On the Design of Distributed Time-Triggered Embedded Systems

Hermann Kopetz
Technical Univ. of Vienna, Austria
hk@vmars.tuwien.ac.at

Received 27 June 2008; Accepted 29 August 2008

The cognitive constraints of the human mind must drive the decisions in architecture and methodology design in order that the systems we build are comprehensible. This paper presents a methodology for the design of time-triggered embedded systems that leads to understandable artifacts. We lift the design process to a higher level of abstractionto the level of computational components that interact solely by the exchange of messages. The time-triggered architecture makes it possible to specify the temporal properties of component interfaces precisely and provides temporally predictable message communication, such that the precise behavior of a large design can be studied in the early phases of a design on the basis of the component interface specifications. This paper shows how the cognitive simplification strategies of abstraction, partitioning and segmentation are supported by the time-triggered architecture and its associated design methodology to construct evolvable embedded systems that can be readily understood and modified.

Categories and Subject Descriptors: Systems Design and Computer Architecture **[Embedded Systems]**

General Terms: Real-Time Systems, Robustness, Time-Triggered

Additional Key Words and Phrases: Interface Specification, Composability, Evolvability, Cognitive Science, Complexity, Simplification Strategies

1. INTRODUCTION

As a consequence of Moores law, the cost of a logic function has decreased by more than six orders of magnitude since the invention of the transistor fifty years ago. No comparative improvement in the design productivity of embedded systems has been realized over this period. As a consequence the cost of design and validation forms the major cost part of the overall lifecycle cost of many of todays embedded systems.

Some of the successfully fielded embedded systems have evolved over the years and

reached, due to the need for continued maintenance and enhancements [Belady and Lehman 1976], a level of complexity that challenges human comprehension. There is thus a need for an architecture and an associated design methodology that leads to understandable and evolvable designs. The idiosyncrasies of human cognition, as researched in the field of cognitive science [Reisberg 2001], must drive design decisions in architecture and design methodology development, such that the characteristics of the evolving artifact match the constrained cognitive capabilities of the human mind [Kopetz 2008].

One way to improve the understandability of a design and the design productivity comes about if we lift the design process to a higher level of abstraction, to the level of computational components that interact solely by the exchange of messages. The clear identification of a component as a unit of functionality, comprehension and data transformation, and the explicit description of the components interconnection to its environment by the well-known message metaphor establish stable structures that help to support the cognitive concept formation and the reasoning about the system behavior. The possibility for a wide reuse of well-defined and self-contained existing components, based solely on their interface specification, reduces the development and integration efforts and leads to the aspired significant improvement in the design productivity. A prerequisite for such a design methodology is the availability of architectural services that support the precise interface specification of the behavior of components and the behavior of communication systems in the domains of value and time. The availability of a global time in the time-triggered architecture [Kopetz and Bauer 2003] provides the looked-for means for such a precise interface specification in the temporal domain.

The rest of the paper is structured as follows. In Section 2 we introduce basic concepts of time-triggered systems. Section 3 gives an overview of our design methodology. Section 4 deals with the design of the platform independent model (PIM) and discusses the operational and semantic specification of the interfaces of a component. Section 5 explains the message-based communication among components. The topic of component restart and evolvability is covered in Section 6. The implementation of the platform specific model (PSM) is dealt with in Section 7. The paper terminates with a Conclusion in Section 8.

2. TIME-TRIGGERED SYSTEMS

In this Section we introduce the essential concepts of time-triggered systems.

At the *system level*, we distinguish between the distributed computer system under consideration (we call it a *cluster*) and its *environment*. A cluster consists of a set of components, connected by a cluster-internal time-triggered communication system that transports messages among the components.

A *component* is a hardware/software unit that accepts input messages, provides a useful service, maintains internal state, and produces after some *elapsed physical time* output messages containing the results. A component is thus an isolated and identifiable functional unit of data transformation and comprehension and forms an abstract high-level concept in the mental model of the system behavior. The syntax and semantics of the component service must be specified in a component-interface

model without reference to the detailed component internals, i.e., the concrete component implementation. In an embedded system the interface model must include the temporal parameters of the intended component behavior. Preferably, the interface model should specify the data-transformation algorithms of the component in an executable form, such that the algorithms can be automatically translated into the selected implementation technology.

A *message* is an atomic data structure that is formed for the purpose of *transmitting data* and *control signals* from a sending component at a given instant to one or more receiving components that receive the message at a later instant. A message should be the only means for a component to interact with its environment. The message concept does not make any assumption about (abstracts form) the specific transport mechanism or about the meaning of the bit-vector contained in the data field of the message. However, the physical time it takes to transport a message from the sending component to the receiving component is part of the control aspect of the message concept.

A *cycle* is a key concept in any time-triggered system. A cycle is a period of physical time between the repetitions of regular events. A cycle is specified by the duration of its period and the position of its start, the *cycle start phase*, relative to some given global time reference. In order to avoid the exponential explosion in the combination of cycles and thus a massive increase in the complexity, the rule is established that all cycles must be in a *harmonic relationship*, i.e., the duration of any cycle must be a power of two of the duration of the shortest allowed cycle. If we use a binary representation of physical time, such as the IEEE 1588 time standard [IEEE 2002], then any cycle duration can be defined by specifying a particular bit in this binary time-representation. The start of a cycle, i.e., the cycle start phase, can then be identified by specifying the oset of the start instant of the cycle from the start instant of the respective period in the global time representation.

In a time-triggered system a global time of known precision is available within the given ensemble of components, and a cycle is assigned to every time-triggered process. At every cycle-start, a control signal is generated by the architecture to trigger this time-triggered process. For example, a time-triggered message is sent or a time-triggered action is started whenever the cycle-start associated with this process occurs.

3. DESIGN METHODOLOGY

In this Section we give an overview over the design methodology that leads to understandable and robust systems built within the context of the time-triggered architecture.

3.1 Platform Independent Model

Computer-system design normally starts with a conceptualization of the intended *high-level behavior* of the planned system. For example, when we intend to build a computer-controlled braking system for a car, we start from a high-level behavioral specification that relates the inputs to the outputs in the domains of value and time:

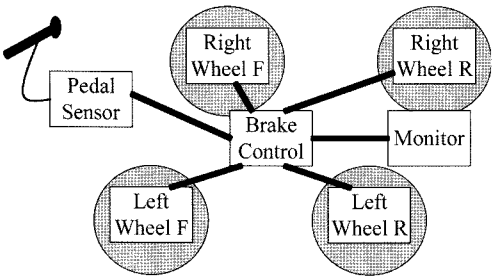

Figure 1. Distributed Brake Control System.

When the brake pedal is pressed, the computer should initiate the braking action within one millisecond.

This high-level behavioral specification can be decomposed into a set of *components* and *messages*. Figure 1 shows an example of such a decomposition that leads to a distributed application system (DAS) for braking a car. In this example we introduce seven components, one component, the *pedal sensor component*, for capturing the brake pedal position, one central component, the *brake control component*, for calculating the brake pressure, four *wheel components* for actuation of the brake pressure and measuring the speed at the four wheels, and one *monitoring component* to support diagnosis and restart. The pedal-sensor components send messages about the current pedal positions to the brake-control component. The four wheel components send messages containing the current speed of each wheel to the brake-control component and receive, from the brake-control component, messages indicating the brake pressure that should be applied at each wheel. Finally, the monitoring component receives all messages to check the sanity of the components, perform on-line diagnosis in case of an error and initiate a quick component restart if required.

We call such a high-level component-based decomposition of a distributed system a *Platform-Independent Model (PIM)* [OMG 2001] if the intended behavior of the PIM components and PIM messages is specified in the domains of value and time without any reference to a concrete execution platform.

3.2 Platform Specific Model

In a later phase of the design process, the PIM model of our application is transformed such that it can be executed on the selected target hardware platform [Huber et al. 2006]. We call the platform-specific component-based description of a distributed system the *Platform Specific Model (PSM)* [OMG 2001].

For example in our example of a brake system a PIM component could later be transformed to a PSM component that provides the specified functionality either by software on the selected CPU, or by an FPGA (field programmable gate array) or by an ASIC (application specific integrated circuit).

It is a fundamental characteristic of our design methodology that we assume a strict one-to one mapping of PIM components to PSM components.

A component is thus a *stable concept* that remains the same at the level of service conceptualization and at the level of implementation, thus contributing to the principles of *stability* and *cognitive economy.* Since the interface behavior of a PIM component and the corresponding PSM component are exactly alike in the domains of value and time, any behavioral analysis that is carried out at the PIM level will remain valid at the PSM level. This strict one-to-one mapping of PIM components to PSM components helps to solve the *technology obsolescence problem* as well: If a given application has to be ported to a new target platform (since the old hardware is not available any more), it is only necessary to recompile the original PIM components to PSM components that are supported by the new hardware platform.

In our approach every component is also a well-defined fault-containment region at the level of conceptualization and at the level of implementation. Any fault in a component manifests itself as an erroneous message. A message can be erroneous either in the domain of time or in the domain of values. If the messages are time-triggered, then errors in the time domain can be detected at the component boundaries by architectural error-detection mechanisms, since the correct arrival time of each message is known a *priori.* Errors in the value domain have to be detected at the application level.

4. COMPONENT SPECIFICATION - THE PIM

Critical to the proposed design methodology is the precise specification of component behavior at the component interfaces. In this Section we first elaborate in detail on the component characteristics, before discussing the operational and meta-level specification of the component interfaces.

4.1 Component Characteristics

As mentioned before, from a behavioral point of view a *component* is considered to be a hardware/software unit that accepts input messages, provides a useful service, maintains internal state, and produces after some *elapsed physical time* output messages containing the results. We call the interface, where the service of a component is oered to the rest of the cluster, the *Linking Interface (LIF)* of the component [Kopetz and Suri 2003a]. In order to support the *simplification strategy of partitioning*

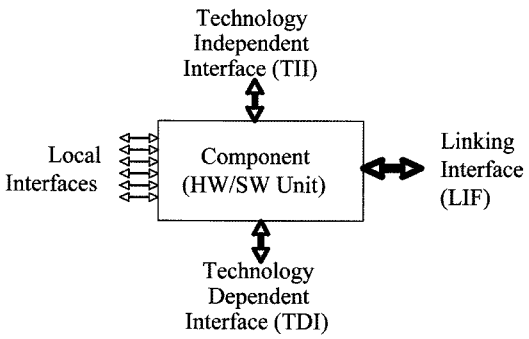

Figure 2. Component Interfaces.

[Kopetz 2008], we introduce in addition to the LIF, three further interfaces of a component (Figure 2): the *Technology Independent Interface (TII)*, the *Technology Dependent Interface (TDI)* and none, one or more *Local Interfaces.* Ifa component contains no local interface we call it a *closed component.* A component with a local interface is called an *open component.* The distinction between open and closed components is of fundamental importance, as outlined in Section 4.4.

We introduce these four dierent interfaces because each one of them serves a dierent purpose and has to be understood by a dierent clientele:

The LIF is the only interface that is of relevance for the component integration into a cluster. Its specification must contain all information that the integrator needs to be aware of and should not contain any further information.

The *Technology Independent Interface (TII)* of a component is used to configure a component for the concrete application, to control the execution speed of a component in order to optimize its power consumption, and to set the internal state of a component to the proper value at component start and at the next restart instant in case the internal state of the component has been corrupted (see also Fig. 3 in Section 6). For example, the assignment of port addresses in order to integrate a component into a given environment is done via the TII. The TII is agnostic about the component implementation.

The *Technology Dependent Interface (TDI)* interface of a component is used to provide a maintenance technician a view into the internals of the concrete component implementation. It is used for debugging a component and is thus implementation dependent. The TDI is of no relevance to the user of a component who is only interested in the component services at the LIF.

The *Local Interfaces* of an open component are used to interface a component to its cluster-external environment, e.g., to the man-machine interface, or to other clusters. *The semantics of these local interfaces must be part of the semantic LIF specification, as far as they are of relevance to the user of the component.*

In our example of the braking system (Fig. 1), the detailed syntactic structure and semantics of the local interface of the brake-pedal-position sensor to the brake-pedal sensor component is of no relevance at the system level, since the semantics of this local interface is covered by the LIF specification of the brake-pedal-sensor component.

The LIF specification of a component is required to be -*as far as theoretically possible* – self-contained. It must provide a complete description of the component behavior as seen from the viewpoint of a LIF user. The LIF specification is the mediator between a service supplier and the service user of a component. The LIF specification must describe the component behavior in the value domain and in the temporal domain and *should* be *complete* and *minimal* in the sense that it contains all information required to understand and use the services of a component that are oered at the particular LIF and nothing more. The LIF specification should be *agnostic* about the component implementationit must be the same for the PIM component and for the corresponding PSM component. From the point of view of a user of a component it should not be distinguishable, whether the components functionality is implemented by software that is executed by a CPU, by an FPGA fabric, where the interconnections of the logic elements are controlled by software, or

by a hardware ASCI (application specific IC) where the logic is part of the hardware. We dierentiate between the *operational LIF specification and the semantic LIF specification.*

4.2 Operational LIF Specification

The *operational LIF specification* covers the syntactic specification of the data items contained in the input and output messages of the component, and the temporal specification of the message send and receive instants as well as the syntactic structure of the component state as seen from this interface at the *restart instant* (see Section 6). Since the operational LIF specifications are instrumental for the *interoperability* of components i.e., the exchange of information among components, they must be precise and formal.

The temporal specification of the messages will be dierent for time-triggered and event-triggered messages. For a time-triggered message, the configuration of the *cycle* associated with a TT message provides for a precise temporal specification of the time-triggered message. For an event-triggered system, the message queues associated with the sender and receiver of the message must be specified (see also Section 5.1).

In the value domain, the operational specification of the messages structures the incoming and outgoing bit streams contained in the messages by establishing *named data items.* The syntactic specification of these named data items is a well-understood topic. Standardized interface definition languages, such as the Interface Definition Language (IDL) of the Object Management Group (OMG), are in use for the syntactic specification of data items exchanged across LIFs. The names of the syntactic data items form the link between the operational level and the semantic level of the LIF specification.

A *time-triggered component*, where the control signal to start a component activity is derived from the progression of the global time, has a higher degree of autonomy than an *event-triggered component*, where the control signal to start a component internal action is derived from the arrival of a message that originates from the outside of the component. The reasoning about the temporal behavior of a time-triggered component does not have to *step beyond the component interface* and is thus self-contained. The strict temporal order established by a time-triggered operation limits concurrency (which the human mind is ill-equipped to handle) and establishes the basis for sequential step-wise reasoning, thus supporting the simplification strategy of *temporal segmentation* [Kopetz 2008]. It follows that it is inherently simpler to reason about and understand the temporal behavior of a time-triggered component than to understand the behavior of an event-triggered component.

4.3 Protocol Abstraction

In many scenarios it useful to introduce the *protocol abstraction*: the grouping of a number of related messages (e.g., *request* and *response* messages on dierent channels) into a single higher level construct, a message protocol [Salloum 2007]. For example, the TCP/IP protocol that groups together two channels (one request, one response)

to perform a higher level error-controlled data exchange service is an example for such a higher-level protocol. A protocol abstraction can be compared to a MACRO in a programming language, where the MACRO stands for a higher-level concept of some composite communication actions that can be decomposed into its elementary parts if desired. Such a decomposition of a higher-level protocol into its constituent messages is required if we need to observe the message traffic originating form a LIF for the purpose of component diagnostics.

4.4 Semantic LIF Specification

The semantic LIF specification assigns meaning to the data items, to the *names*, introduced in the operational specification. It thus bridges the gap between the structured data items formed at the syntactic level and the users mental model of the services provided at the LIF. Central to this semantic specification is the LIF service model.

A user of a component employs the component with the intent to achieve a goal, i.e. to contribute to the solution of her/his problem. The relationship between *user intent* and the *services provided* at the LIF must be exposed in the LIF service model. Concepts that are familiar to a prototypical user must thus be the basic elements of the LIF service model. For example, if a user is expected to have an engineering background, terms and notations that are *common knowledge* in the chosen engineering discipline should be utilized in presenting the corresponding service model.

The LIF service model of a component diers from the model describing the algorithms implemented within a component. The LIF service model is *goal oriented*, while the algorithmic model is *process oriented*. A goal-oriented model specifies the intended goal state, while algorithmic model specifies the actions that must be taken in order to reach this intended goal state.

Furthermore, the LIF service model of an open component goes beyond the component boundaries. It must include all relevant properties of the components environment that are connected to the local interfaces of the component. It is thus not possible to specify the semantics of the LIF service model of an open component without knowing the *precise context of use* of the open component [Kopetz and Suri

Table I. Comparison of the Algorithmic Model and the LIF Service Model of an Open Component

	Algorithmic Model	LIF Service Model
Orientation	Process	Goal state
Properties of the controlled environment	Not included	Included
Scope	Component internal	Internal plus external
Full operational Specification of the LIF of an isolated open component	possible	possible
Full semantic Specification of the LIF of an isolated open component	possible	Not possible

2003b]. It is however possible to specify the algorithmic model of an isolated open component.

Take the example of the brake control system of Figure 1. There is one closed component, the brake control that contains no local interface. The LIF of this component can be fully specified in isolation at the syntactic level and at the semantic level, since all inputs and outputs are part of the LIF. However for the open components at the wheels, the relation between the brake forces and the wheel speeds depends on a number of parameters that are outside the wheel components: the weight of the car, the condition of the road surface, and the inclination of the road, etc.

In many situations the formal specification of a comprehensive semantic LIF model of an open component will not be possible. However, the operational specification of an open component must be formal and complete to ensure the interoperability of the components. For a more detailed discussion about the LIF specification see [Kopetz and Suri 2003a].

5. MESSAGE COMMUNICATION

In the time-triggered architecture, the basic communication mechanism among components is the exchange of *deterministic multicast unidirectional messages.* The message communication among the components makes the interaction of a component with its environment explicit and eliminates hidden dependencies among components. Higher-level inter-process communication mechanisms, e.g., shared memory, can be built on top of this basic message exchange mechanism.

Determinism in the communication is introduced for the following reasons:

(1) *Timeliness*: Many embedded systems require timely responses. The general notion of determinism [Kopetz 2008] subsumes predictable timing.
(2) *Complexity reduction*: it is much easier to reason about the behavior of a communication system, if the message transport is deterministic, i.e. it is exactly known at what instant a message will arrive, than if the behavior of the communication system is probabilistic [Kopetz 2008].
(3) *Testing*: The testability of a system is improved, if the system will produce the same outputs given it has been oered identical inputs [Schütz 1993].
(4) *Active Redundancy*: The implementation of active redundancy requires a deterministic behavior of the replicated components.

Multicast communication is supported as a basic communication property in order to enable the observation of the behavior of a component by an independent external observera monitorwithout introducing the *probe eect* [Schütz 1993].

Only *unidirectionality* can be implemented in a communication system as an *elementary service. Bi-rectionality* is a *composite service* that includes, in addition to the behavior of the communication system, the behavior of the receiving components. Unidirectionaly thus supports the strict separation of communication from computation and supports the *simplification strategy of partitioning.*

A component can have one or more interfaces, where each interface contains one or

more ports. Each port is used for the sequential sending or receiving of a single message at an instant. Depending on the control schema, we distinguish between three types of messages (and the corresponding ports): *event-triggered messages, time-triggered messages* and *data-streams.*

5.1 Event-triggered Message

The event-triggered control schema is the usual control schema associated with the message concept. An *event-triggered message* is sent whenever a significant event occurs at the sender. Event-triggered message must conform with the *exactly-once semantics*, i.e., every message produced by a sender must be eventually consumed exactly once by its receiver(s). In case the communication channel is not free at the instant of event-occurrence, the event-triggered message must be stored in a queue at the senders site before it can be transmitted by the network. Similarly, an event-triggered message that is delivered from the network to the destined receiver must be stored in a message queue before the receiver in case the receiver is not ready to accept the message at the moment of message arrival. There are thus (at least) two queues associated with every event-triggered message, one at the sender and one at the receiver. The sizing of these queues depends on the uniformity and rate of message production at the sender, the uniformity and rate of message consumption at the receiver and the available capacity of the communication channel. For example, if we have a large-bandwidth channel, the queue at the sender will be small and the queue at the receiver will be large. If we have a small bandwidth channel, it may be the other way around.

We distinguish between two types of event triggered messages, depending on the bandwidth allocation to the communication channel. If a fixed (static) bandwidth is assigned to every event-triggered channel, we call the communication channel *predictable.* In a system with predictable event-triggered communication channels, the sizing of the two queues can be performed in the local context of sender and receiver. An example of a communication system with a predictable channel is a time-triggered channel in TT Ethernet [Kopetz and et al. 2005] that is deployed for transmission of event-triggered messages.

If the bandwidth assigned to a sender is dynamic, depending on a the activity of other senders that use the same communication channel, then we call the communication channel *best effort.* In a system with a best-effort channel, the seizing of the two queues can be performed only in the global context of all users of the channel. Examples for a communication system with best-effort channels are standard Ethernet or the CAN [Press 1990] protocol, that is widely used in the automotive environment.

Since it is diffcult to assure that a best-effort unidirectional channel will deliver a message within a given time-interval, a higher-level protocol that binds two event-triggered channels together is commonly formed to be able to inform the sender of the successful receipt of the message and realize a time-constrained error-detection service in case a message does not arrive within the given protocol-specific time interval. If an acknowledgement message is not received with this specified time window, the original message is resent (PAR Protocolpositive acknowledgement or

retransmission).

It is difficult to give precise temporal guarantees to event-triggered messages, since the delay of the event-triggered messages in the sender queue and the receiver queue is difficult to quantify, even if the communication system is predictable.

5.2 Time-triggered Message

A cycle is assigned to every time-triggered message. The *time-triggered message* is sent whenever the periodic cycle start that has been assigned to this message occurs. The cycles assigned to the time-triggered messages must be planned a priori by a *message scheduler*, in order to avoid any message conflicts among time-triggered messages. At the instant of *cycle-start* of a time-triggered message, the contents of the message buer at the senders site are fetched by the communication system and transmitted to the receivers (non-consuming read) within a know interval. At the instant of message delivery by the communication system, the content of the message buer at the receivers site is overwritten by the arriving time-triggered message. There are no queues associated with time-triggered messages. Since the time-triggered communication system is free of conflict and deterministic it is possible to associate temporal guarantees with time-triggered messages. Time-triggered messages are well suited to transmit data items with state semantics in periodic control systems.

From a conceptual point of view, communication by time-triggered messages is easy to comprehend because a time-triggered message provides a powerful abstraction of the components environment – a *temporal firewall* to the component environment [Kopetz and Nossal 1997]. At the receivers side the message buer of a (periodic) time-triggered message always contains an image [Kopetz and Kim 1990] of the most-up-to-date value of a remote state variable. This value can be accessed locally just like the value of any other local variable. The message buer of a time-triggered message provides the only interface to the external world and eliminates control-error propagation from the external environment into the component by design. The cycle of the time-triggered message determines the worst-case temporal validity of the accessed value.

In a time-triggered system, the detection of a lost or corrupted message can be performed by the receiver on the basis of the a priori knowledge about the expected arrival time of the periodic time-triggered messages. Examples for time-triggered protocols are the TTP, FlexRay [R. Mores et al. 2001] or time-triggered (TT) Ethernet [Kopetz and et al. 2005].

In our brake example of Figure 1, the freshest values of the speed sensors are delivered periodically to the brake-control component by a time-triggered communication system.

5.3 Data Stream

A *data stream* is a regular sequence of timed messages that is produced by a sender and consumed by a receiver. Data streams are important in multimedia systems. The temporal interval between any two messages of the data stream is known a priori. It is thus possible to perform *on-the-fly* processing of data streams in this

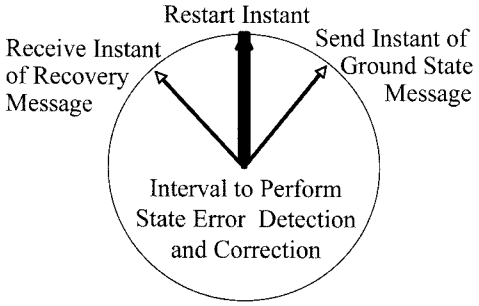

Figure 3. Timing of the Component Restart.

temporal interval. Data streams have the temporal properties (*precise instant of transmission*) of time-triggered messages and value properties (*exactlyonce-semantics*) of event-triggered messages. In a time-triggered architecture it is possible to synchronize data-streams from dierent sources which makes this architecture well-suited for multi-media applications (e.g., lip synchronization by synchronizing an audio stream with a video stream).

Since the production rate and consumption rate of multimedia messages is often content dependent, it is common to associate *watermarks* with a data stream to monitor the length of the sender and receiver queues. A higher-level protocol sends these watermarks from the receiver to the sender in order to inform the partners about the fill-level of the queues and alert the producer of messages about the need to adapt the production rate of messages.

The precise synchronization of the producer and consumer of multimedia messages reduces the need for intermediate buers and saves storage and energy, which is of special importance in portable devices.

5.4 Virtual Networks

Given the basic communication primitives introduced above, it is possible to implement any virtual network at a higher level [Obermaisser and Peti 2005]. For example, it is possible to give to the application programmer the interface of a given legacy communication system, such as the CAN (Control area network) with temporal properties that are in agreement with the CAN specification [Obermaisser 2006].

6. COMPONENT RESTART AND EVOLUTION

The era of nano-scale devices will bring an increase in the soft error-rate of micro-electronic devices [Calhoun and et al. 2008]. It is therefore expedient to provide systematic means to handle transient device failures at the architectural level in order to increase the robustness of an embedded system. In the proposed component-based architecture, every component forms a fault-containment region. A transient fault, e.g., one caused in the hardware by ambient cosmic radiation or by a Heisenbug [Gray 1986] in the software, can result in a corruption of the internal state of the

component and possibly in an erroneous output message. If the failure is detected and the corrupted state can be corrected quickly, the component can be brought back into full service and the transient fault has been tolerated.

In order to enable the swift recovery of a component in case its internal state has been corrupted by a transient fault, it is necessary to plan during the design for recovery instants, i.e., *periodic instants* where the internal state of a component is well-defined and small. We call such a small state where no process is active and all communication channels are empty a *ground state* [Kopetz 1997] and the corresponding (periodic) instant the *restart instant.* Ina *state-aware design*, the precise specification of the ground state at the restart instants must be part of the design. We propose that the ground-state of a component is periodically published in a ground state message immediately after the restart instant in order that the health of the ground state can be monitored by an external monitoring component (see Fig. 3). This external monitoring component must be part of an independent fault containment region, such that a failure of the component-to-be-monitored and the monitor are not correlated. The monitor component can check the existence and correctness of the ground state message and, in case an error is detected, calculate a proper restart state that will be relevant at the next restart instant (Fig. 3 which depicts a cyclic representation of time). Immediately before this next restart instant, the monitor component will send a recovery message that contains the relevant restart state to the TII interface of the erroneous component. The arrival of this recovery message at the TII interface will cause a component reset and a restart of the component with the provided restart state at the next restart instant.

In the example of the brake system of Figure 1, the monitor component will periodically receive the ground state of all other components. In case any one of these ground state messages is missing or the ground state is erroneous, the monitor component will reset and restart the failed component such that the component service will be reestablished after the outage of at most two restart periods.

Successful long-lived systems evolve continuously. Over time the system is expanded, new components, sensors and actuators are added and the functionality of existing components is modified. As long as the *operational message-based interface specification* of a component remains unchanged, modifications in the functionality of a component or in its local interfaces do not impact the operational structure of the rest of the system. Since the basic communication system among components is unidirectional and multicast, new information-consuming components can be added and the external behavior of components can be observed without disturbing the operation of the existing legacy system. For example, an existing component can be expanded to form a new cluster, without changing the component interface to the existing legacy cluster.

7. COMPONENT IMPLEMENTATION-THE PSM

The design methodology, which has been outlined in the previous Sections, requires proper services at the architectural level. In the context of the EU IST project GENESYS (Generic Embedded System Architecture) we work towards a generic embedded system architecture that will provide the needed architectural services,

such as clock synchronization, ground state monitoring, multicast unidirectional message transmission etc.. GENESYS introduces three levels for the integration of components, the *chip level*, the *device level* and the *closed or open system level*. At the open system level, devices normally communicate via wireless channels and can enter and leave an ensemble dynamically.

7.1 Chip Level: A Component as an IP-Core

A component can be implemented in the form an IP-Core of an heterogeneous Multiprocessor-System-on-Chip (MPSoC). Since a component is a self-contained functional unit that communicates with its environment solely by the exchange of messages, the corresponding IP-core must contain all resources that are needed to provide such an autonomous function. A processor-based IP-core consists of a CPU, local memory for instructions and data, the message interfaces as well as the system and application software to realize the specified functionality. An FPGA based IP-core consists of the hardware gates and the software for the interconnection logic. In an ASIC implementation, the complete control logic is implemented in hardware. Although in a time-triggered system all three implementations provide the same functional service with the same external timing, they can have very dierent meta-functions characteristics, such as energy efficiency or silicon real-estate requirements.

7.2 The Time-Triggered NoC

If the components of an applications are mapped into IP-cores of an MPSoC, then the communication network becomes a time-triggered Network-on-Chip (TT-NoC). In order to maintain the architectural property of independent communication channels, this NoC must avoid any dynamic global resource sharing. In an NoC the distances between the nodes are short and the communication channel can be highly parallel. It is thus possible to build networks with a very high bandwidth (hundreds of Gigabits) without undue eort. In a TT-NoC we can thus aord to assign a dedicated time-triggered slot to every single communication link. This gives us a deterministic time-triggered network free of any hidden interferences among communicating partners.

The TT-NoC establishes the global time among the components. However, the components themselves form *islands-of-synchronicity*, where the clock-rate of each component can be controlled via its TII interface in order to optimize its power consumption. The clock domain-crossing between a component and the TT-NoC takes place in the network interface of the component. The architecture thus provides a *global time without the need of a global clock.*

7.3 Device Level: Time-Triggered Ethernet

Inter-chip and inter-device communication are more expensive than intra-chip communication via an NoC. Here we require a communication infrastructure that is sensitive to the cabling costs and thus shifts the tradeo between channel cost and network control into the direction of a more elaborate network control. The time-triggered Ethernet protocol, which is fully compatible with the Ethernet standard

and provides in addition to the standard event-triggered best-eort Ethernet service a temporally predictable time-triggered service satisfies our requirement for a cost-eective inter-device communication system.

7.4 Safety Concerns -Component Allocation

If the braking application of Figure 1 were a non-safety critical application, we could integrate all seven components of Figure 1 as IP cores on a single MPSoC. However, in a safety-critical application, such as a braking system, such an integration of all components on a single die is not feasible, because in a safety case it is commonly assumed that any chip can fail in an arbitrary failure mode with a probability of 106/ hour, while the overall system reliability has to be better than 10-9 failures/hour. In such a safety-critical application it is thus necessary to introduce redundancy by replicating components and allocating them to dierent chips in such a way that the arbitrary failure of any one chip or of any one physical connection can be tolerated without the total loss of the system service.

Common-mode failure concerns are thus a decisive argument when allocating components to chips in a safety-critical environment. In the time-triggered architecture all IP cores on a single chip are free of any hidden design dependency. It is thus possible to integrate application subsystems of dierent criticality on the same chip, since a lower-criticality subsystem cannot interfere with a higher criticality subsystem. In an automotive control system, where the number of physical ECUs (electronic control units) has to be small, the component-to-hardware allocation will consider all distributed application subsystems together and try to find an allocation that meets all constraints (safety, wiring, space, cost, etc.) in an nearly optimal manner. To a significant extent, the physical structure of a safety-critical system will be determined by the requirements of independence of the replicated components that are introduced to meet safety concerns.

8. CONCLUSIONS

In this paper we presented a design methodology for the component-based design of robust time-triggered distributed embedded systems that are simple to understand and can be modified without difficulty. This design methodology supports the three conceptual simplification strategies of *abstraction*, *partitioning* and *segmentation*. The rigorous one-to-one mapping of software components to hardware components at all levels of the design, the strict message orientation and the precise specification of the linking interfaces of components in the domains of value and time are the characteristics for this design methodology. As a consequence of this one-to-one software/hardware mapping every component forms a well-defined fault-containment region with a pragmatic restart strategy in case a transient hardware fault or a Heisenbug in the software corrupts the internal state of the component. The key issue of this design methodology is the specification of the message-based linking interfaces of a component. A distinction is made between the *operational interface specification* and the *semantic interface specification*. While the operational interface specification of all components must be precise in the domains of time and value, the

semantic interface specification of an open component can only be established if the context of use of the component is known.

ACKNOWLEDGEMENTS

This work has been supported in part by the European research project GENESYS under project Number FP7/213322 and by the Austrian research project TTSoC under project number FIT-IT 813299. The many discussions within the GENESYS project and within the research group at the TU Vienna, particularly with Roman Obermaisser, Christian ElSalloum, Bernhard Huber and Christian Paukovits are warmly acknowledged.

REFERENCES

BELADY, L. AND M. LEHMAN. 1976. A model of large program development. *IBM Systems Journal* 15(3):225–252.

CALHOUN, B. AND ET AL. 2008. Digital circuit design challenges and opportunities in the era of nanoscale cmos. *Proceedings IEEE*:343–365.

GRAY, J. 1986. Why do computers stop and what can be done about it? In *Proceedings of the 5th Symposium on Reliablity in Distributed Software and Database Systems.*

HUBER, B., R. OBERMAISSER, AND P. PETI. 2006. MDA-Based Development in the DECOS Integrated Architecture-Modeling the Hardware Platform. In *Proceedings of the 9th IEEE International Symposium on Object and component-oriented Real-time distributed Computing (ISORC'06).*

IEEE. 2002. 1588 standard for a precision clock synchronization protocol for network measurement and control systems. Technical Report.

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

KOPETZ, H. 2008. The complexity challenge in embedded system design. In *Proceedings of the 11th IEEE International Symposium on Object-oriented Real-time distributed Computing.*

KOPETZ, H. AND G. BAUER. 2003. The Time-Triggered Architecture. *IEEE Special Issue on Modeling and Design of Embedded Software*:112–126.

KOPETZ, H. AND ET AL. 2005. The design of tt ethernet. In *Proceedings of the 8th IEEE Int. Symposium on Object-oriented Real-time distributed Computing.* IEEE Press, Seattle, USA.

KOPETZ, H. AND K. KIM. 1990. Temporal uncertainties in interactions among real-time objects. In *Proceedings of Ninth Symposium on Reliable Distributed Systems*:165–174.

KOPETZ, H. AND R. NOSSAL. 1997. Temporal firewalls in large distributed realtime systems. In *Proceedings of IEEE Workshop on Future Trends in Distributed Computing.* IEEE Press, Tunis, Tunesia.

KOPETZ, H. AND N. SURI. 2003a. Compositional design of RT systems: A conceptual basis for specification of linking interfaces. In *Proceedings of the 6th IEEE Int. Symposium on Object-Oriented Real-Time Distributed Computing*:51–60.

KOPETZ, H. AND N. SURI. 2003b. On the limits of the precise specification of component interfaces. In *Proceedings of the Ninth IEEE Int. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003)*:26–27.

OBERMAISSER, R. 2006. Reuse of CAN-based legacy applications in time-triggered architectures. *IEEE Trans. on Industrial Informatics* 2(4):255–268.

OBERMAISSER, R. AND P. PETI. 2005. Realization of virtual networks in the DECOS integrated architecture. In *Proceedings of the Workshop on Parallel and Distributed Real-Time Systems 2006 (WPDRTS).* IEEE.

OMG. 2001. Model-driven architecture (MDA). Technical Report. rep., Object Management Group (OMG).

PRESS, S. 1990. Controller Area Network CAN, an in-vehicle serial communication protocol. *SAE Handbook, SAE Press*, 20:341–20.355.

MORES, R., G. HAY, R. BELSCHNER, J. BERWANGER, C. EBNER, S. FLUHRER, E. FUCHS, B. HEDENETZ, W. KUFFNER, A. KRÜGER, D. MILLINGER, M. PELLER, J. RUH, A. SCHEDL, AND M. SPRACHMANN. March 2001. FlexRay – the communication system for advanced automotive control systems. *SAE 2001 World Congress, Detroit, MI, USA Doc. No 2001-01-0676*:303–314.

REISBERG, D. 2001. Cognition, 2nd ed. *W. W. Norton Company.*

SALLOUM, C. E. 2007. Interface design in the Time-Triggered System-on-Chip Architecture. Ph.D. thesis, Institute fr Technische Informatik, Technische Universität Wien: Vienna, Vienna, Austria:142.

SCHÜTZ, W. 1993. The Testability of Distributed Real-Time Systems. *Kluwer Academic Publishers,* Boston.

Hermann Kopetz received his PhD in physics "*sub auspiciis praesidentis*" from the University of Vienna, Austria in 1968. After he accepted 1998 an appointment as a Professor for Computer Process Control at the *Technical University of West-Berlin*, moving to the Technical University of Vienna in 1992. From 1990 to 1992 Kopetz was chairman of the IEEE Technical Committee on Fault-Tolerant Computing and from 1996 to 1998 Chairman of the IFIP WG 10.4 on Dependable Computing and Fault-Tolerance. Kopetz is a full member of the Austrian Academy of Science and a member of the ISTAG. He has published a widely used textbook on *Real-Time Systems*, more than 150 papers and more than twenty patents on the topic of dependable embedded systems. Kopetz received the IEEE Computer Society 2003 Technical Achievement Award with the citation*: For outstanding contributions to the field of safety-critical real-time computing.* In 2006 Kopetz chaired the *ARTEMIS* Strategic Research Expert Group on *Reference Designs and Architecture.* In June 2007 he received the honorary degree of *Dr. honoris causa* from the University Paul Sabatier in Toulouse, France.