Model-Based Deployment in Automotive Embedded Software: From a High-Level View to Low-Level Implementations

Andreas Bauer, Jan Romberg
Institut für Informatik, Technische Universität München
Boltzmannstr. 3, D-85748 Garching b. München, Germany
{baueran|romberg}@in.tum.de

Abstract

The electronic components in present-day automobiles are based on networks of electronic control units (ECU) running embedded software. The development of distributed, concurrent software applications based on such ECU networks is becoming increasingly complex and error-prone. In current practice, system-level views of the network are merely used to characterise technical constraints such as performance requirements, and to choose the hardware and software components accordingly. In contrast, the semantic integration of the distributed functions is typically deferred to later points in the development process, yielding a high effort for integrating and validating such distributed functions. To address in particular this issue, our paper advocates a more stringent use of high-level models based on distinct abstractions and a well-defined behavioural semantics. We introduce the corresponding notations and tools, and the overall methodology developed to support a stepwise development of distributed automotive applications. The paper then details on the issues of using such high-level models to facilitate deployment, and to obtain low-level implementations from integrated system models.

1 Introduction

Until recently, the electronic control system in a vehicle was mostly concerned with light switches, windshield wipers, or starter motors all of which were, more or less, realised as isolated systems provided from independent suppliers. Traditionally, the software for such embedded systems was implemented in a relatively low-level fashion as C, or Ada programs, and often directly in terms of native machine code. The last decade, however, saw an increasing use of integrated development toolkits such as ASCET [1], or the Simulink Real-Time

Workshop [2] which facilitate reuse and provide retargetable generation of code based on dataflow models.

However, the nowadays increasing number of distributed ECUs in vehicles imposes fundamentally different problems for the automotive industry which is not tackled by the existing tool support alone. The sharing of data between ECUs that communicate via dedicated busses and bus protocols (e. g. CAN, MOST) allows the integration of additional functionality at lower costs. Consequently, this domain now requires different abstraction levels to be able to capture the actual dataflow between distributed ECUs inside vehicles as well.

Such higher-level models are also necessary to simulate and verify the behaviour and communication between ECUs to guarantee for safety and reliability of the deployed software. Ideally, the abstract models also facilitate reuse on various levels of abstraction. In a distributed system, isolated solutions at the level of programming languages are clearly not suitable for these requirements. However, high-level models raise a number of other issues though: for example, is it feasible to use them directly for code generation in a domain which, traditionally, confronts its users with limited computational resources?

Therefore in § 2, this paper first outlines such a typical target platform for (safety critical) embedded software as we encounter it prominently in the automotive domain. The actual partitioning and deployment issues are described in § 5 for which we first introduce abstractions and system descriptions that will also help classify the presented concepts in a realistic automotive industry context. Additionally, we briefly sketch the synchronously clocked computational model underlying our modelling formalism.

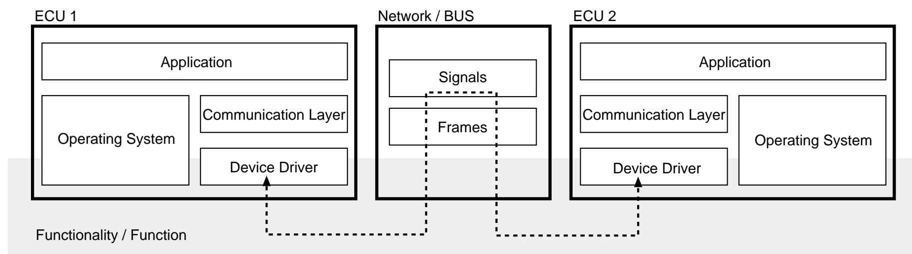


Figure 1: A distributed target architecture.

2 An Abstract Target Platform

Our abstract target consists of a network of ECUs connected via a bus. As can be seen in Fig. 1, each ECU is embedded into a *host node* which consists of the ECU itself, an operating system, a device driver module for interfacing the bus, one or many application tasks, and a dedicated communication layer.

The horizontal bar at the bottom of Fig. 1 indicates that the functionality contained within a high-level system model may be arbitrarily distributed among the nodes of the network, i. e. distribution of a functionality is transparent in a top-down systems view. (Note that the terms “function” and “functionality” are used as synonyms in this context to describe a certain ability, or property of the system.)

The dedicated communication layer is merely a wrapper around the inter-task communication between applications of spatially separated ECUs. Its main purpose is to manage resources needed to buffer signals whenever necessary (see also § 5.1). Communication itself, however, is handled by the device drivers which can be automatically generated for each ECU and protocol variant.

3 Abstraction & System Description

With the ongoing shift in the automotive industry towards distributed — and ideally reusable — software components, practitioners are not faced with a uniform system view anymore, e. g. source code. Components are now designed to be *automatically* deployed in a range of different vehicle types within a single class many of which offer, say, varying on-board electronic controllers as well as a different number of available ECUs for deployment. On a more abstract level, the behavioural view of the rather differently deployed components is expected to remain constant though. Fig. 2 illustrates how different abstract views on automotive software components can be assimilated to a common integrated system model.

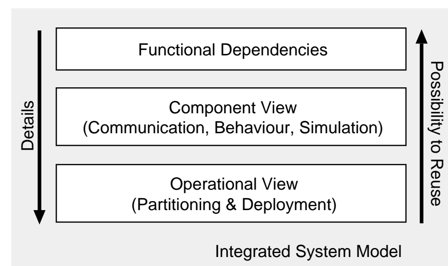


Figure 2: Abstract system views.

The view on functional dependencies is, typically, the most abstract model of an automotive software system. It captures the structure as well as the functional dependencies common to a class of vehicles by the same manufacturer. A component view, on the other hand, contains the internal interaction patterns of individual software components in terms of dataflow, communication and behaviour. This is already sufficiently expressive and detailed to allow for validation and simulation of designs, while an operational view, typically, contains

aspects which are unique to the actual target platform. Naturally, reuse of components gets increasingly difficult with a decreasing level of abstraction.

Each level needs to be associated with a number of custom description techniques, first to allow for independent top-down systems design, e. g. abstract definition of sensor and actuator components, and secondly for a subsequent refinement down to a mapping onto actual hardware.

Functional Dependencies: Common to this view are structure oriented views, i. e. system structure diagrams (SSD), to describe the overall structure of a system. Typically, SSDs are specified as hierarchical component networks where components communicate via *typed* and *directed channels* and typed *ports*, similar to the visual representation of UML-RT [3] and some Architecture Description Languages.

Component View: In this abstraction, we require a description of the individual software components to be complete with respect to behaviour. Therefore, the employed description techniques, typically, include state transition diagrams (STD), low-level dataflow diagrams (DFD), or more message-oriented diagrams (see also [4]). DFDs can be viewed upon as a refinement of SSDs and describe the algorithmic dataflow occurring during a computation. They consist of components performing the computation (i. e. blocks), interface elements of those components (i. e. ports), and connections between those interface elements (i. e. channels).

Operational Model: The operational model employs a similar visual representation as the component view — cluster communication diagrams (CCD) — but it is an implementation-driven refinement containing those details essential for deployment. CCDs then represent the main computational units (i. e. clusters, abstract tasks) that interact *directly* with the real-time operating system (scheduler) and the dedicated computational layer; that is, clusters are the least distributable units from the integrated system model: clusters are not split across two tasks and are always connected using explicit signal sampling operators (see § 4 and § 5.2). In this context, however, clusters must not be confused with TTP-clusters [5].

Note, at this point, we abstain from giving a more detailed description of the individual visual representations and their exact properties as the important graphical notations relevant for this paper are introduced in § 4 and § 5 by practical examples, respectively.

4 Computational Model & System Behaviour

The behavioural model of the systems described in this paper is that of current AutoFocus [6, 4] models. It is based on the *synchrony hypothesis* using a

discrete notion of time. The synchronous paradigm [7, 8] basically states that a system reacts to external stimuli within one instant, i. e. the delay between internal computations cannot be observed. This approach has enjoyed widespread acceptance in the control and hardware design domains, and is largely compatible with the commercially established tools such as ASCET, or Simulink. As opposed to several other approaches used for real-time specification and programming, the discrete-time semantics and deterministic concurrency keep behavioural evaluation of large designs manageable. The AUTOFOCUS framework is based on such a deterministic time-synchronous interpretation: *components* communicate through timed *streams*, where each stream uses the same global time base.

In order to support the multiform event patterns and frequencies observed in typical real-time systems, each stream of signals is associated with a *clock*. Similar to other synchronous dataflow languages [7], an AUTOFOCUS clock can be thought of as a boolean stream that merely indicates whether a value is currently present (*tt*), or not (*ff*). Clocks characterise streams both external, such as frequencies imposed by surrounding actors or real-time constraints, and internal to the system: by using clock inference properties the internal clocks can be inferred from the according inputs, respectively. (Think of the integration of black-box "legacy components", for example.)

Our current tool prototypes provide both automated inference of internal clocks and static checking of well-formedness of the model, i. e. detecting absence of causal cycles and a soundness verification of clocks. The implementation is very similar to that of a static type system in strongly typed programming languages.

In AUTOFOCUS, each clock is defined w. r. t. a *base clock*, k, which is the fastest clock in and underlying a system; that is, the most fine-grained time scale upon which a system reacts to external stimuli. The base clock itself is represented by the boolean expression tt, i. e. the expression that evaluates to *true* at any instance of k. A model's clock expressions are typically ordered using a $\leq$-relation.

Furthermore, in AUTOFOCUS it is not only possible to infer clocks, but also to make up new ones based on other clock expressions. The DFD given in Fig. 3 bears an explicit when operator which samples the input stream a to the rate of boolean stream b; that is, $a' = a$ whenever b evaluates to tt. The output and input ports are depicted by black and white rectangles, respectively.

In accordance with the notion of using clock expressions, all of a system model's entities can be represented using a dedicated language based on expressions. Consequently, *expressions* in AUTOFOCUS range over channels, ports, and combinations thereof using dedicated operators.

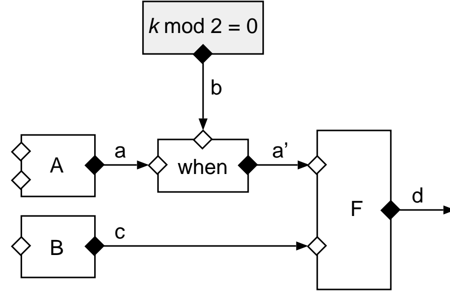


Figure 3: Explicit signal sampling in DFDs.

Let Exp be the set of all such expressions used in a system model and let $Exp_{\mathbb{B}}$ denote the set of all boolean expressions. We can now introduce a function ck which gives us the actual clock of any $e \in Exp$:

$$ck : Exp \rightarrow Exp_{\mathbb{B}}.$$

Example. To illustrate how clocks are put into practice, let's assume that the following virtual values are being transmitted in the model as it is given in Fig. 3 where τ denotes an absent signal/value:

a:	1	2	3	4	5	6	7	8	...
b:	*tt*	*ff*	*tt*	*ff*	*tt*	*ff*	*tt*	*ff*	...
$a' = a$ when b:	1	τ	3	τ	5	τ	7	τ	...
c:	0	τ	1	τ	2	τ	4	τ	...

Obviously, a never yields τ, indicating that its clock is exactly the system's base clock. Here, sampling is necessary since F expects its inputs both at the same pace as c: every second "tick" relative to the base clock. Hence, the when operator projects the stream a to the slower clock explicitly defined by boolean stream b. Note, however, up sampling works accordingly and is achieved using the same operator. □

In order to allow for well-defined feedback loops and to provide memory slots holding temporary values, explicit delay operators are necessary. Fig. 4 depicts a model which makes use of an explicit delay block (black and white diamond shape) that behaves as follows: a value is held for one clock period respectively; the period is determined by the clock speed of the stream setting that value.

Here, the delay is used to "feed back" a previous value of F's computation, b. Each delay block is associated with an initial value. Note that the clock of the delayed signal equals the clock of the original signal. That is, the clock of a' is that of b, and if $ck(a) = ck(b)$, then $ck(a) = ck(b) = ck(a')$.

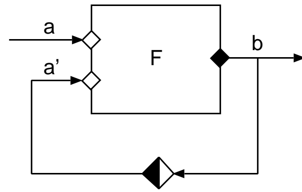


Figure 4: DFD with delayed signal.

5 Partitioning and Deployment

One inherent property of SSDs in AUTOFOCUS is the underlying assumption that communication between components is always delayed (i. e. each connecting channel contains exactly one implicit delay operator). This property of SSDs enforces the introduction of "predefined breaking points", which will be needed on the Operational View level to partition the design into individual tasks. From a methodological point of view, this definition facilitates the individual and also more independent development of each specified component. In the graphical notation, delayed communication is expressed with rounded ports (see Fig. 5).

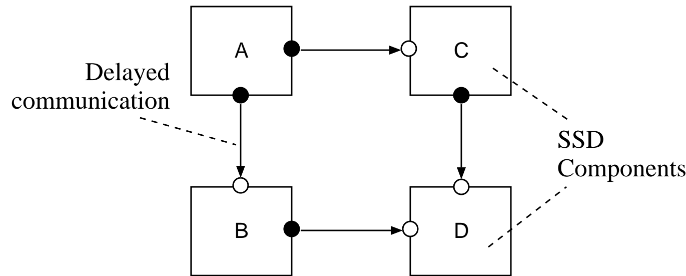


Figure 5: Example SSD.

We will show in the following that the introduction of explicit delays in early stages of development in our time-synchronous system model are prerequisites for deploying a distributed application across several tasks, or even across network of ECUs.

5.1 Communication Layer

Instead of dealing directly with inter-task communication, data-consistency, and I/O handling, we define an abstract communication layer that "wraps" all read and write accesses, respectively (see § 2). This layer acts as a kind of middleware providing basic communication services and data integrity to the application's tasks running on it. In particular, the middleware provides a `write` handler (similar to `SendMessage()` service in OSEK COM [9]), and a `read`

handler for messages (similar to `ReceiveMessage()` in OSEK COM).

Effectively, the layer constitutes a transparent communication model for each node and the tasks running on it, i. e. its technical realisation ensures that sufficient heap (register, or buffer) space is allocated when messages need buffering as is the case, for instance, when tasks with different clock speeds exchange signal frames. The following prerequisites are essential for the communication layer to yield the desired behaviour in practice:

- Execution of an accurate static analysis for minimal message allocation,
- and predetermination of an appropriate task scheduling algorithm.

Our AUTOFOCUS-based prototypes already provide for the former by allowing the static analysis and by associating appropriate memory with each delay operator in the model. The *exact* required amount of temporary space in total is then determined by a subsequent "clock comparison" of the communicating tasks (see § 5.2).

For the remainder of the paper, we assume a rate-monotonic scheduling policy, based on an operating system with a fixed-priority preemptive scheduler, where task priorities may be statically assigned; the latter limitation is, for instance, imposed by the OSEK standard [10] for automotive operating systems. Rate monotonicity simply asserts that tasks with smaller periods are assigned higher priorities than tasks with greater periods [11].

5.2 Variables and Message Slots

The operational system abstraction/view, as sketched in § 3, contains the transition from the hierarchic and connected SSD components to a *clustered* system view yielding all delay and sampling operators; that is, relevant implementation details.

The CCDs then present a *flat* description of the time-synchronous system model which allows for the static analysis of the heap (register, or buffer) consumption in terms of message buffers as well as for (almost) arbitrary partitioning variants: unlike SSDs which are grouped according to conceptual coherency and as reusable units, CCDs are typically partitioned to either

- yield a maximum of technical efficiency in the implementation,
- to account for physical proximity of an application part to sensors and actuators, or
- to adapt the software structure to other non-functional requirements, such as fault tolerance requirements.

There are also cases where it is required to partition CCD clusters along the same boundaries as SSD components. For instance, if components A and B are

known to be always mapped to different processors, then the clusters A and B should be fully disjoint, i. e. there exists no cluster containing parts of both A's and B's functionalities.

Note that the semantics of SSD composition, i. e. every channel incorporates a delay, ensures that the following delay constraints for CCDs are met given rate-monotonic scheduling and when communication follows the boundaries of the SSD components. Let A and B be a sending and a receiving cluster, respectively:

$ck(A) = ck(B)$**,** i. e. equally fast clocks. In this case, the priority of A's and B's tasks are the same, so communication occurs delayed; in effect, the communication layer needs to provide two message copies to avoid data inconsistencies.

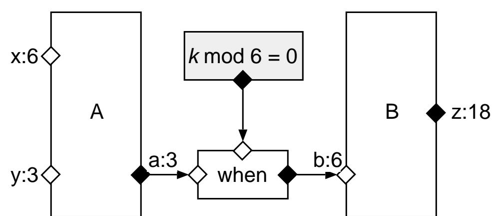
$ck(A) < ck(B)$**.** When the clock of A is faster than the clock of B, i. e. the period is smaller, we may use undelayed communication; only one message copy is needed.

$ck(A) > ck(B)$**.** Communication is delayed, when the clock of A is slower than the clock of B; in this case, two message copies are needed.

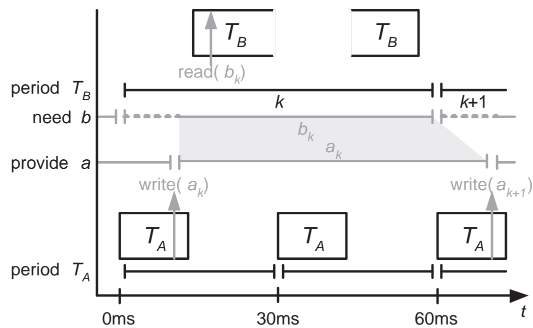
Obviously, the above comparison of task periods and static memory analysis is only possible by extending the clock associations from individual ports, or channels to the entire clusters themselves. Therefore, for periodic designs, a *cluster clock* is inferred as the "greatest common divisor" (gcd) of its individual clock periods. Note that internal clocks cannot — by definition of blocks and DFDs — be faster than the fastest external clock, so considering the clocks of incoming channels/ports in order to determine a cluster clock is fully sufficient. The following examples elaborate on that.

Example (fast cluster $\rightarrow$ slow cluster). In Fig. 6(a) an example CCD consisting of two clusters A and B is depicted. For the sake of simplicity, we only consider periodic clocks, and write the clock periods next to the corresponding ports. By attaching a label $x : 6$, we indicate that a port x holds a value every 6th tick relative to the base clock; that is, in our case we obtain $ck(A) = gcd(6, 3, 3) = 3$. A writes signal a, which is sampled by a when-operator and read as signal b by cluster B. Communication between A and B is not delayed.

Furthermore, let's assume that cluster A corresponds to a task T_A with a period and deadline of 30ms, and that cluster B corresponds to a task T_B with a period and deadline of 60ms. In other words, T_A and T_B are each released periodically at the beginning of their respective cycles which are indicated in Fig. 6(b) by black horizontal lines.



(a) Undelayed CCD.



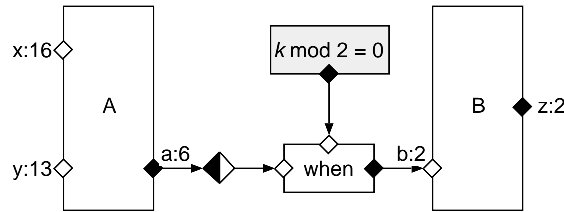
(b) Copped need-provide interference-polygon (grey).

Figure 6: Fast cluster writes to slower cluster.

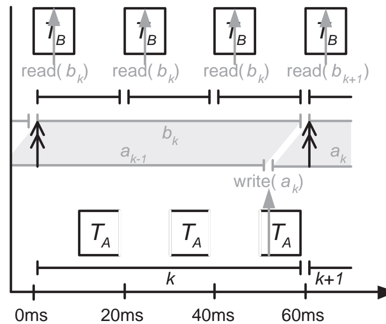
Both tasks are executed on the same ECU, and are scheduled according to the rate monotonic policy, i. e. T_A has a higher priority than T_B. In order to avoid data inconsistencies, for any step k, T_B needs a stable value b_k during the whole duration of its period. In the time scale, this is indicated by the grey "need b" bar.

A new value for port a is provided periodically by T_A, indicated by the "provide a" bar. Note, for any step k of B, the "provide a_k" bar starts chronologically after the "need b_k" bar.

Because of T_A's higher priority, b_k will never actually be read before T_A has finished its computation, and a_k has been written. We indicate this by a dashed bar for "need b" during T_A's activation. Therefore, we can safely associate a_k with b_k, which corresponds to immediate communication in the model. Since the written variable and the read variable correspond to the same memory location, the communication layer does not have to perform an explicit message copy operation. This example has shown that communication from fast to slow clusters does not require the introduction of additional delays in the model. □



(a) CCDs connected using a delay.



(b) Bent need-provide interference-polygon (grey).

Figure 7: Slow cluster writes to faster cluster.

Example (slow cluster → fast cluster). Fig. 7(a) depicts a CCD with two clusters A and B. The overall cluster clock of A is 6, B's is 2. Now the slower cluster A writes to the faster cluster, and the clocks are in a relationship $ck(A) > ck(B)$. According to the rules on page 8, an explicit delay is imposed in such cases, indicated by the diamond-shaped operator between a and b.

Fig. 7(b) shows how the delay relates to the time scale of two associated tasks T_A (period/deadline 60ms) and T_B (period/deadline 20ms). Since $ck(A) > ck(B)$, the "need b" period can be safely extended to T_A's period of 60ms.

This illustrates that if all tasks meet their respective deadlines, for any step k, T_B will never read b_k before a_{k-1} has been written. We can, therefore, safely associate a_{k-1} with b_k for any k, corresponding to a delay in the model. The black double-headed arrows indicate explicit message copy operations performed by the communication layer. □

6 Conclusions & Summary

In this paper we have shown that deployment related issues in the development of distributed automotive controlling software, like insertion of explicit delays in a time-synchronous system model, must not necessarily be driven in a bottom-up manner, but can also be asserted high-level and from a top-down perspective. Given the underlying assumptions regarding schedulability and the various static analyses, the introduction of delay operators in early development stages through the use of SSDs yields several advantages: firstly, the delays constitute predetermined breaking points in subsequent refinement and implementation processes, and secondly upon partitioning and clustering of the components, delays must not be added manually, i. e. the original communication structure remains mostly unchanged. The latter is particularly important, because essentially it means that a formerly verified behavioural model of the system, remains stable in the final implementation; all the implementation's delays have been present in the structural view as well. This lowers the validation and verification efforts drastically and increases the reusability of components.

Although, as we have sketched in § 5.2, delays are not always essential to support, say, the writing of a fast cluster to a slower cluster. However, early assertion of a delay does not alter the communication's behaviour if inserted *after* the down sampling operator that lies in between the CCD clusters. What is more, in that case it is theoretically possible to assert an arbitrary amount of delay operators after the down sampling occurs; the result being a higher memory consumption due to excessive message buffering.

On the other hand, this example illustrates that top-down asserted delay operators do not necessarily guarantee for the most efficient implementation of a distributed application. In fact, this paper comprises a trade-off between these very aspects of optimisation and the advantages of having separate, reusable

and verifiable system components. In other words, using the presented methodology results in lower verification efforts on the one hand, and in a less efficient implementation on the other.

Furthermore, in § 3 we have introduced and sketched several graphical notations to support the presented development process of distributed embedded systems: a hierarchical SSD description to capture a system's overall structure, DFDs to express a component's computation and dataflow, and CCDs to explicitly visualise deployment details and to facilitate partitioning according to, say, "clock boundaries", or SSD component boundaries. (Compared to SSD-driven partitioning, a clock-driven strategy groups clusters according to common clock speeds which often results in faster implementations.)

Editors for the discussed notations, the key algorithms underlying the analysis (e. g. clock inference and well-formedness checks) and the various abstract system views are already supported by a tool prototype based on the existing AUTOFOCUS framework.

Acknowledgements

We would like to thank Timothy Bourke, National ICT Australia, Sydney, who read and gave valueable feedback on early drafts of this paper.

References

[1] ETAS Engineering Tools GmbH. *ASCET-SD Benutzerhandbuch*, 2001.

[2] The MathWorks Inc. *Using Simulink*, 2000.

[3] B. Selic and J. Rumbaugh. Using UML for Modelling Complex Real-Time Systems. *ObjecTime Limited/Rational Software White Paper*, 1998.

[4] Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. AutoFOCUS: A Tool for Distributed Systems Specification. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 467–470, 1996.

[5] H. Kopetz and G. Grünsteidl. TTP — A Protocol for Fault-Tolerant Real-Time Systems. *IEEE Computer*, 27(1), 1994.

[6] The AUTOFOCUS Homepage. http://autofocus.informatik.tu-muenchen.de/.

[7] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. LeGuernic, and R. De Simone. The Synchronous Languages Twelve Years Later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[8] M. Broy and K. Stølen. *Specification and Development of Interactive Systems:* FOCUS *on Streams, Interfaces, and Refinement.* Springer, 2001.

[9] OSEK VDX consortium. *OSEK/VDX Communication Version 3.01*, 2003.

[10] OSEK VDX consortium. *OSEK/VDX Operating System Version 2.2*, 2001.

[11] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1), 1973.