurately capture the control flow of the application such as loops or conditional statements. For this, a C grammar has been developed to place preprocessor marks at the source code spanning a CFG. The marks can be replaced to instrument the source code by execution delays or cycle count accumulation statements according to the transition between two source code marks. Such values can be approximated, measured or analyzed statically. We implemented a tool chain for back annotation of static timing analysis. For this, source code marks are replaced by volatile declared assembler labels for target compilation. Such labels can be used to identify linear instruction level SW segments according to the source code CFG. Best case and worst case execution time (BCET/WCET) for segments can be derived in order to be back annotated to source code. The approach can provide sufficient accuracy for early estimates. However, the accuracy depends on the target platform’s complexity and the target compiler as mapping of the instruction level and source level CFGs is challenging when the compiler applies advanced optimization techniques such as function in-lining or loop unrolling. Details on this method can be found in [17].

Traditional cycle-accurate and binary code interpreting ISS turned out to be a performance bottleneck in fast virtual platforms. QEMU is an open source SW emulator implementing Dynamic Binary Translation (DBT) technique. Unlike static binary translators only code encountered at runtime is considered avoiding unnecessary translation overhead. In contrast to traditional ISS, DBT is performed at compiler basic block level, i.e., linear code segments until a final branch instruction. Moreover, translated blocks (TB) are buffered in a translation cache in order to provide execution speed close to native execution by avoiding redundant translations. QEMU is suitable for fast functional CPU modeling when there is no need for a detailed model of the CPU’s micro architecture. Besides x86 QEMU supports many different embedded systems architectures such as ARM, PowerPC, SPARC, MIPS or Microblaze. In general, QEMU operates in two emulation modes: user mode and system mode. The user mode provides CPU emulation for a single user program on top of a Linux kernel. QEMU (full) system mode provides emulation of an entire target system including CPU cores, system bus and I/O in order to run a complete software stack.
i.e., including boot firmware, operating system and HAL. QEMU can be considered as state of the art concerning fast ISS. However, QEMU does not naturally provide performance estimates for the executed target SW. We extended QEMU’s binary translator for dynamic timing estimation and run-time code mutation (which we refer to as XEMU [15]).

We integrate both the user mode and the system mode of XEMU according to interface refinement of the SW models. For this, we implemented two SystemC wrappers. The user mode wrapper connects XEMU with SystemC using a common SW wrapper interface which we also use to connect to host-compiled target SW wrapper. As such, functions of the wrapped SW stacks can be triggered by SystemC. Trapped API calls coming from wrapped SW stacks are forwarded to the RTOS, HAL and TLM APIs in SystemC. Callbacks trapped by XEMU user mode need special care as the wrapped instruction level code is typically compiled for a different target ISA. Thus, the XEMU user mode wrapper requires a specific target to host adapter dealing with binary interface issues such as calling conventions and byte ordering. Moreover, pointer arguments need to be converted from target to host address space. For this, the adapter needs to subtract the base address of the emulated target memory which is mapped to the host’s address space using an offset.

**XEMU for Time Estimation and Mutation Testing:** We applied two extensions to QEMU in order to provide time estimation (XEMU-T) and mutation testing (XEMU-M) for instruction level target code. For this, we modified the binary translator to instrument the generated back end code. XEMU-T annotates code blocks by static cycle counts according to the CPU timing specification. In order to capture data-dependent delays evaluation code is inserted into translated blocks which then accumulates additional CPU cycles during execution. XEMU-M emulates small SW faults at run-time by modifying translated code blocks according to a mutation table which is generated by code analysis in advance. More details on this method can be found in [14].

**C. Synchronization**

Synchronization of time can be either loose or tight impacting the speed vs. accuracy trade-off. Different schemes can be applied according to the SW abstraction levels. The proposed performance models rely on time annotated functional segments. As such, we do not apply any clock cycle synchronization. SW processes must call `CONSUME_CPU_TIME` in order to synchronize locally accumulated execution delays which is limited by the granularity of functional segments. For source level SW models processes can be segmented down to the level of linear control flow segments according to source code. For instruction level SW models segments can be smaller such as compiler basic blocks or even single processor instructions. However, for the sake of performance we apply a more coarse-grained synchronization scheme [18]. As such, local time is decoupled by accumulating time annotations. In order to simulate the application’s data flow in a correct causal relationship a synchronization must be performed before each communication. Synchronization in TLM 2.0 depends on the coding style of the bus model. For LT buses, synchronization with XEMU system mode can be applied on the level of compiler basic blocks in order to benefit from efficient dynamic binary translation. Thus, we do not propose application of a fixed time quantum as the XEMU execution is driven by translated basic blocks which can have varying execution time estimates. However, the maximum size of a basic block in terms of the instruction count can be limited resulting in less simulation performance. For AT buses, each memory transaction needs to be synchronized according to the bus protocol. For this, XEMU system mode has to be executed in single instruction mode providing more accuracy on the cost of a major performance degradation.

**V. Case Study: Automotive Software Design**

**A. AUTOSAR**

The AUTomotive Open System ARchitecture (AUTOSAR) [9] initiative is a partnership of leading automotive companies founded in 2003. The automotive industry is widely organized as chains of suppliers and integrators such as tool vendors, automotive HW/SW suppliers and car manufacturers (OEM). In order to increase interoperability among companies the initiative developed a methodology based on an XML meta model and a well-defined software architecture for Electronic Control Units (ECU). Application SW models in AUTOSAR are strictly component-oriented. Atomic components can be organized hierarchically and connected via ports and interfaces using client/server or sender/receiver patterns. The internal behavior of atomic components is modeled through runnables, data accesses and events. This view is referred to as the Virtual Functional Bus (VFB) which is defined upon the Run-Time Environment (RTE) middleware layer API. The work in [19] describes a mapping of the VFB view to SystemC/TLM concepts which matches with the ASM concepts of the HeroeS methodology and framework. By defining the RTE layer the application SW is deployed to task sets running on a network of ECUs. This requires mapping of RTE concepts to the RTOS and comm. API which we refer to as the Task Modeling step. The ECU architecture furthermore defines an interface for the RTE layer the Microcontroller Abstraction Layer (MCAL) which the Basic Software (BSW), i.e., the system software, is defined upon. Following the HeroeS methodology, such layers can be introduced stepwise. We investigated the mapping of AUTOSAR and HeroeS methodologies by means of a commercial AUTOSAR design environment. For this, we considered the integration of the HeroeS virtual platform framework into the AUTOSAR tool chain provided by dSPACE in order to simulate virtual ECU (V-ECU) networks.

**B. Example Application: Fuel Injection Controller**

For experimental results we investigated the SW of a fault-tolerant fuel injection controller, which is a part of a motor management system. The fuel injection controller is modeled by a SW component (SWC) which is internally composed of atomic components for sensor correction and for fuel rate computation. The controller requires four sensor signals, such as throttle angle and engine speed. The sensor correction computation calculates a fuel injection rate. The controller’s output signals are transmitted to a further component which drives a combi instrument. For testing purpose, the application comes with another component wrapping a physical model of the engine which generates stimuli in a closed-loop manner. The generated application C code consists of 10 functions with a total complexity of 3397 lines of code. According to the AUTOSAR methodology the application was mapped to
a network of three ECUs connected by a CAN bus. We used dSPACE SystemDesk [20] to generate the ECU production code with a total complexity of approx. 30,000 lines.

C. AUTOSAR Tool Chain Integration

Fig. 4 shows the prototypical integration of the HeroeS framework into the dSPACE tool chain. For this, an interface for the XCP protocol and A2L (ASAP2) descriptions was implemented. Furthermore, we implemented an interface for virtual ECU descriptions (V-ECU) in order to import SW stacks generated by SystemDesk. The platform can be operated in two modes: (i) fast simulation mode and (ii) interactive mode. Mode (i) executes as fast as possible for testing and estimating different system configurations. Mode (ii) uses host-adaptive speed control to interface with an experimentation environment for interactive control and measurement of variables. Fig. 5 shows a snapshot of the running tool setup. Tasks, signals and events can be traced in VCD format through SystemC tracing facilities.

Fig. 4. Prototypical integration into dSPACE tool chain.

VI. RELATED WORK

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 [21][22]. 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 [23].

For the implementation of a virtual system prototype, a system level language - today mainly SystemC - is applied in combination with an abstract RTOS [24] and/or an ISS [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 [26]. The combination of a SystemC RTOS scheduler and interpretive ISS was proposed in [27] first. Later, more hybrid approaches for source code and instruction level target SW were proposed such as HVP [28], HySim [29] and HyCos [30]. However, the authors do not investigate the employment of advanced emulation techniques such as QEMU user mode or full system emulation. The combination of QEMU full system emulator and SystemC/TLM 2.0 for early SW development was proposed by GreenSocs [31] first focusing on functional aspects. Later, more advanced approaches were introduced, e.g. TIMA RABBITS [32], addressing efficient investigations of non-functional properties such as timing and power. Some other virtual platforms also rely on SystemC/TLM 2.0 and similar DBT technology such as OVPsim [33] and Synopsys CoMET [34]. However, they are either commercial or source code is partially closed. Thus, they are not as suitable for academic research. To the best of our knowledge we are not aware of any approach that evaluates the combination of QEMU/DBT technology and SystemC models at different HW/SW and SW/SW interfaces such as the system bus, the HAL API and the RTOS API.

The HeroeS approach proposed in this article continues our former works w.r.t. system-level RTOS-aware HW/SW modeling flows. A dedicated approach for the refinement from functional SystemC PV models to SystemC/ISS cosimulation was introduced by [35]. 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, we extended the flow in [36] 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 RTOS running in the full system mode of the QEMU software emulator. In the third step, aRTOS system calls for abstract RTOS simulation are introduced [16], which are replaced by system calls of the target RTOS in the final step. Later, we refined the flow by combining QEMU user mode emulation with SystemC at the RTOS API [18].

VII. CONCLUSION

We presented the novel HeroeS methodology and virtual platform framework for early SW integration and performance estimation of heterogeneous SW stacks for embedded multi-core and real-time architectures. The implemented virtual platform framework is based on SystemC employing state of the art abstraction techniques for generating fast simulation models by means of TLM, RTOS/HAL models and extended QEMU user/system mode emulation. Our approach was evaluated by prototypical integration into a commercial AUTOSAR design environment. Our future work will focus on scalability investigations w.r.t performance vs. accuracy trade-off that can be provided by the proposed platform models.

VIII. ACKNOWLEDGEMENTS

This work was partly funded by the German Ministry of Education and Research (BMBF) through the project SANTITAS (16M3088I), the DFG SFB 614, the ITEA2 projects VERDE (01S09012) and TIMMO-2-USE (01IS10034).

REFERENCES