The GENESYS Architecture: A Conceptual Model for Component-Based Distributed Real-Time Systems

R. Obermaisser and B. Huber

Vienna University of Technology, Austria

Abstract. This paper proposes a conceptual model and terminology for component-based development of distributed real-time systems. Components are built on top of a platform, which offers core platform services as the basis for the implementation and integration of components. The core platform services enable emergence of global application services of the overall system out of local application services of the constituting components. Therefore, the core platform services provide elementary capabilities for the interaction of components, such as message-based communication between components or a global time base. Also, the core services are the instrument via which a component creates behavior that is externally visible at the component interface. In addition, the specification of a component's interface builds upon the concepts and operations of the core platform services. The component interface specification constrains the use of these operations and assigns contextual information (e.g., semantics in relation to the component environment) and significant properties (e.g., reliability requirements, energy constraints). Hence, the core platform services are a key aspect in the interaction between integrator and component developer.[1]

1 Introduction

It is beyond doubt that complex embedded real-time systems can only be reasonably built by following a component-oriented approach. The overall system is divided into components that can be independently developed and serve as building blocks for the ensuing computer system.

A platform is required both as a baseline for the component developers and for the integrator to combine the independently developed components to the overall system. For the latter purpose, the platform needs mechanisms for the interaction between components. However, these elementary interaction mechanisms not only serve for the coupling between components. Beyond that, the elementary interaction mechanisms of the platform determine how component behavior comes into existence. The use of these mechanisms is the instrument via which a component generates behavior that is externally visible. From the point of view of a component's interface to the platform, the use of these elementary interaction mechanisms (along with associated meaning in the application context) is what constitutes a component.

In this paper, we define the notion of a platform and its elementary interaction mechanisms that we denote *core platform services*. We argue that the core platform services

[1] This work has been supported in part by the European research project INDEXYS under the Grant Agreement ARTEMIS-2008-1-100021.

are the result of emergence from local implementations of platform functionality at the components in conjunction with shared platform functionality (e.g., network switches, network-on-a-chip). On their behalf, the core platform services permit the interaction between application services and enable another form of emergence: the emergence of application services out of more elementary local application services of components.

The presented conceptual model is a result of discussions on embedded system architectures within the GENESYS project and the ARTEMIS Strategic Research Agenda. Experts from five domains of embedded systems (i.e., automotive, avionics, industrial control, mobile systems, consumer electronics) were involved in devising the underlying concepts. The model serves as the conceptual basis of the cross-domain architecture developed within the GENESYS project.

The contribution of this paper is the analysis and conceptualization of the relationship between embedded platforms and applications. In this regard, we go beyond related work on specific component-based frameworks (e.g., CORBA [17], AUTOSAR [19] to name two widely-used frameworks). We analyze platform services and their implications on the development of application services, ranging from platform capabilities, to instruments for behavior generation, to concepts for specifying behavior.

2 Basic Concepts

This section introduces fundamental concepts that will be used in the paper.

System: We use the definition of a *system* introduced in [2]: *an entity that is capable of interacting with its environment and is sensitive to the progression of time.* The environment is in principle another system. The environment takes advantage of the existence of a system by producing input for the system and acting on the output of the system.

A system combines physical and logical aspects. As a consequence, behavior can be associated with a system taking into account both the value and time domain. This definition excludes, for example, a software module without associated hardware.

In general, systems are hierarchic and can on their behalf be recursively decomposed into sets of interacting constituting systems. The constituting elements of a system are denoted as components.

Time: The temporal awareness of systems requires a model of time. We assume a model based on Newtonian time, where the continuum of real-time can be modeled by a directed timeline consisting of an infinite set of instants [20]. In a distributed computer system, the progression of time is captured with a set of physical clocks. Since any two physical clocks will employ slightly different oscillators, clock synchronization is required to bring the time of clocks into close relation with respect to each other. A measure for the quality of clock synchronization is the precision, which is defined as the maximum offset between any two clocks during an interval of interest.

Service: A *service* is what a system delivers to its environment according to the specification. Through its service, a system can support the environment, i.e., other systems that use the service. The specification for a system defines the service. Given a concrete

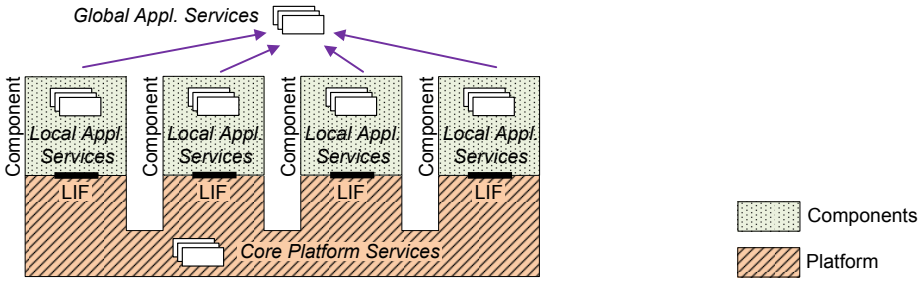

Fig. 1: Architectural Model

paradigm of interaction between systems, the notion of a service can be refined. For example, in context of message-based interaction, the service of a system can be defined as *the sequence of intended messages that is produced by a system in response to the progression of time, input and state* [2, page 28]. An overview of formalisms for the definition of services in different interaction paradigms can be found in [4] (e.g., Statecharts, Specification and Description Language (SDL)).

***Behavior**:* In the presence of faults (e.g., design fault in the implementation), a system can violate its specification. In this case, the system exhibits a *failure* [1] instead of its specified service. We use the term *externally visible behavior* (or behavior for short) as a generalization of the notions of service and failure: behavior $=$ service $\cup$ failure

The correct behavior (as defined by the specification) is the system's service. The faulty behavior (violation of the specification) is a failure. In the absence of a specification, we can only reason about the behavior of systems.

3 Architectural Model

The overall system consists of two types of constituting systems: a set of *components* and an underlying *platform* (cf. Figure 1). Based on this system structure, we can distinguish different types of services: *global application services* of the overall system, *local application services* of components, and *platform services*.

3.1 Users of the Architectural Model

When developing a system that follows this architectural model, we can differentiate two roles: *component developers* and the *integrator*.

The component is the unit of delegation and the unit of integration. Using the platform, the integrator is responsible for binding together the components to an overall system with global application services. The platform offers the means to integrate the components based on the specification of the components' local application services.

The component developers are concerned with the design and implementation of individual components. A component developer can delegate subtasks for the realization of a component to suppliers/subcontractors. Nevertheless, the component developer

delivers an entire component with a local application service to the integrator. The integrator need not be aware of the inner structure of the component and the involvement of suppliers/subcontractors.

3.2 Components and Application Services

A component is a self-contained building block of the computer system. The borderline between a component and the platform is called the component's *Linking Interface (LIF)* [9]. At the LIF, the component provides its *local application services* to the other components. A local application service is the intended behavior of a component at the LIF. The component exchanges information with other components at the LIF and the specification of the local application service must cover all aspects that are relevant for the integration of the component with other components:

(1) Values. The syntax of the information exchanged at the LIF needs to be defined. In addition, relationships between inputs and outputs are specified.

(2) Timing: In a real-time system, the specification of the LIF encompasses temporal constraints, e.g., for consuming inputs or producing outputs.

(3) Relationship to the (natural) environment of the computer system: For each component that interacts with the environment of the computer system, the LIF specification must capture the (semantic) relationship between the information exchange at the LIF and the interaction with the environment. While abstracting from the details of the component's local interfaces (e.g., I/O interfaces or fieldbuses), the semantics of the LIF interaction in relation to the component environment need to be described. Due to the inability to fully formalize the relationship to the environment [10], natural language or ontologies are examples of suitable specification methods. For example, information provided at the LIF to other components can resemble entities in the natural environment with a given delay and originate from a sensor. Likewise, information consumed at the LIF can cause an effect on the natural environment via an actuator. In addition to the value domain, this relationship must be specified in the temporal domain. For example, the lag between sensory information at the LIF and the state of the environment is of concern (cf. temporal accuracy [8]).

In addition, many other aspects can be relevant for the specification of the LIF, e.g., reliability, energy, security.

3.3 Platform and Platform Services

The platform is the foundation for development and integration of components. It consists of platform services, which we denote as core platform services (in order to discriminate them from platform-like services within components that we will call optional platform services). More precisely, a platform is essential for two reasons:

(1) Baseline for development of components. Using the platform services, the component developer establishes the local application services of a component. Component developers need a starting point for realizing components. The platform offers a foundation on top of which application-specific functionality can be established. This

foundation consists of generic services, which are required to be useful in many specific components. Although most of these services could also be realized within the components, their availability in the platform simplifies the component development.

As an example, consider a sensor component that periodically samples the lateral acceleration in a car and produces a message on the LIF with this measurement. The local application service of this component would be 'acceleration measurement'. An example of a platform service that can be used to construct this application service would be a time service. Such a time service can provide the periodic sampling points (e.g., with respect to a global time base).

Using platform services, recurring problems are solved once-and-for-all in the platform without the need to redevelop them in every component. Principally, the development of components becomes easier if more functionality is offered by the platform. However, overloading the platform with a plethora of functionality is likely to lead to a high overhead. The reason for the overhead is that part of the functionality will be too specific to be applicable except for a few very specialized components. Furthermore, the complexity of the platform will increase. Thus, the likelihood of design faults in the platform will increase. Such a design fault in the platform is of particular severity. While a design fault of a component would affect this specific component, potentially all components can be affected by a design fault in the platform. All components build upon the platform and depend on its correctness. The issue of the complexity of the platform and the susceptibility to design faults is of particular importance for safety-critical applications that need to be certified.

(2) Framework for integration of components. Besides serving as the baseline for the establishment of local application services of components, the platform services are an instrument for emergence. The platform services enable the emergence from local application services of the components to global application services of the system.

Therefore, the platform offers mechanisms to compose the overall system out of the independently developed components. These mechanisms include *communication services* enabling the exchange of information between components. In addition, other services can serve as a useful basis for integration, e.g., *fault isolation services* [12] that prevent component failures from propagating between components or *clock synchronization services* [13] to establish a common notion of time.

Following up on the previous example, the introduced component can be integrated with other in-vehicle components, thereby composing the speed measurement service with local application services of other components. The result is the emergence of a global application service (e.g., passive safety in the example with the lateral acceleration measurement) out of local application services (e.g., brakes, steering, and suspension in addition to the mentioned acceleration measurement service).

4 Core Platform Services

This section characterizes the constituting elements of the platform: the core platform services. The introduced concept is exemplified using the GENESYS architecture.

4.1 Three Roles of Platform Services: Platform Capabilities, Instrument for Behavior Generation and Concepts for Service Specification

A core platform service is an elementary building block of the platform. Inversely, the set of core platform services defines the platform.

Each core platform service is a *capability of the platform* that is offered to the components. An example of a core platform service is *'message multicasting'* that enables components to interact by the exchange of messages. In its simplest form, this service would consume messages from one port and transport these messages to a set of destination ports with defined properties (e.g., latency, reliability). This core platform service would enable a component to deposit messages at a port in order to be delivered via multicast communication to ports belonging to other components.

Secondly, the core services are the instrument by which a component generates behavior at the LIF. The use of the core services results in activities that can be perceived at the LIF from outside the component. The core services offer elementary operations, a sequence of which forms behavior at run-time. For example, in case of the platform service *'message multicasting'*, the instrument would be the ability to send messages.

The instrument for behavior generation is a part of the capability. The capability is, however, more than empowering the component to generate behavior. The capability is also concerned with the reaction to the component behavior in the platform. In particular, the capability links behavior at different components (e.g., output of one component becomes input of another component).

In addition, the core services provide the underlying concepts for the specification of component services. This statement about behavior relates to behavior on a meta-level. Given a set of core services, different service specification languages are possible that use these concepts.

4.2 Core Platform Services of the GENESYS Architecture

To give examples of core platform services, we will give an overview of the services of the GENESYS architecture [16]:

(1) Periodic and sporadic messaging. The GENESYS architecture uses the communication paradigm of messaging. The concept of a *message* is an atomic structure that is formed for the purpose of inter-component communication. A messages encompasses data and control information, where the control information addresses timing, acknowledgement, and addressing. In addition, the syntax of the data is defined and the message is ascribed meaning in the application context. Messaging has been chosen as a core platform service of GENESYS, because messaging provides an ideal abstraction level for embedded real-time systems. In particular, messages offer inherent consistency, synchronization, and the timing is explicit compared to shared memory abstractions. Nevertheless, messaging is a universal model and other communication paradigms can be realized on top of messages (e.g., a virtual shared memory [11]). Depending on the message timing, the core service distinguishes periodic and sporadic messages. The timing of *periodic messages* is defined by a period and phase. Sporadic messages possess a minimum message interarrival time.

In addition to the concept of messaging (as explained above), the platform service is

associated with a platform capability and an instrument for behavior generation. The capability associated with this platform service is the transport of messages from a specific sender-component to a set of receiver-components (i.e., multicast). The capability involves non-functional properties, such as the the reliability (e.g., probability for message omission failure) and temporal properties (e.g., end-to-end latency, bandwidth).
The instrument for behavior generation is the ability for the transmission of a message, i.e., the production of a message by the sender-component. In case of periodic communication, the sender-component performs an update-in-place of a state message. For sporadic communication, the sender-component places an event message into an outgoing message queue.

(2) Global time. The concept of a global time is a counter value that is globally synchronized across components. If the counter value is read by a component at a particular point in time, it is guaranteed that this value can only differ by one tick from the value that is read at another component at the same point in time. The concept of a global time base enables service specifications with temporal constraints w.r.t. global time (e.g., adherence to the action lattice of a sparse time base [7], specification of phase of periodic messages, etc.). In addition, activities at different components can be temporally coordinated, e.g., avoiding collisions by design or creating phase-aligned transactions [14].
The capability of the platform is the execution of a clock synchronization algorithm. For example, in case of a distributed clock synchronization algorithm, local views regarding the global time are exchanged. Thereafter, a convergence function is applied and the local clocks at the components are adjusted (e.g., using rate correction).
The instrument for behavior generation comprises the use of the global time in activities at the LIF, such as the transmission or reception of a message at specific instants of the global time base.

(3) Network diagnosis and management. The membership vector contains globally consistent information about the operational state of every component (e.g., correct or faulty) within a given membership delay. The platform capability involved in this core platform service is the construction of a membership vector. Firstly, the platform performs error detection for individual components. For example, the platform can determine whether the temporal properties of the messages are satisfied. Secondly, the platform needs to perform agreement on a globally consistent view on the operation state of the components. A membership vector offers an instrument to generate behavior by taking different actions depending on the operational state of components. For example, in a brake-by-wire car with four components associated with the four wheels of the car, components can react differently to messages with braking actuation values when they possess knowledge about the failure of the component at another wheel.

(4) Reconfiguration. Reconfiguration is concerned with adapting the platform depending on application contexts, the occurrence of faults (e.g., permanent failure of a component) and the availability of resources (e.g., low battery power). To accomplish these goals, the other platform services are altered within given limits, e.g., by introducing a new message or altering the timing in the first core service. The capability of the platform involves the processing of triggers for reconfigurations, such as the acceptance of requests for reconfigurations or monitoring resource availability. Secondly, reconfiguration needs to be executed, e.g., by creating a new communication schedule

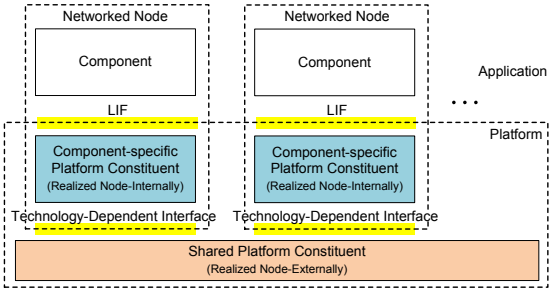

Fig. 2: Deployment of Core Platform Services: A computer system consists of *networked nodes* (nodes for short) that are conjoined through a *shared platform constituent*.

for periodic messaging. The instruments for behavior generation include the operations to request reconfiguration. For example, a component can request the ability to send an additional periodic message.

4.3 Realization of Platform Services

Two structural elements can be distinguished in the realization of the platform: component-specific and shared constituents of the platform (cf. Figure 2).

Firstly, the platform comprises a shared part that is not specific to any component in particular. This *shared platform constituent* includes the communication medium for the communication between the components. The communication medium can be a bus (e.g., CAN), a routed network (e.g., IP network with routers), a starcoupler (e.g., FlexRay starcoupler), or even air or vacuum in case of a wireless connection.

In addition to the shared part of the platform, each component is associated with a component-specific platform constituent. Core platform services can be realized using distributed implementations. For example, a common time service usually depends on a distributed clock synchronization capability. The common time service provides each component with access to a local clock, which is repeatedly adjusted to ensure a bounded precision in relation to all other clocks of the system.

In addition to distributed implementations of core platform services, the *component-specific platform* constituents serve a second purpose. They map the technology-independent LIF to a technology-dependent interface. The platform maps the technology-independent abstractions provided by the core platform services (e.g., messaging concept) to the physical world (e.g., a specific communication protocol and physical layer).

The benefit of a technology-independent LIF is the abstraction over different implementation technologies, which are not fixed when the LIFs are specified. Thus tradeoffs between different implementation technologies can be performed later in the development process. For example, a general purpose CPU implementation offers more flexibility for modifications and often involves less implementation effort. An FPGA, on the other hand, can have superior non functional properties (e.g., energy efficiency, silicon area). In addition, as a system evolves the component implementation for a given LIF can be changed without requiring changes at other components.

A component together with its component-specific platform constituent typically forms a self-contained physical unit (called *networked node*).

The integrator will provide to component developers a component-specific platform constituent as a hardware/software container for the incorporation of a particular component. For this purpose, the integrator can use platform suppliers (similar to Integrated Modular Avionics (IMA) platform suppliers in the avionic domain [3]), which provide implementations of the core services. The networked node is the result of combining the supplier's component with the component-specific platform constituent.

During integration the networked node (e.g., an ECU in the automotive domain) is combined with the shared platform-constituent. For example, the networked node is connected to an Ethernet network in case of Ethernet as a communication medium.

Figure 2 depicts the layout of the platform and its structural elements graphically. In order to illustrate the introduced concepts, a few examples will be outlined in the following: Let's first consider *wireless sensor devices* as networked nodes. The component-specific platform constituent contains a wireless transceiver/receiver that maps the message-based abstraction to radio transmissions. The wireless sensor device will also contain a host (e.g., general purpose CPU, FPGA) as the component providing the application services. The shared platform-constituent can simply be the communication medium (e.g., air) or comprised of wireless base stations.

A second example of a networked node is an *Electronic Control Unit (ECU)* in a car. The shared platform-constituent can be an automotive communication system (e.g., Controller Area Network (CAN) [6], FlexRay starcoupler [18]). The component-specific platform constituent is a communication controller (e.g., CAN controller, FlexRay controller) to the respective communication network.

Another example is an *IP core on a Multi-Processor System-on-a-Chip (MPSoC)*. Here the shared platform-constituent is the interconnect for the communication between the IP cores, e.g., a network-on-a-chip. The component-specific platform constituent provides the access to the on-chip interconnect (e.g., Trusted Interface Subsystem (TISS) [15], Memory Flow Controller [5]).

4.4 Emergence of Core Platform Services

The *core platform services are emergent services* of the services provided by the shared and component-specific platform constituents. Consider for example the global time service. Each component-specific platform constituent comprises a local clock synchronization capability. The local clock synchronization capability is responsible for repeatedly computing correction terms to be applied via state or rate correction to the local clock. For this purpose, the difference of the local clock to other clocks is determined via synchronization messages or the implicitly known reception times of messages.

Through the interplay of the local clock synchronization capabilities, a global notion of time emerges. As discussed in Section 3.3, this emerging service can be used to specify the LIF and as a basis for the realization of application services. Likewise, the periodic message transport service is an emerging service. The periodic message transport service is more than the sum of the capabilities of the component-specific platform constituents to perform periodic transmissions/receptions of messages.

5 Component Model

From the point of view of the LIF, a component provides application services expressed w.r.t. the core platform services. The integration of a system out of components is helped through the existence of the core platform services. The core platform services introduce a uniform instrument across all components for generating component behavior.

However, component developers can favor different sets of platform services to express application behavior. Firstly, legacy applications have been developed for different platforms with different sets of platform services. In order to avoid a complete redevelopment of these legacy applications, a mapping of the legacy platform services to the uniform core platform services is desirable. In addition, different domains can have unique requirements regarding the capabilities of the platform. For example, a safety-related control subsystem can build upon platform services for active redundancy. A multimedia system, on the other hand, might have to cope with a large number of different configurations and usage scenarios. Hence, a multimedia system can need dynamic reconfiguration capabilities that go beyond the support of the core platform services.

We introduce a layered component model in order to resolve this discrepancy between uniform core services and the need for application-specific platform services. For providing application services, the component can employ an intermediate form of the application services: application services expressed w.r.t. optional platform services.

5.1 Optional Platform Services

On top of the core services, the optional platform services establish higher-level capabilities for certain domains (e.g., control systems, multimedia). Besides, the optional services within a component establish an instrument for behavior generation just as the core services do.

In contrast to the core platform services, the optional platform services provide additional constructs that are not always needed or useful in all types of components. The optional services reflect the heterogeneity of a system by no longer enforcing uniformity of platform services throughout the system. A particular optional platform service can prove to be useful in one component, whereas the deployment of this service in another component might impede the component development. Consider for example an optional service for dynamic reconfiguration, which would make difficult the certification of a safety-related component.

The instrument for behavior generation of the optional platform services must have a defined mapping to the underlying instrument for behavior generation of the core platform services. Hence, the optional platform services transform the application services towards the core platform services.

5.2 Equivalence of Optional Services and Application Services

In the following we will point out that optional platform services are application services that have been internalized in a component. An application service exploits another application service by using the core platform services. In contrast, an application service exploits the optional platform services directly, i.e., without using the core

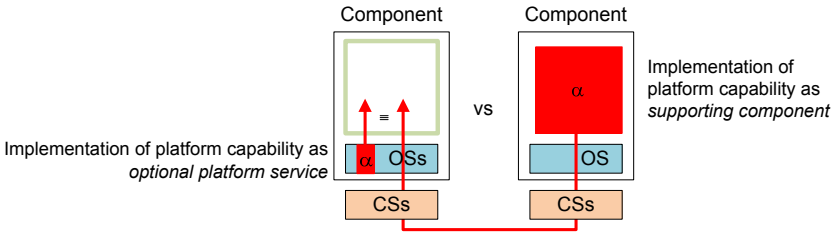

Fig. 3: Optional Platform Service vs. Supporting Component

platform. However, from a logical point of view it is equivalent whether a particular capability is introduced into a system via an application service or via an optional platform service. Figure 3 depicts that a certain capability α can be provided by an optional platform service or as an application service in a supporting component.

Preexisting Standard Components A preexisting standard component is a component that provides an application service that can be used for the construction of other application components. The integrator defines which preexisting standard components exist in a system and provides their LIF specifications to the suppliers. Preexisting standard components are visible to the integrator and their services are expressed w.r.t. to the core platform services. An example of a preexisting standard component would be a voting component for a Triple Modular Redundancy (TMR) configuration. This component accepts three redundant messages and outputs a single message based on a majority decision. Another example of a preexisting standard component would be an *encryption component*. Such a component accepts 'plaintext' messages and outputs encrypted messages by applying a suitable ciphering method (e.g., DES). Such a component would facilitate the interaction between a subsystem with trusted components and a subsystem with untrusted components (e.g., a public network or the Internet).

Optional Platform Services The component developer is responsible for the optional platform services. For the integrator, who combines components based on their LIFs, the optional platform services are invisible. The integrator cannot discern a component that implements the LIF behavior directly from a component that employs optional platform services.

6 Conclusion

The contribution of the paper is a conceptual model for component-based distributed real-time systems. In particular, the notion of a platform in relation to components has been analyzed. The platform enables the emergence of global application services out of local application services of components. Starting from a decomposition of a platform into core platform services, we have elaborated on the implications for the design and implementation of components. The core platform services represent the concepts

for the specification of interfaces, the instrument for the generation of component behavior, and the capabilities for the interaction of components. In addition, different organizational roles in a component-based development were analyzed: the integrator, component suppliers and platform suppliers.

References

1. A. Avizienis, J. Laprie, and B. Randell. Fundamental concepts of dependability. In *Proc. of ISW 2000. 34th Information Survivability Workshop*, pages 7–12. IEEE, 2000.
2. C. Jones et al. DSoS conceptual model. *DSoS Project (IST-1999-11585)*, December 2002.
3. P. Conmy, J. McDermid, M. Nicholson, and Y. Purwantoro. Safety analysis and certification of open distributed systems. In *Proc. of the Inter. System Safety Conference Denver*, 2002.
4. A.M. Davis. A comparison of techniques for the specification of external system behavior. *Communication of the ACM*, 31, 1988.
5. IBM, Sony, and Toshiba. Cell broadband engine architecture. Technical report, 2006.
6. Int. Standardization Organisation, ISO 11898. *Road vehicles – Interchange of Digital Information – Controller Area Network (CAN) for High-Speed Communication*, 1993.
7. H. Kopetz. Sparse time versus dense time in distributed real-time systems. In *Proc. of 12th Int. Conference on Distributed Computing Systems*, Japan, June 1992.
8. H. Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Boston, Dordrecht, London, 1997.
9. H. Kopetz and N. Suri. Compositional design of rt systems: a conceptual basis for specification of linking interfaces. In *Proc. of the Sixth IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing*, pages 51–60. IEEE, May 2003.
10. H. Kopetz and N. Suri. On the limits of the precise specification of component interfaces. In *Proc. of the Ninth IEEE Int. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003)*, pages 26–27, 2003.
11. E. Kühn. *Virtual Shared Memory for Distributed Architectures*. Nova Science Pub Inc, 2002.
12. J.H. Lala and R.E. Harper. Architectural principles for safety-critical real-time applications. In *Proc. of the IEEE*, volume 82 of *1*, pages 25–40, January 1994.
13. L. Lamport. Using time instead of timeouts for fault-tolerant distributed systems. *ACM Trans. on Programming Languages and Systems*, pages 254–280, 1984.
14. R. Obermaisser, C. El-Salloum, B. Huber, and H. Kopetz. Modeling and verification of distributed real-time systems using periodic finite state machines. *Journal of Computer Systems Science & Engineering*, 22(6), November 2007.
15. R. Obermaisser, C. El Salloum, B. Huber, and H. Kopetz. The time-triggered system-on-a-chip architecture. In *Proc. of the IEEE Int. Symposium on Industrial Electronics*, 2008.
16. R. Obermaisser, C. El Salloum, B. Huber, and H. Kopetz. Fundamental design principles for embedded systems: The architectural style of the cross-domain architecture GENESYS. In *Proc. of the 8th IEEE ISORC*, pages 3–11, Tokyo, Japan, March 2009.
17. Object Management Group, Needham, MA 02494, U.S.A. *The Common Object Request Broker: Architecture and Specification*, July 2002.
18. R. Belschner et al. FlexRay – requirements specification. Technical report, BMW AG., DaimlerChrysler AG., Robert Bosch GmbH, and General Motors/Opel AG, 2002.
19. T. Rolina. Past, present, and future of real-time embedded automotive software: A close look at basic concepts of AUTOSAR. In *Proc. of SAE World Congress*, April 2006.
20. G.J. Whitrow. *The Natural Philosophy of Time*. Oxford Univeristy Press, 2nd edition, 1990.