

RT-Druid

*A tool for architecture-level design
of embedded systems*

White Paper



Evidence S.r.l.

<http://www.evidence.eu.com>

info@evidence.eu.com

This document is copyright © 2004 Evidence S.r.l.

Information and images contained within this document are copyright and the property of Evidence S.r.l. All trademarks are hereby acknowledged to be the properties of their respective owners. The information, text and graphics contained in this document are provided for information purposes only by Evidence S.r.l. Evidence S.r.l. does not warrant the accuracy, or completeness of the information, text, and other items contained in this document.

Matlab, Simulink, Mathworks are registered trademarks of Matworks Inc. Microsoft, Windows are registered trademarks of Microsoft Inc. Java is a registered trademark of Sun Microsystems. All other trademarks used are properties of their respective owners.

1 Introduction

1.1 Background

Embedded control systems development is about producing systems or sub-systems according to a set of specifications dictating their required (physics) properties. The output of the design stage is a set of models that describe the system at different levels of abstraction. Properties and constraints are captured by mathematical formalisms.

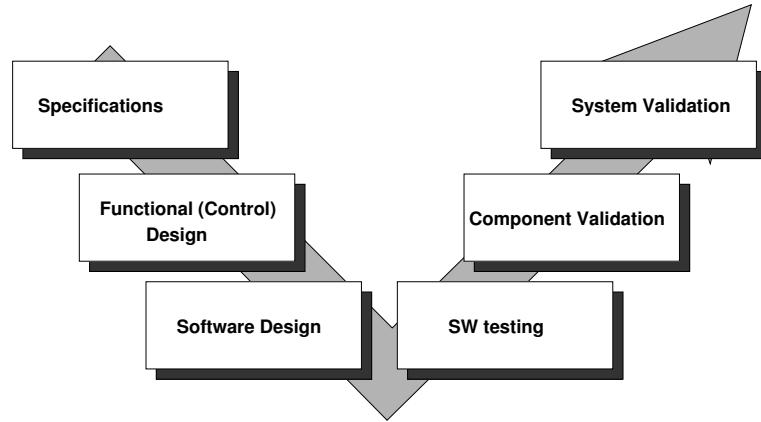


Figure 1: V-shaped design process.

A typical design flow of embedded software (a v-shape process is represented in Figure 1) includes (among others) two stages of fundamental importance. The first stage, at the logical level, is the domain of application developers such as control engineers or other domain experts and produces a model of the software application providing a solution to the functional requirements in terms of one or more block diagrams. Most commercial design flows, such as those based on Ascet-SD [5] or Simulink [9] provide extensive support for this stage. Compliance against functional requirements is typically validated by extensive simulation and the risk of functional errors due to incorrect implementation is reduced by tools for the automated translation of the model (Real-Time Workshop Embedded Coder [9] or TargetLink [4]).

The second stage of the design process is at the architecture level, where software engineers (real-time systems experts) map the functions/components developed in the previous stage to real-time threads, select the scheduler and the resource managers by exploiting the services of a Real-Time Operating System (RTOS) and ultimately check the correctness of the timing requirements upon the target (HW) architecture. This design stage is transparent to software functionality (only provides its implementation), but it is crucial to achieve performance/cost tradeoffs with the objective of providing the best possible quality and the best possible accuracy to system functions within the timing constraints and/or the performance target.

1.2 The problem

Unfortunately, even in state-of-the-art processes, the above steps are performed relying solely on the skills or the experience of project developers. As a consequence, projects often suffer from design errors that show up only after deployment with obvious economic losses. If the software manufacturer detects timing errors or unsatisfactory performance during the extensive testing period, a chain of engineering changes is initiated that implies many iterations between the logical and the architectural view, trading implementation complexity for schedulability (or time performance) and tuning many design parameters, including system resources and (when possible) activation rates. A typically occurring situation is that sophisticated algo-

rithms developed by control engineers simply cannot be programmed with acceptable performance on the chosen platform.

1.3 Innovating the architecture design

Schedulability analysis theory bears the promise to fill in this gap, since it enables formal validation of the timing behavior of architecture-level solutions at the highest possible level in the flow. To this purpose, timing properties and constraints should be explicitly stated (by appropriate design annotations) and the (hardware and software) resources available for the execution of the application tasks and the resource allocation and scheduling policies must be clearly specified.

We take inspiration from recent literature in the field of embedded systems design [10, 1], where a clear separation of concerns between functional design and implementation choices is advocated and the use of a shared set of abstractions allows for an efficient exchange of information between the two activities. In this way, the functional design remains independent from architectural choices and it does not require an advance commitment to any specific implementation.

2 Architecture-level design

2.1 Introduction

The distinctive aspect of schedulability analysis is a match between the timing (or, in general, QoS) requirements of logical entities in the logical layer and the corresponding QoS offers of an implementation or architecture layer (Figure 2) and consequently raises the fundamental problem of mapping logical-level entities into architecture-level entities [2].

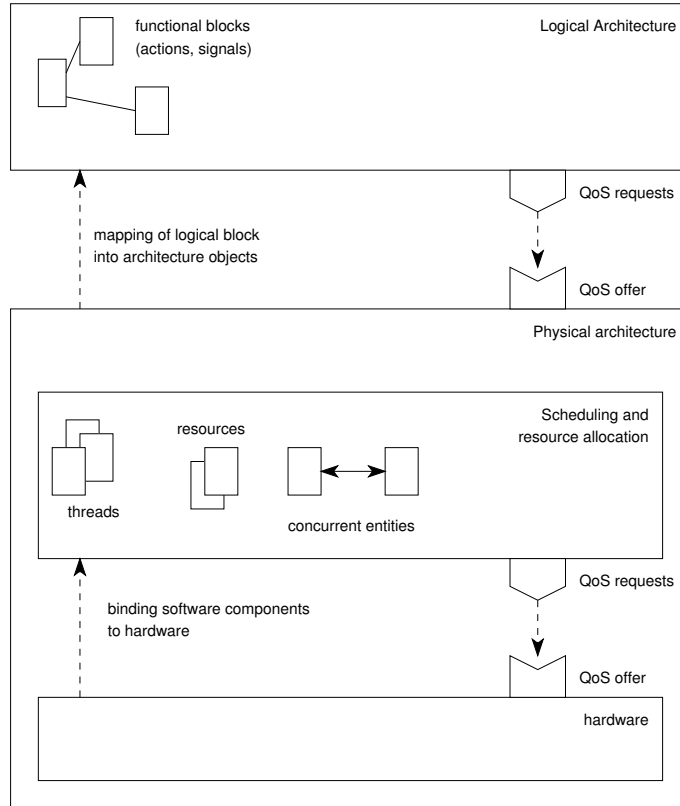


Figure 2: Schedulability analysis requires creating a correspondence between logical and physical architecture.

The logical architecture contains the design of the objects implementing the system functionality and their interactions. The functional model consists of a network of functional blocks or components connected by means of communication variables. This network is the outcome of the control design phase; its description is derived from the control schemes produced, for example, by tools such as Matlab/Simulink or ETAS ASCET-SD. The concurrency description layer contains an implementation description consisting of all the threads that are running in the system and of the policies for managing resources (usually implemented in the operating system). Finally, the hardware layer describes the hardware and its QoS features. The two bottom layers collectively define the so-called physical architecture layer, describing the mapping of the functional or logical design to a particular real-time execution environment.

The mapping stage consists of assigning each functional block to a software thread and each communication variable to a communication facility of the implementation (e.g. a task's private variable, a protected shared variable, etc.). It also includes an appropriate choice for high level execution parameters such as task activation rates. Designers of real-time embedded systems should be supported by adequate tools in exploiting the degrees of freedom that are available at the different layers of the architecture and in controlling the implications of

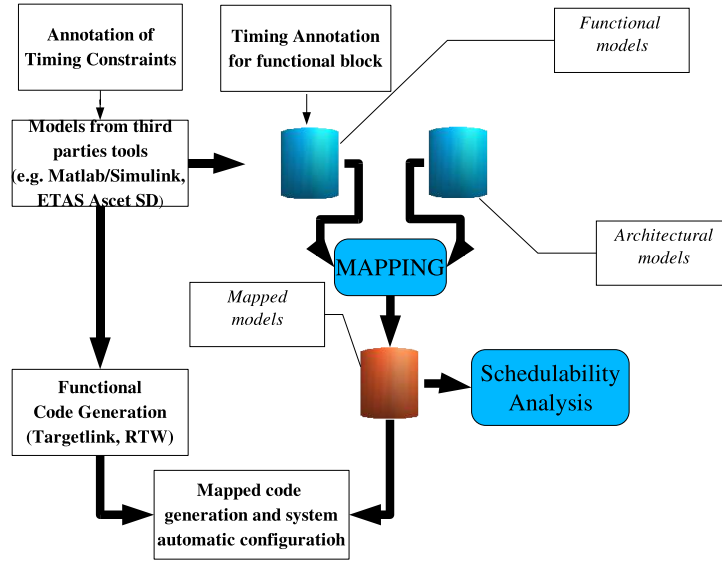


Figure 3: Schedulability analysis in the design flow.

mapping on timing and performance requirements. This capability requires support for the selection of the schedulability analysis techniques that are available at the concurrent entities level, and the corresponding restrictions that they impose onto the logical architecture design and the mapping itself.

In order to validate this stage the designer performs schedulability analysis (Figure 3) on his/her mapping hypothesis. Design parameters, such as ceilings for semaphores protecting shared resources, and task priorities, may be automatically synthesized (see the OSEK specification [8]).

Several iterations between mapping and schedulability analysis can be required before the final design decision and timing validation of the software architecture is achieved. An important feature of our envisioned methodology is that such iterations are not “blind-eyed” since schedulability analysis does not simply provide a boolean answer on the schedulability of the system, but it returns performance information (i.e. response times) and sensitivity analysis. The latter feature is particularly important because it provides insight as to the criticalities in the mapped system (e.g. execution times and/or rates jeopardizing the system schedulability or possible bottlenecks). This feedback drives mapping and/or architecture design changes but it can also provide a measure of the resource (time) budgets that should be accounted for in a possible functional redesign stage.

2.2 Logical level design

At the logical level, most real-time systems feature a control and a dataflow part. The dataflow part is often time-driven, since inputs are sampled at regular intervals and it is suitable for an implementation consisting mainly of periodic activities. In contrast, the control-dominated functionality is highly asynchronous and state-dependent. This software part typically executes in loops where it waits for an event, reacts to it by issuing an action and then blocks, waiting for the next event (event-driven).

The model of the system represents both parts with suitable abstractions. The syntax and the semantics of the modeling language used by control engineers may vary, but in general, all models provide a meta-representation of the design as a structure or network of (functional) components connected via nets to each other ports (mechanisms to communicate be-

tween blocks, such as shared variables) and communicating with each other using message (event) abstractions. Messages may be asynchronous (signals) or synchronous (call messages). Each component, and thus eventually the whole design, has a functional model, describing its behavior, that is how its output ports relate over time with its input ports. Both structure and functionality can be described using any appropriate modeling language such as Simulink, StateCharts, UML, but also C, Verilog, VHDL, and so on.

We provide a very simple and general metamodel (and a corresponding syntax based on XML) for representing an abstraction of the structure part of the functional model where each component and event is annotated with timing attributes and constraints, such as assumptions on the maximum rate of arrival of events and on the worst case computation time of components. This abstraction is suitable for specification of architecture-level mapping and verification of timing properties.

2.3 Software architecture design

If the logical architecture primarily focuses on functional requirements, the physical architecture level is where physical concurrency, resource requirements and schedulability constraints are expressed. At this level, the units of computation are the processes or threads (or in general, tasks), executing concurrently in response to environment stimuli or prompted by an internal clock. Threads cooperate by exchanging data and synchronization or activation signals and contend for use of the execution resource(s) (the processor) as well as for the other logical (e.g., shared buffers, network buffers, shared data, shared memory, semaphores) or physical (e.g., network element, backplane, I/O port, disk) shared resources.

The physical architecture level is also the place where the concurrent entities are mapped onto target hardware. This activity entails the selection of an appropriate scheduling policy (for example, offered by a real-time operating system), which must be clearly supported by techniques for analyzing schedulability. Some proposals exist for architecture level modeling, but commercial options are much less limited (if compared to the functional domain).

Our conceptual model uses the concepts of execution engine, scheduling policy, resources, and schedulable resources to model the elements of a physical architecture. An execution engine models the processing resource and has a scheduling policy that determines how the tasks will be scheduled. The mapping from a logical model to the elements in the physical architecture is done through relationships expressing the concept of realization or deployment. In practice, this may be accomplished by mapping or binding logical level entities to elements of the physical architecture (e.g., schedulable resources and shared resources).

Please note that the schedulability analysis model concepts of schedulable resources have straightforward mapping to most standard operating environments including OSEK, POSIX, Real-Time Specification for Java (RTSJ) and Ada.

2.4 Mapping

The mapping process consists of establishing an "implemented by" or realization relationship between functional components and architecture-level objects (such as threads) and in the definition of a binding or deployment relationship between threads and resources and hardware components. The mapping relationship is constrained by the specifications defined at both the logical and the architecture levels, such as precedence and exclusion constraints among functional blocks or timing constraints enforcing the activation rates of software actions.

Mapping and binding decisions have crucial impact on the result of timing analysis. Load balancing and local schedulability clearly depend on binding decisions, when multiprocessor architectures are considered, but the impact of the mapping stage can extend to many other (less intuitive) aspects. For example, communication messages may be handled transparently with respect to schedulability or they may result in shared variables with mutual exclusion at the local or global level with possible blocking implications on real-time software threads.

Tool support must be provided for exploring the space of all possible mappings by enforcing all constraints and providing insight on the consequences on schedulability of each mapping decision. Schedulability feedback must be expressed in a form, which allows the designer to pinpoint the exact cause that lead to schedulability or timing errors and give a range of actions as well as a measure of the entity of the corrections that would make the design feasible.

2.5 Schedulability Analysis

Once the application functional model has been mapped on to a set of real-time tasks, it is necessary to verify that all temporal constraints are respected before actually implementing the system. For example, a typical constraint for a periodic real-time task is that each instance (or "job") of the task must complete before the next period. In other cases, all jobs of a task must complete before a deadline.

A real-time task τ_i is characterized by a period T_i (or minimum interarrival time or maximum activation rate), by a relative deadline D_i (i.e. the time computed from the activation by which each job must complete) and by a worst-case execution time T_i . The goal of the schedulability analysis is to check if all jobs can execute C_i units of execution time between their activation time and their deadline.

In order to analyze a system, we must first specify the scheduling algorithm. Usually, real-time operating systems provide a scheduling algorithm based on fixed priorities. Each task is assigned a priority, and the highest priority task is executed at each instant. For example, all OSEK-compliant OS provide the fixed priority scheduling algorithm. For the fixed priority scheduling algorithm, three different schedulability analysis can be performed: the **utilization-based** test, the **response time** test and the **hyperplane** test.

The **utilization-based** test [7] assumes that deadlines are equal to periods ($D = T$) and that priorities are assigned according to the *Rate Monotonic Assignment (RMA)* order. According to this priority assignment, each task is assigned a priority directly proportional to its rate. Therefore, the task with the minimum period has the highest priority. It has been proved that for a set of independent task, the rate monotonic priority assignment is the best possible assignment, i.e. the assignment that permits to best utilize the processor.

The utilization test is based on the computation of the maximum processor utilization (or load) U of the application. The utilization demand of task τ_i is defined as

$$U = \frac{C_i}{T_i}$$

The processor utilization is the sum of the utilization demands of all tasks.

$$U = \sum_{i=1,n} \frac{C_i}{T_i} \quad (1)$$

The upper bound U_{lub} on the processor utilization is the maximum U that can be possibly achieved without violating the deadline constraints. In order for a task set to be schedulable under RMA, it suffices that the processor utilization is lower than the bound $U_{lub} = n \left(2^{\frac{1}{n}} - 1 \right)$ where n is the number of tasks in the system. U_{lub} decreases as n increases eventually approaching the limit value of $\ln 2 = 0.693$. This well-known limit value tells that, if the RMA policy is used and the processor utilization is below 69%, than any task set is schedulable.

The utilization-based schedulability test only gives a sufficient condition. If the test fails, results are not conclusive. However, the test is useful since it gives us an indication of the maximum possible load of the system.

The **response-time** test [6] can be applied to any priority assignment (not only to RMA) and to every value of the deadline. It is a necessary and sufficient test, and it is more complex than the utilization-based test. It consists in computing the worst-case response time W of every

task. The critical condition happens when all tasks are released at the same time instant. To compute the worst-case response time, an iterative formula is used:

$$W_i(k) = C_i + \sum_j \left\lceil W_i \frac{(k-1)}{T_j} \right\rceil$$

The iteration stops when $W_i(k) = W_i(k-1)$. If the obtained value of the worst case-response time W_i is less than the deadline D_i , the task τ_i is schedulable.

The **hyperplane test** [3] is the most complex of all tests. Like the response time test, it is necessary and sufficient and it does not require the deadline to be equal to the period. It is based on the concept of the "space of schedulability" of the system. The method consists in building a set of linear inequalities for each task τ_i . If any of the inequalities is verified, the task is schedulable. The methodology is repeated for all tasks and if all tasks are schedulable, the system is schedulable. We do not report the formula here for they are very complex. The advantage of this test over the previous ones is that it can be used for sensitivity analysis. In addition to the schedulability of the system, the test gives us the following information:

- if the system is schedulable, it is possible to know how much the computation time of a task can be varied maintaining the system schedulable. This is very useful if we want to replace one procedure with another one that is more complex but more accurate. Also, it is possible to know how much the processor clock can be slowed down without compromising the schedulability.
- if the system is not schedulable, it is possible to know how much we should reduce the computation time of a task in order for the system to become schedulable. Again, this is useful in the case we can substitute one functional procedure with another more simple one. Also, it is possible to know how much we can increase the processor clock speed to make the system schedulable.

This additional information is very useful during system design because it is possible to change and play with the system parameters until a satisfying configuration is found.

3 The RT-Druid Toolset

An integrated set of design tools (called RT-Druid) aids the designer in modeling, analyzing, and simulating the timing behavior of embedded real-time systems according to the methodology presented in this paper. An open and extensible environment, based on XML and open standards (Java), RT-Druid can be easily integrated with existing systems.

Furthermore, because of its generic framework, the tools give a flexible modeling and analysis platform for modeling any hardware and software, providing compatibility with most of the model-based design methodologies on the market. At the same time, our metamodel contains precise behavior and synchronization information to aid in assessing worst-case timing response and providing feedback to the design cycles.

The tool was designed aiming at the following general goals:

Modularity: once the kernel module is installed, each design activity in the development flow is in charge of a module that can be separately purchased and used as standalone.

Interoperability with third parties: the tool architecture and the generality of the internal models make it easy to interoperate with state-of-the-art tools for functional design (Matlab Simulink and Ascet/SD), code generation (Real-time workshop/Embedded coder, Targetlink) and architecture design (Cadence VCC). Instrumental to interoperability is the adoption of open standards for file formats and repositories (XML).

Portability across different execution environments: the tool is designed and implemented in Java for maximum portability to different environments and operating systems (MS Windows 2000/Xp, Linux, Macintosh).

Extensibility: future extensions include custom plug-ins and integration with third party production tools for code generation complying with industrial standards: such as the OSEK and its related standards (OIL, ORTI) in the automotive domain.

Moreover, the tool will provide seamless integration with our OSEK family of products, including the ERIKA real-time kernel.

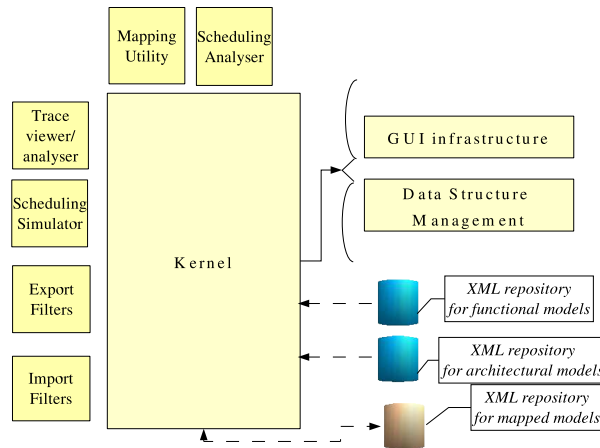


Figure 4: RT-Druid architecture

The RT-Druid architecture is shown in Figure 4. Model information (i.e. for both the functional and the architecture-level parts) is stored in an internal repository and it is made available by means of an open format based on XML. The toolset architecture is based on a kernel

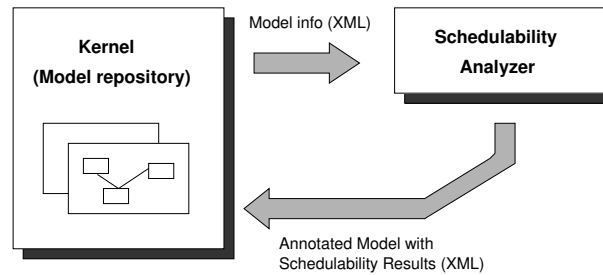


Figure 5: Interactions between the kernel module and the Schedulability analyzer.

module, providing management of internal data structure and basic services for GUI and additional plugin modules.

Plugins exploit kernel services in order to provide support to the design stages in a completely independent way. Here is a list of the plugins currently planned:

- The **modeling and mapping module** provides general Graphics and Text Diagrams to capture textual requirements, flowcharts, component and deployment notation and other general purpose information. Design teams can develop a representation of the embedded real-time system in graphical or textual form and capture the interaction between functional and architecture-level designs without disrupting the existing development process. This is done by using a neutral (XML-based) format and metamodel.

Both logical and architecture-level modeling entities can be interactively defined within the context of the tool or imported by translation from external models. XML-based import/export utilities enable the transfer of information between our tools and other tools supporting XML or text-based standards.

Modeling elements for logical and architecture-level abstractions mirror the definitions given in the previous sections. In general, they capture the structure of the functional part, including components/blocks with the corresponding actions, events, messages and the timing constraints acting upon them and the elements describing the software and hardware architecture of the design, including the specific multi-tasking and concurrency aspects of the system, task synchronization, communication, and mutual exclusion of resources.

A binding facility enables the designer to fasten logical blocks to software components and the latter to hardware resource components.

- The **scheduling analyzer module** provides support for worst case timing analysis. The module interacts with the other components by exchanging design information in the form of back annotations according to the schema of Figure 5. The forward path of Figure 5 represents the communication of the system model (as defined in the previous sections) to the schedulability analysis tool. The inverse model conversion provides an alternate option for using the results of the schedulability analysis since RT-Druid can produce its results in the same XML files that are used to encode the design information. In particular, it stores the values for the attributes (priority, priority ceiling) that are synthesized based on the selected scheduling policy and the schedulability results themselves. Back-annotating the results into the same XML file that represents the model of the system helps the designer in keeping all the information related to the current iteration in the design flow aligned and consistent.
- **Scheduling simulator**: for those systems where the evaluation of the worst case response time of the threads needs to be supplemented by simulations of average as well as critical runs highlighting the expected time behavior of the system.

- **Trace viewer/analyzer:** providing support for estimating the execution time attributes of software components. A graphical front-end is able to display the timing properties of the system, letting the user understand the run-time application behavior and enabling useful back-annotation that will help in a better system characterization. Input can be taken from a simulated trace and from real execution on the target microcontroller.
- **Export/import filters:** for automatically importing model diagrams from existing tools such as Simulink, ASCET or UML-based design frameworks.
- **Automatic code generation features:** for automating the production of functional as well as architecture-level code and debugging information in OSEK-compliant systems.

4 More information

For more information on the RT-Druid Toolset and other Evidence Products, please send an e-mail to the following address: info@evidence.eu.com.

Other informations about the Evidence product line can be found at the Evidence web site at: <http://www.evidence.eu.com>.

5 About Evidence S.r.l.

Evidence is a spin-off company of the ReTiS Lab of the Scuola Superiore S. Anna, Pisa, Italy. People at Evidence are experts in the domain of embedded and real-time systems with a deep knowledge of the design and specification flow of embedded SW, especially for the automotive market. We keep providing significant advances in the state of the art of real-time analysis and multiprocessor scheduling. Our solutions aims at bringing new approaches to conventional embedded systems as well to the frontier of embedded computing, such as multiprocessor-on-a-chip, reconfigurable hardware, dynamic scheduling and much more!

References

- [1] *Hardware-Software Co-Design of Embedded Systems: The POLIS Approach*. Kluwer Academic Publishers, 1987.
- [2] *UML Profile for Schedulability, Performance, and Time*. OMG document ptc/02-03-02, 2002.
- [3] Enrico Bini and Giorgio C. Buttazzo. The space of rate monotonic schedulability. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pages 169–178, Austin, Texas, December 2002.
- [4] dSPACE Inc. Targetlink. <http://www.dspace.de/>.
- [5] ETAS Gmbh. Ascet-SD. <http://www.etas.de>.
- [6] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1989.
- [7] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1), 1973.
- [8] OSEK/VDX. Osek/vdx standard. <http://www.osek-vdx.com>.
- [9] Inc. The Mathworks. The mathworks simulink and stateflow. <http://www.mathworks.com>.
- [10] Alberto Sangiovanni Vincentelli. Defining platform-based design. In *EE Design*, March 2002.