Formal Methods for Integration of Automotive Software

Marek Jersak, Kai Richter, Rolf Ernst
Technische Universität Braunschweig
Institut für Datentechnik und Kommunikationsnetze (IDA)
D-38106 Braunschweig, Germany
{jersak, richter, ernst}@ida.ing.tu-bs.de

Jörn-Christian Braam, Zheng-Yu Jiang, Fabian Wolf
Volkswagen AG
Aggregateelektronik (EAOL)
Brieffach 1687, D-38436 Wolfsburg, Germany
{joern-christian.braam, zheng-yu.jiang, fabian.wolf}@volkswagen.de

Abstract

Novel functionality, configurability and higher efficiency in automotive systems require sophisticated embedded software, as well as distributed software development between manufacturers and control unit suppliers. However, at least for engine control units, there exists today no well-defined software integration process that satisfies all key requirements of automotive manufacturers.

We propose a methodology for safe integration of automotive software functions where required performance information is exchanged while each partner's IP is protected. We claim that in principle performance requirements and constraints (timing, memory consumption) for each software component and for the complete ECU can be formally validated, and believe that ultimately such formal analysis will be required for legal certification of an ECU.

1. Introduction

Embedded automotive software plays a key role in increased efficiency of today's automotive system functions, in the ability to compose and configure those functions, and in the development of novel services integrating different automotive subsystems. Automotive software runs on electronic control units (ECUs), which are specialized programmable platforms with a real-time operating system (RTOS) and domain-specific basic software, e.g. for engine control. Different software components are supplied by different manufacturers and have to be integrated. This raises the need for an efficient, secure and certifiable software integration process, in particular for safety-critical functions.

The functional software design including validation is largely mastered today through a well-defined process including sophisticated test strategies [5]. However, integration of software functions from different vendors on the automotive platform requires validation of the integrated system's performance. Here, non-functional system properties, in particular timing and memory consumption are the dominant issues. At least for engine control units, there exists today no well-defined software integration process that satisfies all key requirements of automotive manufacturers.

In this paper, we focus on a methodology and the resulting flow of information that should be established between the automotive software manufacturer, different ECU suppliers, RTOS supplier, and system integrator to establish a certifiable software integration flow for engine electronics that a) allows to exchange key performance information between the individual partners and b) at the same time protects each partner's intellectual property (IP). We highlight the shortcomings of todays approach, in particular in the area of timing analysis. We claim that in principle techniques and required information are available for formal timing analysis of automotive software at all levels, including individual tasks, RTOS overhead, single ECUs and networked ECUs.

The remainder of the paper is organized as follows. In the following section, we summarize the current practice in automotive software design and integration, and highlight the shortcomings from the automotive manufacturer's perspective. In Sec. 3, we outline key methodological issues that have to be solved for a safe integration and certification flow. This is detailed in Sec. 4, where we focus on RTOS configuration, communication conventions and memory budgeting, and in Sec. 5, where the key issue, namely timing analysis, is discussed. We conclude with a summary and an outlook on future work.

2. Current Practice in Software Integration

The software of sophisticated programmable automotive ECU is usually composed of three layers. The lowest, system layer consists of the RTOS and basic I/O, typically based on the OSEK [7] automotive RTOS standard. The next higher level is so called 'basic software'. It consists of functions that are already specific to the role of the ECU, but either simple and designed for more sophisticated functions to build upon them, or standard automotive function that the automotive OEM (original equipment manufacturer) does not want to design herself. Generally speaking, with properly calibrated parameter values, an ECU with RTOS and basic software is a working control unit for its specific automotive domain. However, OEM-specific performance characteristics may not be optimal. The system-layer is typically out-sourced to an RTOS vendor by the ECU supplier. The ECU supplier then adds basic functions and ships the ECU with RTOS and basic software to the automotive manufacturer.

The highest layer are sophisticated control functions where the automotive OEM uses her vehicle-specific know-how to extend and thus improve the basic software, and to add new features. The automotive OEM also designs distributed vehicle functions, e.g. adaptive cruise-control, which span several ECUs. Sophisticated control and vehicle functions present an opportunity for automotive product differentiation, while ECUs, RTOS and basic functions differentiate the suppliers. Each partner therefore has a vested interest in IP protection.

From the automotive manufacturer's perspective, a software integration flow is preferable where the vehicle function does not have to be exposed to the supplier, and where the OEM herself can perform integration for rapid design-space exploration or even for a production ECU. This can only be supported in a scenario where software functions are exchanged and integrated using object codes. The crucial requirement here is that the integrated software must meet the stringent safety requirements for an automotive system in a certifiable way.

A key problem that remains largely unsolved is the validation of performance bounds for each software component, and formal methods to calculate conservative performance bounds for the whole ECU, or even a network of ECUs. A suitable validation methodology is currently not in place.

Therefore, in many cases, software can only be integrated by the ECU supplier, who then requires a detailed design specification from the OEM for executable code-generation and optimization. While this approach is no more formal, at least the ECU supplier can claim that the fully integrated ECU software passed all its tests before shipping the ECU to the automotive OEM. This potentially leads to lengthy and costly iteration cycles between OEM and ECU supplier, with little opportunity for design-space exploration.

3. Proposed Solution

We are interested in a software integration flow for automotive ECUs where sophisticated control and vehicle functions can be integrated as black-box (object-code) components. The automotive OEM should be able to perform the integration herself for rapid prototyping, design space exploration and performance validation. The final integration can still be left to the ECU supplier, based on validated performance numbers that the automotive OEM provides. The details of the integration and certification flow have to be determined between the automotive partners and are beyond the scope of this paper.

We focus instead on the key methodological issues that have to be solved. On one hand, the methodology must allow safe integration of software functions without exposing IP. On the other hand, performance requirements and constraints (timing, memory consumption) for each software component and the complete ECU have to be formally validated, to be able to certify the ECU. Those software components include not only the OEM-provided functions, but also the basic software functions from the ECU vendor and the RTOS.

A possible integration and certification flow which highlights these issues is shown in Fig. 1. Is consists of a well defined OSEK configuration, adherence to software interfaces, performance analysis and characterization of all entities involved, and a performance analysis of the complete system. Partners exchange properly characterized black-box components. The required characterization is described in corresponding agreements.

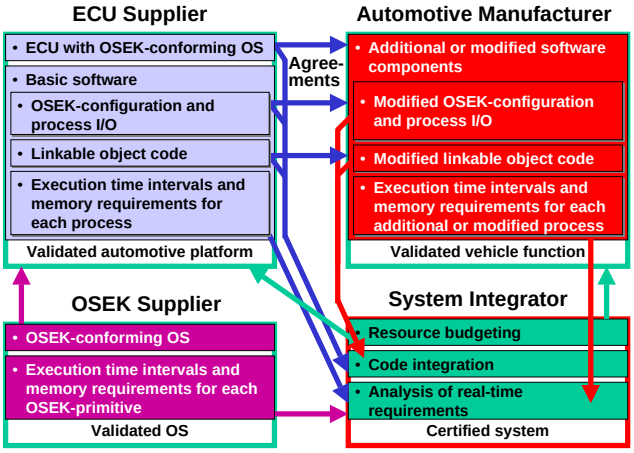

Figure 1. Proposed integration and certification flow

4. Software Integration

In this section, we address RTOS configuration, communication conventions and memory budgeting for a safe software integration flow.

4.1. RTOS Configuration

Basic RTOS configuration is performed by the ECU provider. In particular, this includes the number of available priorities and timer periods. In automotive applications, there are typically only a few periods required, e.g. 10ms, 20ms and 100ms. Additionally, tasks may run synchronously with engine RPM. The developer of a software component must specify its required period and its location in the sequence of functions performed within that period.

In OSEK, which is an RTOS-standard widely used in the automotive industry [7], the configuration can be performed in the 'OSEK implementation language' (OIL [11]). Tools then build C or object files that capture the RTOS configuration, and insert calls to the individual functions in appropriate places. With the proper tool-chain, integration can also be performed by the automotive OEM.

In our experiments, we used ERCOSEK [2], an extension of OSEK. In ERCOSEK, code is structured into tasks and processes. An ERCOSEK task consists of a sequence of processes which are typically written in C. Each task is assigned a priority and scheduled by the RTOS. Processes inside each task are executed sequentially. Tasks can either be activated periodically with fixed periods using a timetable mechanism, or dynamically using an alarm mechanism. The latter is a very flexible but expensive solution in terms of memory requirements and execution time. For the communication between tasks and processes global variables or state messages can be used. Messages have the advantage that context-switches cannot lead to data inconsistencies. The mechanism is based on copying global to local variables, and vice versa, at the beginning and end of a process, respectively, while interrupts are disabled. Furthermore it is possible to protect exclusive resources using a priority ceiling protocol, to avoid deadlocks.

The configuration of ERCOSEK was realized using the tool ESCAPE [3]. ESCAPE consists of an ANSI-C checker, pre-compiler, ESCAPE parser and a code optimizer. ESCAPE reads a configuration file that is based on OIL, with some extensions for features that go beyond the OSEK standard. The configuration statements are translated by the ESCAPE parser into ANSI-C code. The individual software components called from this code can be pre-compiled, black-box components. To facilitate object-code linkage, the usage of the same compiler seems imperative.

4.2. Communication Conventions and Memory Budgeting

Black-box components with standard software interfaces are needed to satisfy IP-protection. At the same time, validation, as well as modularity and flexibility requirements have to be met. Furthermore, interfaces have to be specific enough that any integrator can combine software modules into a complete ECU function.

IP protection and modularity are goals that can be combined if read-access are hidden and write-accesses are open. An open write access generally does not uncover IP. For example, the fact that a functions in an engine ECU influences the amount of fuel injected gives away little information about the function's internals. However, the variables read by the function can yield valuable insight into the sophistication of the function.

From an integration perspective, hidden write-accesses make integration very difficult since it becomes impossible to determine when a value is potentially changed, and thus how functions should be ordered. Hidden read-accesses pose no problem from this perspective.

The ECU vendor, in his role a main integrator, provides a list of all pre-defined communication variables to the SW-component providers. Some of these may be globally available, some may be exclusive to a subset of SW-component providers. Furthermore, the software integrator budgets and assigns memory available to each SW-component provider, separated into memory for code, local data, and private communication variables.

For each software component, its provider specifies the memory actually used, and actual write-accesses performed to shared variables. If the ECU exhibits integration problems, then each SW-component's adherence to its specification can be checked on the assembly-code level using a debugger. While this is tedious, it allows a certification authority to determine which component is at fault. An alternative may be to use hardware-based memory protection, if it is supported. Reasonable levels of granularity for memory access tables (e.g. vendor, function), and the overhead incurred at each level, still have to be investigated. An analysis of access violation at compile or link-time, on the other hand, seems overly complex, and can be easily tricked, e.g. with hard-to-analyze pointer operations.

Another interesting issue is the trade-off between performance and flexibility as a result of basic software granularity. Communication between SW-components is only possible at component boundaries (see communication mechanisms described in Sec. 4.1). While a fine basic software granularity allows the OEM to augment, replace or introduce new functions at very precise locations, overhead is incurred at every component boundary. On the other hand, coarse basic software may have to be modified more frequently by the ECU vendor to expose interfaces that the OEM requires.

5. Timing Analysis

The second, more complex set of integration issues, deals with software component and ECU performance, in particular timing. Simulation-based techniques for timing validation are increasingly unreliable with growing application and architecture complexity. Therefore, formal timing analysis techniques which consider conservative min-max behavioral intervals are becoming more and more attractive as an alternative or supplement to simulation. We expect that ultimately, certification will only be possible using a combination of agreed-upon test-patterns and formal techniques.

In formal analysis, the goal is to obtain conservative execution time intervals for every system function. Two approaches can be safely combined:

- Timing of (sub)functions that is input-data *in*dependent, or for which worst- and best-case input data is known, can be measured.
- These lower-level timing intervals can then be used to perform min-max calculations for higher-level functions where the timing is input-data dependent and so complex that worst- and best-case input data is not known.

Apart from conservative performance numbers, timing analysis also yields better system understanding, e.g. through visualization of worst-case scenarios. It is then possible to modify specific system parameters to assess their impact on system performance. It is also possible to determine the available headroom above the calculated worst-case, to estimate how much additional functionality could be integrated without violating timing constraints.

In the following we indicate that a formal approach is consistently applicable at all levels (single process, RTOS, single ECU and networked ECUs), thus opening the door to formal timing analysis for the certification of automotive software.

5.1. Single Process Analysis

Formal single process timing analysis determines the minimum and maximum execution time of one activation of a single process assuming an exclusive resource. Recent analysis approaches, e.g. [8], first determine execution time intervals for basic blocks. Using an integer linear programming (ILP) solver, they then find a shortest and a longest path through the process based on basic block execution counts and cost, leading to an execution time interval for the whole process. Often, the designer has to bound loops and exclude infeasible paths to tighten the process-level execution time intervals.

For architectures with pipelines and caches, execution time intervals for basic blocks can be rather pessimistic because empty pipelines or cache flushes have to be assumed. First commercial tools like AbsInt [4] consider the static cache behavior to tighten interval bounds. However, the inherent pessimism when considering each basic-block individually remains.

If input data independent sequences of basic blocks (process segments) and the resulting address access sequence are considered, then established cache tracing techniques can be applied . This approach significantly reduces the problem size of previous approaches based on transition graphs for single basic blocks. For input data dependent control structures between process segments, data flow analysis can be applied to predict cache line contents. Cache analysis results can then be accounted for when calculating execution time intervals for process segments. The process-level execution time intervals are then determined using the known technique from [8] for the remaining data dependent control structures between process segments instead of basic blocks. This methodology can lead to much tighter execution time intervals compared to previous approaches, as has been shown with the prototype tool SYMTA/P [17].

To evaluate the applicability of SYMTA/P, we used it to identify segments in C-code taken from a real-world engine control function. Each segment boundary was then instrumented with a trigger point [18, 17], in this case an inline-assembly store-bit instruction. The target platform was a TriCore evaluation board with Tasking EDE. Using appropriate stimuli, we executed each segment and recorded the setting of a special bit with a logic state analyzer (LSA). With this approach, we were able to obtain clock-cycle-accurate measurements for each segment. These numbers, together with path information, were then fed into an ILP solver, to obtain minimum an maximum execution times for the example code.

The applicability of SYMTA/P was encouraging. However, inter-procedural analysis has to be improved, and a few minor issues have to be resolved before larger experiments are conducted.

5.2. RTOS Analysis

Apart from influencing the timing of individual tasks through scheduling, the RTOS itself may consume a considerable amount of processor time. The RTOS has to activate tasks, make scheduling decisions, and to terminate tasks. Additionally, low-level device drivers, e. g. system timers and I/O, require processor time. Typical RTOS primitives are described e.g. in [1]. The most important RTOS influences are: task or context switching including start/preemption/resumption/termination of tasks; and general OS overhead, including periodic timer interrupts and some house-keeping functions. For formal timing analysis to work, these numbers need to be considered in a conservative way.

One one hand, we need to determine execution time intervals for each RTOS primitive, and their dependency on the number of tasks scheduled by the RTOS. The second interesting question concerns patterns in the execution of RTOS primitives, in order to derive the worst- and best-case RTOS overhead for task response times.

Ideally, this information would be provided by the RTOS vendor. She has detailed knowledge about the internal behavior of the RTOS, allowing her to perform appropriate analyses that cover all corner cases. However, it is virtually impossible to provide numbers for all combinations of targets, compilers, libraries, etc. Alternatively, the RTOS vendor could provide test patterns that the integrator can run on her own target and in her own development environment to obtain the required worst- and best-case values. Some OS vendors have taken a step in that direction, e.g. [10].

In our case, we did not have sufficient information available, and thus decided to measure the influence of RTOS primitives ourselves. We performed our measurements by

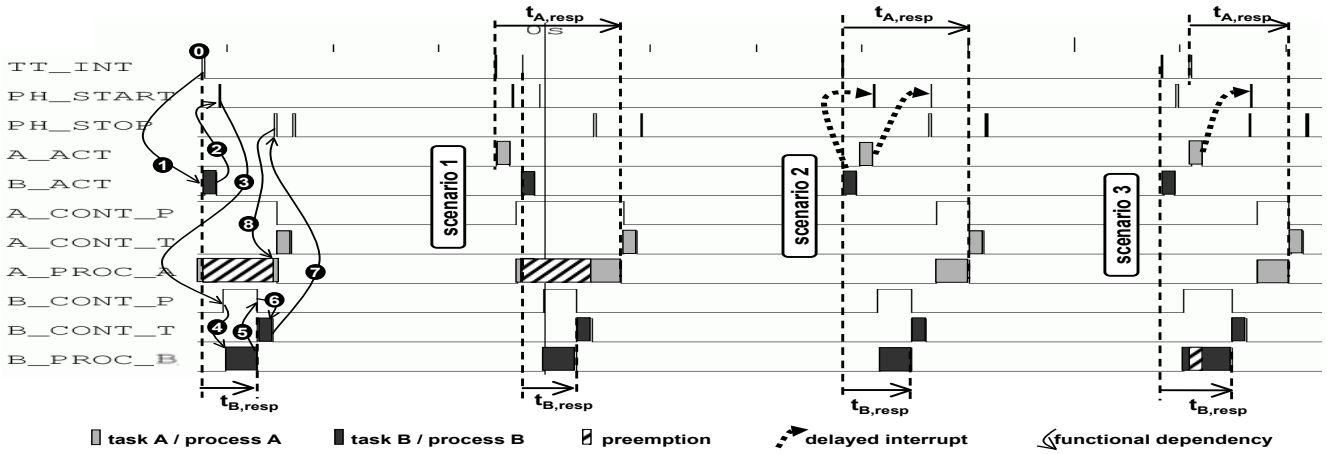

Figure 2. Waveform: Measurement of two Preemptive Tasks A and B

instrumenting accessible RTOS functions, using the LSA-based approach described in Sec. 5.1. This is not ideal, since it is tedious work and does not guarantee corner-case coverage. The instrumentation approach described in Sec. 5.1 is limited to the C-code level. Fortunately, the ESCAPE tool chain (Sec. 4.1) generates RTOS configuration functions in C, which then call the corresponding RTOS functions (object code). The C functions provide hooks for instrumentation. We inserted code that generates unique port signals before and after each RTOS function call. We measured:

TT_INT TimeTable Interrupt: Executed whenever the time table needs to be evaluated to start a new task.

PH PreemptionHandler: Executed whenever a task has to preempt the actually running one.

X_ACT ActivateTask X: Executed whenever a task is activated (i.e. ready for execution).

X_CON_P ProcessContainer: Each task X has a process container that subsequently calls all processes within this task.

X_CON_T TerminateTask: Executed after task X has finished.

X_PROC_Y Process: This is the actual user process Y within task X.

Our first set of measurements indicates two things. First, for a given RTOS configuration and a given task set, the execution time of RTOS primitives varies little from measurement to measurement. This supports our claim that the RTOS vendor should be able to appropriately characterize the timing of each RTOS primitive, or alternatively provide suitable tests such that the user can determine timing intervals for her architecture herself. Second, as expected we have observed that RTOS overhead prolonging task response times varies from measurement to measurement. However, patterns in the execution of RTOS primitives can be found and analyzed. Therefore, we believe that it is generally possible to calculate conservative bounds on RTOS overhead for task response time calculation. The RTOS vendor should be able to provide rules how this overhead can be calculated.

A sample pattern of RTOS primitives that is traversed for one execution of one task is shown in the LSA printout in Fig. 2. A system timer generates a time-table interrupt (step 0). This selects and activates the corresponding tasks by executing the ActivateTask function (step 1). Although not visible in the source code, based on the LSA output it seems that this in turn generates a software interrupt which then starts the preemption handler. Since all tasks seem to be using the same preemption handler, we distinguish between start and stop using two different I/O signals in order to record recursive interrupts. The preemption handler calls the dispatcher, which seems to start the task, i. e. the process container (step 3), which finally executes the process (step 4). After completion, control returns to the process container (step 5), and since there is only one process in the task, the TerminateTask function is executed (step 6). Finally, the preemption handler finishes (step 7) and a preempted task is resumed (step 8).

In Fig. 2, we also show three different scenarios for task interference. Depending on the order in which tasks A and B are activated by TT_INT (A before B in scenario 1, A and B at the same time in scenario 2, A after B in scenario 3), the tasks experience different delays. Task A is the lower-priority task, task B the higher-priority task. One interesting observation is that preemption handlers seem to have the same priority as their corresponding task, since task A's preemption handler is delayed until the completion of task B in scenarios B and C. Another interesting observation is that the ActivateTask functions have a higher priority than all tasks. Therefore, the activation of lower-priority task A can delay or preempt the execution of higher-priority task B (scenarios 2 and 3, respectively). These observations show that it is important to model the RTOS at such a detailed level to reliably calculate its influence of task response times. The observed patterns also indicate that it generally should be possible to provide formulas for these

calculations.

5.3. Single and Networked ECU Analysis

Single ECU analysis builds upon single-process performance (Sec. 5.1) and RTOS performance results (Sec. 5.2). It employs formal analysis techniques to determine schedulability for all processes running on the ECU based on known process-activation patterns.

Our system consists of independent, periodic tasks without data-dependencies. The tasks are scheduled by a fixed-priority scheduler, and each task's deadline is equal to its period. As a first approximation, such a system can be analyzed using the static-priority preemptive analysis developed by Liu and Layland [9]. We can account for the high priority level of the ActivateTask functions (see previous section) by treating them as independent periodic tasks at a very high priority level. Periodic timer interrupts can be captured the same way. The TerminateTask function and the preemption handlers can be added to a task's internal (core) execution time.

ERCOSEK additionally implements the priority ceiling protocol [15] to avoid deadlocks when accessing exclusive resources, and to reduce priority inversion influences. These influences can also be taken into account during analysis using the techniques presented in [15].

At this point, we are still confined to single ECU applications. Heterogeneous distributed architectures require more complex analysis techniques, such as developed in [13, 14, 12]. We intend to use event models to couple analysis techniques for single ECUs and busses following the ideas in [13, 14].

6. Conclusion

In this paper we focused on key methodological issues that have to be solved for certifiable integration of automotive software components, where required performance information can be exchanged while each partner's IP is protected. The proposed component characterization and agreements should allow both the automotive OEM and the ECU vendor to assume the integrator role. The agreements include conventions for RTOS configuration, task communication and memory budgeting. The main issue, however, is performance validation, in particular the validation of timing. We presented methods for performance characterization of software functions and the RTOS. We expect that ultimately each partner will have to characterize his software components appropriately. Fist promising steps towards performance characterization of her RTOS have been taken by at least one OS-vendor [16]. Timing analysis of the complete ECU can then be performed by any integrator using appropriate scheduling analysis techniques.

We are currently working on performance analysis of a single ECU using the techniques described in this paper. We are planing to extend our work to networked ECUs using the techniques from [13, 14]. Also, the improvement in analysis bound tightness when process and system contexts are considered has to be investigated [6].

References

[1] G. Buttazzo. *Real-Time Computing Systems - Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers, 2002.

[2] ETAS. *ERCOSEK Automotive Real-Time Operating System*. http://www.etas.info/html/products/ec/ercosek/en_products_ec_ercosek_index.php.

[3] ETAS. *ESCAPE Reference Guide*. http://www.etas.info/download/ec_ercosek_rg_escape_en.pdf.

[4] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Journal of Real-Time Systems, Special Issue on Timing Analysis and Validation for Real-Time Systems*, pages 131–181, Nov. 1999.

[5] ISO. TR 15504 information technology - software process assessment 'spice'. Technical report, ISO IEC, 1998.

[6] M. Jersak, K. Richter, R. Henia, R. Ernst, and F. Slomka. Transformation of SDL specifications for system-level timing analysis. In *Tenth International Symposium on Hardware/Software Codesign (CODES'02)*, Estes Park, Colorado, USA, May 2002.

[7] J. Lemieux. *Programming in the OSEK/VDX Environment*. CMP Books, 2001.

[8] Y. S. Li and S. Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic Publishers, 1999.

[9] C. L. Liu and J. W. Layland. Scheduling algorithm for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20, 1973.

[10] LiveDevices Inc. *Realogy Real-Time Architect Overview*. http://www.livedevices.com/realtime.shtml.

[11] OSEK/VXD. *OIL: OSEK Implementation Language*, version 2.3 edition, Sept. 2001.

[12] T. Pop, P. Eles, and Z. Peng. Holistic scheduling and analysis of mixed time/event-triggered distributed embedded systems. In *Tenth International Symposium on Hardware/Software Codesign (CODES'02)*, Estes Park, Colorado, USA, May 2002.

[13] K. Richter and R. Ernst. Event model interfaces for heterogeneous system analysis. In *Proc. of Design, Automation and Test in Europe Conference (DATE'02)*, Paris, France, Mar. 2002.

[14] K. Richter, D. Ziegenbein, M. Jersak, and R. Ernst. Model composition for scheduling analysis in platform design. In *Proceeding 39th Design Automation Conference*, New Orleans, USA, June 2002.

[15] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), Sept. 1990.

[16] K. Tindell, H. Kopetz, F. Wolf, and R. Ernst. Safe automotive software development. In *Proc. Design, Automation and Test in Europe (DATE'03)*, Munich, Germany, Mar. 2003.

[17] F. Wolf. *Behavioral Intervals in Embedded Software*. Kluwer Academic Publishers, 2002.

[18] F. Wolf, J. Kruse, and R. Ernst. Segment-Wise Timing and Power Measurement in Software Emulation. In *Proc. IEEE/ACM Design, Automation and Test in Europe Conference (DATE'01), Designers' Forum*, Munich, Germany, Mar. 2001.