Automatic Generation of Schedules for Time-Triggered Embedded Transducer Networks

Wilfried Elmenreich, Christian Paukovits, Stefan Pitzek
Vienna University of Technology
Institute of Computer Engineering
Treitlstraße 1-3/182-1, Vienna, Austria
{wil,pauko,pitzek}@vmars.tuwien.ac.at

Research Report 8/2004

Abstract – *Time-triggered systems are advantageous for embedded applications, when determinism, hard real-time behavior, and a straightforward way for certification are required. However, when it comes to flexibility, time-triggered systems often require that possible extensions have been planned in advance, which makes it difficult to apply unforeseen changes to such a system. This paper presents an approach that overcomes this problem by an automatic configuration of time-triggered distributed applications. Each application is decomposed into jobs and mapped on a set of distributed nodes, whereas each node hosts one or more jobs. Each job is defined by the following interfaces: a real-time input/output, a configuration and planning, and a diagnostic and management interface. Jobs, nodes, the application, and the global system configuration are represented with XML-based description formats, that provide a language-neutral semantic specification of the respective properties. A scheduling tool uses these descriptions to generate a time-triggered application that complies to the application specification.*

1 Introduction

The time-triggered approach is a well-suited approach for building distributed hard real-time systems. Time-triggered systems have the great advantage over traditional event-triggered systems that they are easier to understand and to analyze [1]. Furthermore, the time-triggered paradigm supports composability with respect to the temporal behavior [10] of a system. Since many applications of transducer networks have real-time requirements, using a time-triggered communication architecture becomes an interesting option for interconnecting smart transducers.

However, due to the static scheduling approach of time-triggered systems, event-triggered systems are more flexible than time-triggered ones [4]. One reason for this difference in flexibility is the higher off-line configuration effort required for time-triggered systems, since communication usually depends on schedules that must be calculated a priori. Accordingly, manual configuration becomes difficult or even infeasible very fast, so adequate tool support is especially important. Since time plays such a central role, the automatic creation of communication schedules is essential for automating configuration tasks for time-triggered systems.

In the literature, there exist the following approaches to increase flexibility in time-triggered systems: Almeida et al. present a flexible time-triggered communication approach that operates on a CAN network [2]. In this approach, messages are scheduled by a central authority, which allows for a compromise between flexibility and run-time overhead of the planning scheduler. Other approaches [16, 3, 17] follow an integration between a safety-critical static time-triggered communication channel with a flexible event-triggered channel. Lisner proposes a flexible TDMA slotting scheme, where communication partners have a fixed schedule in a regular part and may request extra slots in an extension part [15]. On the other hand, model-based approaches like Giotto [8] target the problem from a different side by separating the temporal properties of an application from its implementation. This facilitates the problem of modifying the static schedule, since the schedule is automatically created from the application specification by a re-run of the compiler tool. However, the Giotto approach poses considerable resource requirements on the particular nodes, which makes this approach less feasible to be used for low-cost smart transducer systems.

In this paper we propose a model-based approach to support flexibility and composability for a time-triggered system by presenting a scheduling tool that generates time-triggered message schedules based on an application description. Since a scheduler cannot perform actual configuration tasks by itself, we also outline the overall system model and how the scheduling tool interacts with other parts of the system. The distributed target application is decomposed into jobs, where each job is described by its interfaces, i. e., a real-time input/output, a configuration and planning (CP), and a diagnostic and management (DM) interface. The overall application is defined by the annotated communications among jobs, specified with XML descriptions. A scheduling tool uses this formal description to generate the static schedule for the time-triggered application and verifies whether the schedule fulfills the temporal requirements or not. The unused bandwidth is used for diagnostic and management purposes. When a change or extension of the application is necessary, the XML descriptions are adapted and the tool recreates the time-triggered schedule and performs the verification again. If verification fails, the tool reports that the given resources (e. g., bandwidth) are not sufficient for creating a valid schedule.

The approach has been implemented on embedded hardware using TTP/A [13] as time-triggered communication protocol.

The remainder of the paper is organized as follows: Section 2 deals with the underlying conceptual model of this approach. Section 3 presents the scheduling tool and algorithm and Section 4 applies the tool in a case study. Section 5 concludes the paper and gives an outlook.

2 Conceptual Model

A distributed embedded application is described by a set of interconnected *jobs*, implemented as single tasks which receive their input prior their execution and provide their output after execution.

The target system is a set of distributed embedded *nodes*. A node can host one ore more jobs, as long as it is guaranteed that the job can fulfill its service in terms of required resources. For example, a job that performs I/O operations on a sensor can only be hosted on a node that is connected to this particular sensor. Some jobs might also be required to be hosted on a particular node due to legacy issues (job and node come as a packet) or fault-tolerance requirements (jobs are required to be run on independent hardware).

The described architecture follows the conceptual model of the integrated architecture presented by Kopetz in [11]. In the scope of this paper, we focus on a single distributed embedded application and assume that each job is implemented by a single task as described above.

A job is described by its operational interface. In embedded real-time systems we can distinguish the following interface classes:

Service Providing Linking Interface (SPLIF): This interface provides the real-time services to other jobs (cf. [9]).

Service Requesting Linking Interface (SRLIF): A job that requires real-time input requests these data via the SRLIF (cf. [9]).

Diagnostic and Management (DM): This interface is used to set parameters and to retrieve information about intermediate and debugging data, e. g., for the purpose of fault diagnosis. Access of the DM interface does not change the (a priori specified) timing behavior of the service.

Configuration and Planning (CP): This interface is used during the integration phase to generate the "glue" between the nearly autonomous services (e. g., communication schedules). The CP interface is not time critical.

The task implementing a job may also use local interfaces to physical sensors, actuators, displays, and input devices. When another job needs to access this information, there must be a consistent access to this data via the SPLIF or SRLIF interfaces.

An distributed application consists of one or more jobs that interact with each other via the real-time interfaces SPLIF and SRLIF. One or more jobs can be hosted on a *node*, whereas nodes communicate via a real-time communication system. Jobs that require local data exchange communicate via a shared memory interface. Jobs that are required to exchange data between different nodes communicate via the node's communication interface. Since at component level it is not yet defined, if a data exchange will we local or remote, the real-time interfaces of each job are uniformly implemented as ports whereas each port can be connected locally or via network to a matching port of another service.

The representation of an application by jobs and their interfaces only deals with the functional and data flow parts of the application. Additionally, we also require a temporal specification of a job that deals with the following properties:

- Properties that must be configured depending on the requirements of the application (e. g., the sample rate of a sensor).
- End-to-end requirements of the application that can only be specified and verified at the level of the distributed application (e. g., end-to-end signal delay in control loops).

We use the so-called Interface File System (IFS) [12] as central interfacing mechanisms to the smart transducer nodes. The IFS is an "extended" shared memory that acts as a temporal firewall between smart transducer nodes and the network. The memory is organized as a simple flat file systems, where files store data and can be executed. All application and configuration related information is mapped to the IFS and thus available at the network level over the respective interfaces. Thus, it also hides internals of the node for complexity reduction.

The specific properties are modeled in the semantic description of the job. The end-to-end requirements of the application are expressed by so-called *dependencies*. We have identified the following kinds of dependencies:

Connection: This dependency represents the data flow between ports of jobs. A connection is directed by having a source and a target part. An input port may have only one connection to an output port, while an output may feed several input ports.

Causal: By defining a causal dependency between execution of two jobs, all in-between jobs (on the directed application graph) must comply to this dependency, i. e., execute subsequently. In our context causal dependencies always incur timing requirements. An *instant* identifies the timing requirements of participating jobs. We distinguish the instant before and after execution. The *before* instant of a job is before its executed, i. e., the moment when input data must have arrived. The *after* instant happens after the job execution, thus a duration of WCET_{TASK} after the before instant, where WCET_{TASK} is the worst case execution time of the task implementing the job.

Phase: The phase dependency specifies non-causal time-related dependencies of instants among jobs.

All the presented properties are used to model the application requirements. All requirements are expressed in XML descriptions, which support interaction among tools (e. g., for modeling, code generation, and verification).

3 Scheduler for a Time-Triggered Real-Time System

This section presents a scheduling and verification tool that uses the XML application descriptions that have been defined in [7].

3.1 Scheduling Algorithm

The operation of the TTP/A Scheduling Algorithm produces a Round Definition List (RODL) for each application in the cluster configuration description written in XML. This algorithm had been implemented in a tool called the TTP/A Scheduler.

The main idea of the TTP/A Scheduling Algorithm is to represent the set of jobs and their dependencies as a graph, the so-called Precedence Graph. Each *vertex* of the graph embodies an associated task of a job, whereas an *edge* represents a *causal dependency* between the bordering vertices, i. e., two jobs. Additionally, (causal) edges are directed to represent the direction of the data flow and precedence of that dependency. An undirected edge in the Precedence Graph corresponds to a phase dependency. Figure 1 shows the Precedence Graph of a simple control application.

We define three conditions that a valid Precedence Graph must satisfy:

- A job must not depend on results from itself, neither directly nor indirectly. In terms of graph theory we can say, that the Precedence Graph must not include *loops* among any vertices. Note that closed control loops do not fall into this category, since the feedback signal is provided by the control environment in that case, thus using uor approach we can model all kind of control loops.
- Only one directed causal edge, in each direction, is allowed between a pair of jobs.

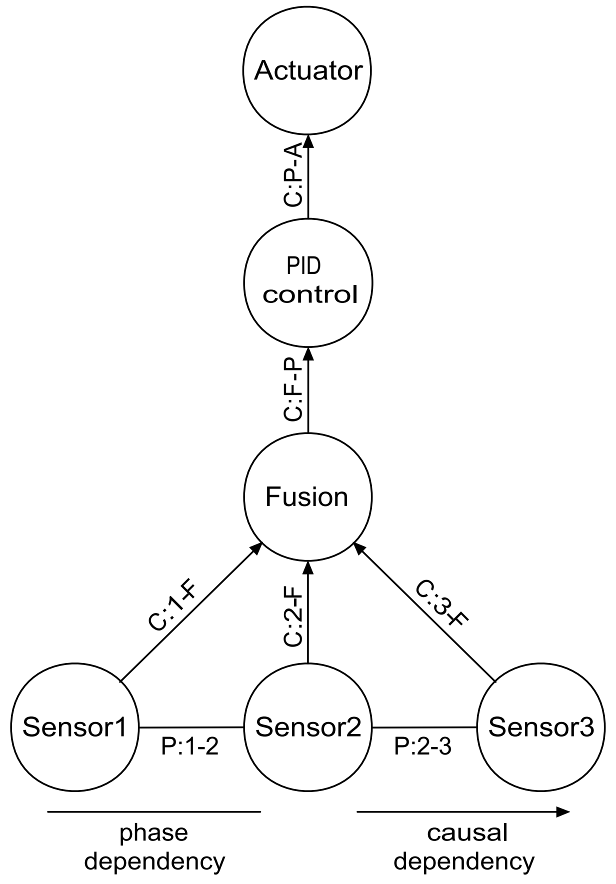

Figure 1: the Precedence Graph of the smart fusion application

- Jobs mutually depending on each other must not be "in phase". Hence there may not be a phase dependency between a pair of jobs, if we already have a *path* of causal edges between the associated vertices.

Phase dependencies are *transitive*. For instance in Figure 1 we see that jobs "Sensor1" and "Sensor2" are in phase, as well as "Sensor2" and "Sensor3". Due to the transitivity we conclude that "Sensor1" and "Sensor3" are also in phase. Therefore it is not necessary to draw a phase edge between "Sensor1" and "Sensor3".

Figure 2 outlines in pseudocode how the schedule works. The TTP/A Scheduler runs the TTP/A Scheduling Algorithm on the Precedence Graph for several iterations. In each iteration the algorithm searches for all those jobs that are not dependent on any other service task. We call this subset the *Candidates*. A Candidate can be recognized by the fact, that its vertex does not possess *incoming* causal edges

$\mathcal{S}$	...	set of jobs
$\mathcal{K}$	...	set of causual dependencies
$\mathcal{C}$	...	set of connection dependencies
$\mathcal{P}$	...	set of phase dependencies
V	...	set of vertices in the Precedence Graph (PG)

```
while(there exist unmarked vertices in PG)

$Candidates = \{x \in (V \models \mathcal{S}) \mid N^{+}(x) = \{\emptyset\}\}$
  $\forall a, b \in Candidates,\ a \neq b,\ \varphi = (a,b) \in \mathcal{P}$
    $b.offset = b.offset + |\varphi|$

  $\forall c \in Candidates$
    $TargetNode(c).RODL[RODLindex + c.offset] =$
      $OP\_EXEC$
    $RODLindex = c.offset + ExecutionTime(c)$

    $TargetNode(c).RODL[RODLindex] = OP\_SEND$

    $Receivers = \{y \in (V \models \mathcal{S}) \mid (c,y) \in \mathcal{C}\}$
    $\forall r \in Receivers$
      $TargetNode(r).RODL[RODLindex] =$
        $OP\_RECEIVE$

      $RODLindex = RODLindex + \#SendBytes$

    $mark\_in\_PG(c)$
```

Figure 2: Pseudocode representation of the scheduling algorithm

in the Precedence Graph. Candidates are sorted using the Earliest Deadline First algorithm.

Phase dependencies may cause a drift for the starting point of a pair of Candidates. Thus, for each pair of Candidates in phase, the one with the higher deadline is assigned the value of the *phase offset*.

Next, the Candidates are marked as done and the appropriate number of TTP/A slots for execution of each job is reserved in the RODL of the assigned target node. In case of an existing connection of any type (phase or causal) between a Candidate and some other service the scheduler inserts some TTP/A slots for the sending operation in the RODL of the Candidate's target node. In addition, the dependent jobs, i. e., the receivers, will have to perform a receive operation in the same TTP/A slot (at the same time). Consequently, the scheduler reserves receive slots in the RODLs of the target nodes involved with the jobs' tasks.

The design of the TTP/A Scheduling Algorithm guarantees an implicit *collision avoidance* at the real-time communication system level. For that purpose each target node possesses a local "RODL index", which points at the last occupied TTP/A slot in the

RODL.

In addition, there exists one “global index”. It points at the last TTP/A Slot, which has been occupied by communication from some job at the real-time communication system. When an execution of a task is to be entered at the target node’s RODL, the local index determines the starting slot. Then the phase offset of that job is added. The local index will be increased by the phase offset plus the number of occupied execution slots.

Moreover, the RODL index serves to determine, whether the communication system is occupied, when a target node intends to send the output of a job. If the local index is greater than the global one then no other node uses the bus. The node may send immediately after the completion of its execution, the local index is increased and the global index is adjusted accordingly. Conversely – if the global index is greater – the bus is occupied by somebody else and the time when the node has to trigger its sending operation is after the globally indexed TTP/A slot has passed by. Then the global index is increased by the number of used TTP/A slots and the local index is adjusted.

The execution of the Candidates leads to the release of causal dependencies. Hence in the next iteration other jobs are defined as Candidates, which had been dependent on other jobs in the prior iteration. The algorithm will terminate when all jobs are marked as done.

Finally, the tool performs a *deadline check* to verify whether the deadline specified by the application description hold for the generated RODLs. Therefore, the points of start and completion, as well as the arrival times of sent data for each job’s LIF and dependencies are compared to the deadline specifications in the cluster configuration description. The overall points in time must be lower than the deadlines, otherwise a deadline miss occurs. This examination will be conducted by the TTP/A Scheduler *after* the termination of the scheduling algorithm. In case of at least one deadline miss the result has to be discarded, because the algorithm could not determine a valid schedule. This indicates that the specified available resources of the environment are not sufficient for the specified application.

In short the TTP/A Scheduling Algorithm is a *greedy algorithm* with a simple, but efficient operation and data structure. It might not, however, produce the optimal result for a schedule of the RODL, but it guarantees that all specified deadlines will hold, when a valid schedule has been found.

4 Case Study

In order to show the capabilities of our architecture and the scheduling and verification tool, we have devised a distributed embedded application as case study.

We use the time-triggered protocol for SAE Class A applications TTP/A [14] as communication system. TTP/A is a low-cost fieldbus for smart transducer applications that implements the interfaces using the IFS concept and provides interfaces for real-time service, configuration/planning, and diagnostics/maintenance.

4.1 Initial Set-Up

The case study comprises two jobs using infrared sensors to measure a distance, a fusion job, a PID controller job and a job instrumenting an actuator. The application shall synchronously perform a distance measurement from the distance sensors, fuse the measurements, calculate a set value and instrument the actuator accordingly. We use the confidence-weighted average [5] method as fusion algorithm.

The temporal requirements of the application are as follows:

1. The set value of the actuator must be updated at least every 100 ms ($d_{update} = 0.1s$).

2. Sensor measurements must be synchronized with a precision of $\pm 1ms$.

3. The temporal accuracy d_{acc} of the value received by the actuator job must be 80 ms.

Figure 3 depicts the jobs, connections, and the mapping of jobs to physical nodes.

The initial set-up of the application requires four steps.

First, the firmware containing generic TTP/A communication and specific I/O jobs has to be programmed into the nodes. In industrial applications, this task belongs to the smart transducer vendor.

Second, all transducer nodes have to be connected to the TTP/A bus and their respective external hardware.

Third, the scheduling and verification tool is applied to the application description. In addition to the application descriptions, the tool also requires information on auxiliary constraints, e. g., the intended communication bandwidth, for the verification process.

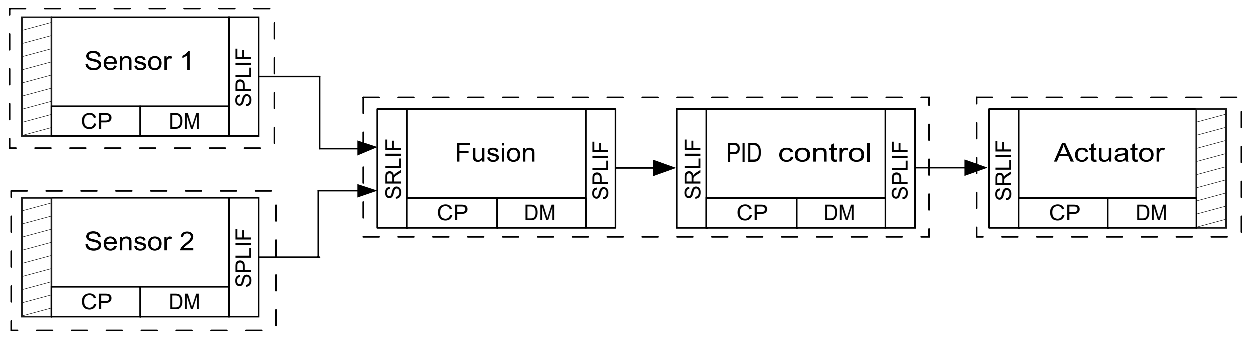

Figure 3: Interaction of jobs in case study. The dashed lines mark the assignment to physical nodes.

	cycle length	temporal accuracy	sensing synchrony	
1200 Bit/s	97.47 ms	75.84 ms	0.83 ms	$\surd$
4800 Bit/s	24.37 ms	18.96 ms	0.21 ms	$\surd$
9600 Bit/s	12.19 ms	9.48 ms	0.104 ms	$\surd$

Table 1: Timing properties for created schedule with 2 sensor jobs at varying communication speeds

	cycle length	temporal accuracy	sensing synchrony	
1200 Bit/s	*108.3* ms	*86.64* ms	0.83 ms	$\times$
4800 Bit/s	27.08 ms	21.66 ms	0.21 ms	$\surd$
9600 Bit/s	13.54 ms	10.83 ms	0.104 ms	$\surd$

Table 2: Timing properties for created schedule with 3 sensor jobs at varying communication speeds

Finally, if the verification holds, the schedule is uploaded to the network nodes using the configuration and planning interface of each transducer node.

Table 1 depicts the results from the Scheduling and Verification tool for the described application. As we can see, the deadlines of the temporal accuracy d_{acc} and the cycle time d_{update} hold for communication speeds of 4800 and 9600 bit/s. However, the temporal accuracy at 1200 bit/s fails with 75, 84 ms, whereas the cycle time is on the brinck of missing its limit of 100 ms with 97, 47 ms.

4.2 Configuration with Three Sensors

As a case study for extending a configuration, we assume that the application should be extended to three sensors instead of two. Since the fusion algorithm is scalable, the application description can be easily extended by adding another sensor.

After running the Scheduling and Verification tool with the new application description we received a positive verification for communication speeds of 4800 and 9600 Bit/s, but not for 1200 Bit/s[1] (see Table 2). The temporal accuracy d_{acc} exceeded its deadline of 80 ms with 86, 64 ms as well as the maximum allowed cycle time $d_{update} = 0.1s$ with 108, 3 ms.

[1]We have selected rather low bit rates in this example in order to depict a case where the verification fails. The intended hardware would support communication speeds up to 50000 Bit/s.

Updating of the previous system to the extended one requires the following steps:

First the additional node must be connected to the network. Then the plug and participate facility [6] of TTP/A automatically assigns a free node ID to the new node. Now the modified schedules can be downloaded to the nodes, either on-line via the CP interface, if the RODL files of the IFS reside in the RAM of the smart transducer, or off-line by using conventional programming tools. Since the configuration state of the fusion job is also represented in the IFS, the configuration of the fusion job can be also changed over the CP interface, so that it knows how many source jobs are connected to it.

With the above mentioned configuration facilities and the scheduling tool it is now possible to support the fully automatic execution of configuration operations, thus greatly enhancing the flexibility of system management.

5 Conclusion

In this paper we proposed a model-based approach to support flexibility for a time-triggered smart transducer system.

The application is described as a network of interacting jobs and temporal end-to-end requirements using a formal representation of this information in XML created in earlier work. The XML data is used as input to a scheduling tool that creates the static schedule and verifies the specified temporal requirements.

It is possible to change or extend the application by feeding the new description into the scheduling tool and upload the new configuration via the configuration and planning interfaces of the smart transducers in the network.

If the available bandwidth is not sufficient to fulfill the temporal requirements of the application, the tool reports on a deadline miss, otherwise it can be guaranteed that the generated schedule meets the specified timing requirements.

Since the scheduling tool parses the XML descriptions off-line on a maintenance computer, the resource requirements on the embedded nodes are extremely low. Implementation experiences have shown that the smart transducer interface can be implemented on low-cost 8-Bit microcontroller nodes.

Acknowledgments

This work was supported in part by the European IST project DECOS under contract No. IST-511764 and by the FIT-IT project "SDSTI" under contract No. 808693. FIT-IT is funded by the Austrian ministry for transport, innovation and technology (BM-VIT). We would like to thank the anonymous reviewers and our colleagues at the Institute of Computer Engineering for their valuable comments on earlier versions of this paper.

References

[1] The Hansen report on automotive electronics, Nov. 1998. 11(9), RYE, NH USA.

[2] L. Almeida, J. A. Fonseca, and P. Fonseca. Flexible time-triggered communication on a controller area network. In S. Goddard, editor, *Work-in-Progress Session of the 19th IEEE Real-Time Systems Symposium*, pages 23–26, Madrid, Spain, Dec. 1998.

[3] J. Berwanger, C. Ebner, A. Schedl, R. Belschner, S. Fluhrer, P. Lohrmann, E. Fuchs, D. Millinger, M. Sprachmann, F. Bogenberger, G. Hay, A. Krüger, M. Rausch, W. O. Budde, P. Fuhrmann, R. Mores, Bayerische Motoren Werke, DaimlerChrysler, Dependable Computer Systems, Motorola, and Philips. FlexRay–The communication system for advanced automotive control systems. *SAE World Congress 2001, Detroit, Michigan, USA*, Mar. 2001.

[4] V. Claesson. *Efficient and Reliable Communication in Distributed Embedded Systems.* PhD thesis, Chalmers University of Technology, Department of Computer Engineering, Göteborg, Sweden, 2002.

[5] W. Elmenreich. *Sensor Fusion in Time-Triggered Systems.* PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, 2002.

[6] W. Elmenreich, W. Haidinger, P. Peti, and L. Schneider. New node integration for master-slave fieldbus networks. In *Proceedings of the 20th IASTED International Conference on Applied Informatics (AI 2002)*, pages 173–178, Feb. 2002.

[7] W. Elmenreich, S. Pitzek, and M. Schlager. Modeling distributed embedded applications on an interface file system. In *Proceedings of the Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 175–182, May 2004.

[8] T. A. Henzinger, C. M. Kirsch, M. A. A. Sanvido, and W. Pree. From control models to real-time code using Giotto. *IEEE Control Systems Magazine*, 23(1):50–64, 2003.

[9] C. Jones, M.-O. Killijian, H. Kopetz, E. Marsden, N. Moffat, D. Powell, B. Randell, A. Romanovsky, R. Stroud, and V. Issarny. Final version of the DSoS conceptual model. *DSoS Project (IST-1999-11585) Deliverable CSDA1*, Oct. 2002. Available as Research Report 54/2002 at http://www.vmars.tuwien.ac.at.

[10] H. Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers, Boston, Dordrecht, London, 1997.

[11] H. Kopetz. An integrated architecture for dependable embedded systems. In *Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems*, 2004.

[12] H. Kopetz, M. Holzmann, and W. Elmenreich. A universal smart transducer interface: TTP/A. *International Journal of Computer System Science & Engineering*, 16(2):71–77, Mar. 2001.

[13] H. Kopetz et al. Specification of the TTP/A protocol. Technical report, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, Sept. 2002. Version 2.00.

[14] H. Kopetz et al. Specification of the TTP/A protocol. Technical report, Technische Universität Wien, Institut für Technische Informatik, Vienna, Austria, Sept. 2002. Version 2.00.

[15] J. Lisner. A flexible slotting scheme for tdma-based protocols. In *ARCS 2004 Workshop on Dependability and Fault Tolerance*, Augsburg, Germany, Mar. 2004.

[16] R. Obermaisser. *Event-Triggered and Time-Triggered Control Paradigms.* Springer, 2004.

[17] P. Veríssimo and A. Casimiro. The timely computing base model and architecture. *IEEE Transactions on Computers*, 51(8):916–930, Aug. 2002.