Software Engineering for Real-Time: A Roadmap

Hermann Kopetz
Technische Universität Wien, Austria
Email: hk@vmars.tuwien.ac.at

ABSTRACT

The next ten years will see distributed real-time computer systems replacing many mechanical and hydraulic control systems in high-dependability applications. In these applications a failure in the temporal domain can be as critical as a failure in the value domain. This paper discusses some of the technology trends that explain why distributed embedded real-time systems for high-dependability applications will move into the mainstream. It then investigates the new requirements that must be addressed by the software engineering process. Two of the most important requirements are the design for composability and the systematic validation of high-dependability distributed real-time systems. In the last two sections, these issues of composability and validation are treated in some detail.

KEYWORDS

Real-time systems, composability, distributed systems, validation, system architecture

1. INTRODUCTION

A hard real-time computer system is required to produce the intended result before a specified point of physical time, the deadline. This point of time is determined by the application the computer system is intended to service. The controlling real-time software must be designed to generate the correct behavior of the computer both in the value domain and in the temporal domain to meet this application requirement. Since the temporal behavior of the software depends on the performance of the computer hardware, software engineering for real-time systems must take into consideration the architectures and capabilities of the available computer hardware. It follows that the software design methods and architectures of real-time systems will be strongly influenced by the given hardware environment.

The impressive advances in the field of computer hardware in the recent years make it economically feasible that the old architectural proverb "form follows function" becomes the guiding principle for the design of distributed real-time computer systems of the future. In nearly all but the high volume applications it will be economically justified to partition a system along functional hardware/software boundaries in order to avoid the extra software design and validation effort needed to cohabitate unrelated functions on the same hardware units. In such an architecture the issue of composability and reusability, how to build systems constructively out of pre-validated software/hardware components that provide well-defined functions, moves to the center of interest. Already today this issue of composability is a key concern in the development of industrial control systems.

It is the objective of this paper to speculate about the future of software engineering for hard real-time over the next ten years. The paper starts by discussing the fundamental difference between soft and hard real-time systems and makes the assumption that the main challenge will be in the design and validation of hard real-time systems for ultradependable applications. Since real-time architectures are strongly dependent on the hardware developments, Section 3 deliberates about the visible trends in the field of hardware and communication. Based on these trends, Section 4 summarizes the requirements on future real-time system design and concludes that the most important requirement is the support of composability. Section 5 investigates this requirement in detail and postulates three principles of composability for distributed real-time systems. Section 6 focuses on the validation of high-dependability systems. The paper closes with a conclusion in Section 7.

2. SOFT VERSUS HARD REAL-TIME SYSTEMS

From the point of view of temporal performance, two types of real-time systems can be distinguished

(i) soft real-time systems: these are systems where a failure to meet a specified deadline reduces the utility of the result, but does not lead to a significant financial loss. An example of such a system is a letter-sorting machine: If a letter is dropped into the wrong box because of a timing failure of the computer, no serious consequences will result.

(ii) Hard real-time systems: these are systems where a failure to meet a specified deadline can lead to catastrophic consequences. An example of a hard real-time application is a computer system controlling a railway-shunting yard: If a wagon is released to the wrong track because of a timing failure of the computer, a serious accident may occur.

From these two simple examples it becomes evident, that the distinction between a soft- or hard real-time computer system does not depend on the computer system per se, but on the characteristics of the application the computer system is intended to serve.

In the not-too-distant past, the majority of the real-time computer systems have been soft real-time systems: In many of the safety critical real-time applications an additional electrical, mechanical or hydraulic backup system (without software) was included to take control (to avoid a catastrophe) in case the computer system failed. This situation has been changing slowly over the last ten years and will continue to change further over the next ten years. With the successful deployment of software controlled flight control systems in civil aircraft (Airbus A 320, Boeing 777), many other industries, e.g., the automotive industry, intend to replace the expensive mixed technology approach to system control by distributed fault-tolerant real-time computer systems without mechanical or hydraulic backup. For example, in a "brake-by-wire" car, a fault-tolerant computer [7] will replace the hydraulic connection between the brake pedal and the brakes. With the move of these safety-critical control systems without backup into the mainstream of industrial control, the software engineering community is up to new challenges: the design of software systems that are guaranteed to meet the specified deadlines of ultra-dependable systems in all operational scenarios within the stated load- and fault-hypothesis and the development of validation technologies that affirm that the software is safe to deploy.

In a hard real-time system, a failure in the temporal domain is as critical as a failure in the value domain. Many of the present-day software engineering techniques, such as object-oriented design methods, focus on the value domain and consider the temporal domain a low-level implementation issue. Temporal properties are system properties that depend on the proper cooperation of all levels of a design: hardware, communication, operating system, and application software. The design of hard real-time systems is thus fundamentally different from the design of soft real-time systems or non real-time systems [11], where the temporal properties of the lower levels of an architecture are not explicitly considered during the design process of the higher level application software. In a hard real-time environment, it cannot be assumed that timeliness can be tested into a system, timeliness must be the consequence of a rational system and software design process. Even some of the more recent software specification and design technologies that are targeted specifically at real-time applications, such as real-time UML [12] or real-time CORBA [14], are not based on a sound model of time and do not consider temporal properties as first order quantities, but rather as an addendum.

3. TECHNOLOGY TRENDS

The architecture of a real-time system is strongly influenced by the capabilities and cost/performance of the available hardware components. At present, the computer industry in general and the semiconductor industry in particular is going through a period of revolutionary qualitative change that deeply affects the environment of the system engineer designing hard real-time systems. In this section, some of these important technological trends, as they relate to real-time system design, are presented.

3.1 System on a Chip (SOC)

Today's level of VLSI technology makes it possible to produce a powerful computer node of a distributed system, including a 32 bit CPU, a megabyte of memory, I/O circuitry and a network controller on a single silicon die. Such a system-on-a-chip (SOC) offers many advantages from the point of view of functionality, dependability and cost. According to the 1997 semiconductor road map, published by the Semiconductor Industry [18] the number of transistors/cm^2 will increase from about 6 Mio to 24 Mio within the next 5 years, enabling the production of much more powerful SOCs. The development costs of an SOC component are high--they can be in the order of 10 Mio US $, while the production costs are quite low, e.g., 10 US $. These very cost-effective mass-produced SOCs are here to stay and will form the core hardware elements of future real-time control systems.

3.2 Smart MEMS Sensors

A smart device is the combination of a sensor or actuator element and a local microcontroller that contains the interface circuitry, a processing element, memory and a network controller in a single unit. More and more sensor elements are themselves microelectronic mechanical systems (MEMS) [5] that can be integrated on the same silicon die as the associated microcontroller. The smart sensor technology offers a number of advantages from the

points of view of technology, cost and complexity management:

(i) Electrically weak non-linear sensor signals that originate from an MEMS sensor can be generated, conditioned, transformed into digital form and calibrated on a single silicon die without any noise pickup from long external signal transmission lines [3].

(ii) It is possible to locally monitor the sensor operation and thus simplify the diagnostics. In some cases it is possible to build smart sensors that have a single simple external failure mode--fail-silence, i.e., the sensor operates correctly or does not operate at all.

(iii) The interface of the smart sensor to its environment can be a simple well-specified digital communication interface to a sensor bus, offering "plug-and-play" capability if the sensor contains its own documentation on silicon.

(iv) The internal complexity of the smart-sensor hardware and software and the internal sensor failure modes can be hidden from the user by a well-designed fully specified smart sensor interface that provides just those services that the user is interested in. Thus, the smart sensor technology can contribute to a reduction of the complexity at the system level.

The same arguments of cost reduction in volume applications apply to smart sensors as they apply to other SOCs.

3.3 COTS Components

The above mentioned trends exert an enormous economic pressure on all but the large-volume applications to use Commercial-Off-The-Shelf (COTS) hardware and software components when designing new computer systems. Even large customers that traditionally have designed their own special components, like the US DOD, are realizing that cost-effective state-of-the-art system development can only be achieved if COTS components are used.

We see three different types of COTS components in the real-time system market:

(i) **Hardware components** as we have them today. In the future, the hardware components will provide more generic system services for the design of distributed real-time systems, e.g., distributed clock synchronization.

(ii) **Software components**, such as RT operating systems, that can be used in many different applications. One problem with the use of COTS software components in real-time applications is that a software component per se does not have any temporal properties--the temporal behavior only emerges if the software is bound to a particular hardware component. It follows that the user has to investigate the temporal performance and the fault-models [4] of a COTS software component on the selected hardware platform. It is thus a myth that COTS software components can be integrated in high-dependability real-time applications with little effort.

(iii) **A hardware/software component**, such as a smart sensor that includes the signal conditioning, calibration, diagnostic, and network control software and provides an established output signal across a stable standardized Communication Network Interface (CNI). The constructive integration of COTS components into larger systems requires precise and stable CNI specifications, both in the value domain and in the temporal domain. Such precise CNI specifications are a prerequisite for a thorough test of the components and for the determination of the preconditions for the robust operation of the components in the new system context. The present integration technologies, e.g., RT-CORBA [14], do not address these issues of temporal interface specification with the required rigor. As soon as these interface problems are solved, we see a bright future for the hardware/software components in the real-time market.

3.4 INTERNET Connectivity

The connection of a control system to the INTERNET can bring about a number of advantages, such as access to WEB sites, remote monitoring of the controlled object, remote diagnosis of failures, etc. However, there are two major problems with the present INTERNET technology in hard real-time applications: lacking security and unpredictable temporal performance. While it can be expected that the security problem will be solved in the near future because of the enormous interest in electronic commerce, the lacking temporal predictability is here to stay for a longer time. The inherently bursty traffic pattern that is intrinsic to most INTERNET applications makes it difficult to guarantee dependable real-time performance with minimal jitter over the INTERNET.

3.5 High-Dependability Systems

There is a visible trend to high-dependability applications in the real-time control market. These high-dependability applications require fault-tolerant computer systems, because the application service must continue even if parts of the control system have failed. Fault-tolerant computer systems will play an important role in the future market for the following reasons:

(i) The successful use of high-dependability computer systems in critical applications, such as flight-control systems, has opened large new markets for embedded computer systems in vehicles that up to now have relied on mechanical or hydraulic control systems, e.g., braking and steering.

(ii) The loss-of-service caused by a single failure of a control system, e.g., on an assembly line, even if not

safety critical, is often more significant than the cost of replicating critical hardware elements of the control system hardware in order to provide a fault-tolerant control system that tolerates hardware failures.

(iii) The smaller geometry of the feature elements on future VLSI circuits (below .1 micron) increases the probability of a transient fault (e.g., a high energy particle) to cause a bit-flip, i.e., a soft error of the device. A fault-tolerant architecture masks these soft errors at the system level.

(iv) In a fault-tolerant system the expensive "on-call" maintenance can be replaced by the less expensive regular preventive maintenance. The cost savings associated with such a change in the maintenance strategy can be larger than the additional cost for providing a fault-tolerant system.

4. WHAT IS REQUIRED?

When analyzing the above mentioned technology trends it can be concluded that the basic structure of future real-time control systems will be that of a distributed real-time system, consisting of a set of node computers connected by a communication system. Such a distributed real-time system will comprise two types of nodes, the powerful system nodes, the SOC computers with the associated peripheral devices, and the smart sensor nodes. The nodes will communicate via real-time networks. All nodes that are connected to a real-time bus form a cluster. Gateway nodes realize the connection between clusters.

4.1 Two-Level Design Methodology

The future software engineer will clearly distinguish between two types of activities: the design of architecture and the development of the components [8]. On the architecture level the interactions among the components must be specified and the communication network interfaces (CNIs) to the components must be defined and frozen, both in the value domain and in the temporal domain. At the component level, the development of the components, i.e., hardware software units, that provide a defined service across the CNIs must be accomplished, taking the architecture- level CNI definitions as constraints. To support such a development process, a two-level design methodology is needed.

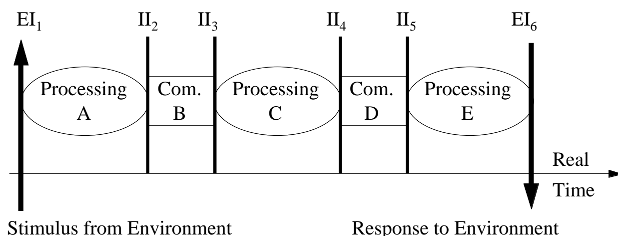

Figure 1: Real-Time Transaction

Consider the example of Figure 1, i.e., a distributed real-time transaction between a stimulus (external interface EI_1), visit to three components (processing component A, C, and E) and a response to the environment (external interface EI_6). This RT transaction can be decomposed into three processing actions (at the components A, C, and E) and two communication actions (B and D). If the temporal properties of the intermediate interfaces (II_2, II_3, II_4, and II_5) are frozen during the architecture design, then it is possible to develop the components A, C, and E independently and to compose the temporal properties of the transaction out of the temporal properties of the subsystems. If the temporal properties of the intermediate interfaces are not specified during architecture design, then composability with respect to timeliness is not supported.

In a "top-down" design process the lay-out of the components must take the system-level specifications of the CNIs as constraints for the implementation. In a "bottom-up" design process the CNI interfaces of the existing components provide the constraints for the architecture level design. This clean separation between architecture-level design and component-level design, often performed in different organizations, requires a technology for the precise specification of the CNIs in the value domain and in the temporal domain.

4.2 Predictable Communication

"The network is the only mechanism suitable to enforce and manage real-time operation of distributed systems" [2]. If the temporal properties of a network are not stable, i.e., if the points in time of the acceptance of a message by the network at the sender's CNI and the points in time of the delivery of the message by the network at the receiver's CNI are not known *a priori*, then it is not possible to precisely coordinate the activities of the nodes in the temporal domain. Any unknown jitter caused by the network results in a degradation of the quality of service of control loops that are closed via the network.

It is much easier to reason about the correct operation of a distributed real-time system if all nodes have access to a global time [8] of known precision than if they have not. The construction of such a global time is in the responsibility of the network.

According to our view, the following two different real-time network types are needed in distributed control applications for economic reasons (from the technical point of view a single system network type would be sufficient):

(i) **System Network**: The system network connects the system nodes, the powerful SOCs. System networks must have distributed communication control and provide fault-tolerance such that the loss of any node or a communication channel will not cause the failure of the system network.

(ii) **Sensor Network**: The low-cost sensor network connects one or more system nodes to the smart sensors and actuators. Since the next generation of

low-cost sensor nodes will have imprecise on-chip oscillators, which require a periodic synchronization from a master with a good crystal oscillator, the sensor networks of the future will be multi-master networks. Fault tolerance will be achieved by replicating sensor nodes and replicating the sensor network.

We assume that the complexity (and cost) of a network controller in a smart-sensor node of the sensor network and the network controller in a node in the system network will differ by an order of magnitude. However, both networks must provide deterministic communication and a global time of known precision to all nodes.

4.3 Generic Fault Tolerance

At present, many fault-tolerant real-time systems are application specific, requiring a significant amount of additional application software for the implementation of the fault-tolerance functions. In the future we see a need for architectures that provide generic services for fault tolerance, such as fault-tolerant clock synchronization, or a membership service [8] at the hardware or system software level. Ideally, the application software for a fault-tolerant system and a non-fault-tolerant system will be the same.

5. COMPOSABILITY

In classic engineering disciplines a component is a self-contained subsystem that can be used as a building block in the design of a larger system. The components provide the specified service to its environment across the well-specified component interfaces. An example of such a component is an engine in an automobile, or the heating furnace in a home. The component can have a complex internal structure that is neither visible, nor of concern, to the user of the component.

5.1 What is an Ideal Component?

Ideally, the larger system should be constructed from nearly autonomous components that can be integrated without violating the principle of composability: that properties that have been established at the component level will also hold at the system level. Examples of such properties in the context of distributed real-time systems are timeliness and testability. An ideal component should be an autonomous unit that maintains its encapsulation when viewed from a number of different vantage points [9].

A unit of service provision: Most importantly, a component must be unit of service provision. The service is offered to the component environment across a real-time service interface. In a distributed real-time system, the service consists of the timely processing and provision of the requested information. Since the validity of real-time information is time dependent, the specification of the precise point in time when the information must be present at the component interface is of relevance. From the point of view of the service user, the internal structure of the component is irrelevant, as long as the specified information is provided at the anticipated points in time at the component interfaces.

A unit for validation: It must be possible to validate the proper operation of a component in the value domain and in the temporal domain in isolation. The preconditions for the correct operation of the component, both in the domains of time and value that must be satisfied by the component environment at the component/environment boundary must be precisely specified in the interface specifications. The specification of these preconditions is the input for the construction of an environment simulator that implements the testbed for the component validation. The post conditions that must be satisfied by a correctly operating component form the basis for the acceptance test of the component.

A unit of error containment: All errors that occur inside a component must be detected before the consequences of these errors propagate across a component interface. Otherwise, a defective component can falsify the operation of other components by the provision of corrupted output data across the component interface. Ideally, a component should support the fail-silence property: it either operates correctly (in the domains of time and value), is silent, or it produces a detectably incorrect result without disturbing the other components in the system. If a component has no other than such clean external failure modes, fault-tolerance can be achieved by replicating replica-deterministic components [15].

A unit for reuse: A component should be a unit for reuse. This requires that the component has standardized interfaces with some flexibility to support the integration of the component in diverse system contexts. For example, the component interface should support name translation to decouple the internal name space of the component from the external name space of the environment. This interface flexibility should not require a revalidation of the previously established component properties, such as correct operation or timeliness.

A unit of design and maintenance: Finally, a component should be a unit of design and maintenance. It is well known that system structures evolve along organization structures. If the work output of an organizational group is a nearly autonomous subsystem with well-specified interfaces, then the management of this group is simplified. The error containment boundaries around a component reduce the possibility of unforeseen consequences of software maintenance actions.

Considering the proposed properties of an ideal component, a hardware/software unit, a node as outlined in Section 3.3, seems to be the best choice for a component in a distributed real-time system. Such a hardware/software unit is a self-contained SOC-computer with its own processor, memory, communication interface, interface to the controlled object, operating system and real-time application programs. This hardware/software unit

performs a coherent function within the distributed computer system.

Figure 2 shows an example of a possible future integrated control system onboard a car [11] with seven components. In this figure, the node replicas that are introduced for the purpose of fault-tolerance are not depicted, nor is the replicated system bus shown. Each component consists of two parts, the communication controller (CC) and the host computer that executes the application software. Between these two parts lies the communication network interface (CNI).

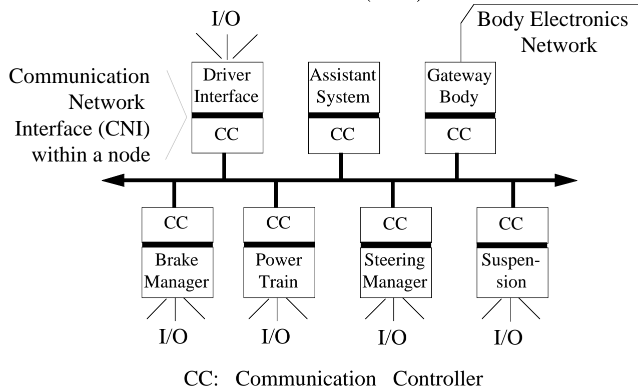

Figure 2: Example of a distributed vehicle control system consisting of 7 nodes.

The grand challenge lies in the development of architectures and software design methods for distributed real-time systems that support *composability*, that large real-time systems can be built constructively out of available components. A component property is said to be *composable* if the system integration will not invalidate this property once it has been established at the component level. Examples of such properties are timeliness or testability.

From the point of view of the analysis of a composable architecture, it is reasonable to distinguish between the following two service classes of an integrated distributed control system:

(i) **Prior Services**: Given a component that has been developed independently to provide a specified service, e.g., the control of an engine by an engine control system. This service has been validated at the component level and is thus available prior to the integration of the component into an integrated control system. We call such a service a *prior service*.

(ii) **Emerging Services**: The integration of components into a system generates new services that are more than the sum of the prior services. Take the example of a car: The integration of the four wheels, the chassis and the engine, all individual components, give rise to a new level of service, the transport service, that was not present at the component level. We call these additional services that come about by the integration the *emerging services*.

In a distributed real-time computer system, the emerging services are the result of information exchanges among the interacting nodes across the component interfaces. Therefore, the communication system plays a central role in determining the composability of distributed computer architecture.

5.2 Component Interfaces

From the point of view of complexity management and composability, it is useful to distinguish between three different types of interfaces of a component: the real-time-service (RS) interface, the diagnostic and management (DM) interface, and the configuration planning (CP) interface. For the composability, the most important interface is the RS interface.

The Real-Time-Service (RS) Interface: The RS interface provides the timely real-time services to the component environment during the operation of the system. In real-time systems it is a time-sensitive interface that must meet the temporal specification of the architecture in all specified load and fault scenarios. The composability of architecture depends on the proper support of the specified RS interface properties (in the value and in the temporal domain) during the operation. From the point of a user, the internals of the component are not visible, since they are hidden behind the RS interface.

The Diagnostic and Management (DM) Interface: The DM interface opens a communication channel to the internals of a component. It is used for setting component parameters and for retrieving information about the internals of the component, e.g., for the purpose of internal fault diagnosis. The maintenance engineer that accesses the component internals via the DM interface must have detailed knowledge about the internal structure and behavior of the component. The DM interface is not contributing to the composability. Normally, the DM interface is not time-critical.

The Configuration Planning (CP) Interface: The CP interface is used to connect a component to other components of a system. It is used during the integration phase to generate the "glue" between the nearly autonomous components. The use of the CP interface does not require detailed knowledge about the internal operation of a component. The CP interface is not time critical.

5.3 The Principles of Composability

In a distributed system the components interact via a communication system as shown in Figure 2 to provide the emergent services. These emerging services depend on the timely provision of the real-time information at the RS interfaces of the components. The RS-interface is the only interface that is relevant from the point of view of composability. For an architecture to be composable, it must adhere to the following three principles with respect to the RS-interfaces:

(i) Independent development of components.

(ii) Stability of prior services.

(iii) Constructive integration of the components to generate the emerging services.

These principles are discussed in detail in the following sections.

Independent Development of Components: A composable architecture must distinguish sharply between architecture design and component design. Principle one of a composable architecture is concerned with design at the architecture level. Components can only be designed independently of each other, if the architecture supports the precise specification of all component services at the level of architecture design. In a real-time system the RS-interface specification of a component must comprise the precise CNI specification in the value domain and in the temporal domain and a proper abstract model of the component service, as perceived by the user of the component. Only then is the component designer in the position to know exactly what can be expected from the environment and what must be delivered by the component.

Many of the present event-triggered architectures [8] do not provide the capability to define the temporal properties of the CNIs with the required rigor. The exact points in time when messages are expected to arrive or are supposed to be sent by a component, and the phase relationships between the messages are not contained in many CNI specifications. This loose specification of the CNIs makes it difficult for the component designer to thoroughly validate a component behavior in isolation, i.e., outside the system context.

For example, if the peak-load scenario of service requests to a component (rate and phase relationships between the requests) is not precisely specified at the architecture level, then vague assumptions about the system context of component use may replace the missing interface specifications. Such a component cannot be validated independently and may not operate correctly in another system context.

Stability of Prior Services: Principle two of a composable architecture is concerned with the design at the component level. A component is a nearly autonomous unit that comprises the hardware, the operating system and the application software. The component must provide the intended services across the well-specified component interfaces. The design of the component can take advantage of any established software engineering methodology, such as object based design methods. The stability-of-prior-service principle ensures that the validated service of a component--both in the value domain and in the time domain--is not refuted by the integration of the component into a system.

For example, the integration of a self-contained component, e.g., an engine controller, into the integrated vehicle control system may require additional computational resources of the component to service the RS-interface, both in processing time and in memory space. Consider the case where the CNI is based on a queue of messages: memory space for the queue must be allocated by the component-local operating system and processing time for the management of the queue must be made available. In a time-critical component it may happen that these additional resource requirements that are needed for the timely interface service, are in conflict with the resource requirements of the application software that implements the prior services of the component. In such a situation, failures in the component services may occur sporadically.

Constructive Integration: Principle three of a composable architecture is concerned with the design of the communication system. Normally, the integration of the components into the system follows a step-by-step procedure. The constructive integration principle requires that if n components are already integrated, the integration of the n+1 component may not disturb the correct operation of the n already integrated components. The constructive-integration principle ensures that this integration activity is linear and not circular.

This constructive integration principle has severe implications for the management of the network resources. If network resources are managed dynamically, it must be ascertained that even at the critical instant, i.e., when all components request the network resources at the same point in time, the timeliness of all communication requests can be satisfied. Otherwise sporadic failures will occur with a failure rate that is increasing with the number of components integrated.

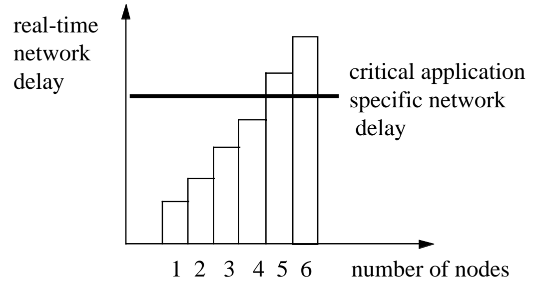

Figure 3: Maximum network delay at critical instant as a function of the number of nodes.

For example, if a real-time service requires that the maximum latency of the network must always remain below a critical upper limit (because otherwise a local time-out within the component may signal a communication failure) then the dynamic extension of the network latency by adding new components may be a cause of concern. In a dynamic network the message delay at the critical instant (when all components request service at the same instant) increases with the number of components. The system of Figure 3 will work correctly with up to four components. The addition of the fifth component may lead to sporadic failures.

Other applications, e.g., when a time-sensitive control loop is closed by the network, may require a network of known and constant jitter in order to support this principles of constructive integration.

If fault-tolerance is implemented by the replication of components, then the architecture and the components must support replica determinism. A set of replicated components is *replica determinate* [15] if all the members of this set have the same externally visible state, and produce the same output messages at points in time that are at most an interval of *d* time units apart (as seen by the omniscient outside observer). In a fault-tolerant system, the time interval *d* determines the time it takes to replace a missing message or an erroneous message from a node by a correct message from redundant replicas. The implementation of replica determinism is simplified if all components have access to a globally synchronized sparse time base.

6. VALIDATION

At the end of the development cycle it must be decided whether a given system is safe to deploy in the intended application domain. If this application domain is safety critical, i.e., a failure of the computer system can result in high financial loss or even a catastrophe where human lives are endangered then this decision is difficult [13]. Many safety critical applications demand a level of dependability that cannot be established by state of the art testing technology [12]. In this section we discuss some trends in the field of validation of high dependability real-time systems.

6.1 Process versus Product

Since it is beyond the state of the art to validate by testing that a large real-time system is free of critical design errors, the validation emphasis has shifted from the analysis of the product to the analysis of the development process of the product in the last few years (see, for example, the well-known ARINC-178/B standard [1] for software in airborne systems). It is my opinion that this trend will change during the next ten years. If composable architectures are widely deployed and real-time system design is guided by the recursive application of the two step design methodology (see Section 4.1) then the emphasis will shift back to product (component) validation. System design will consist of the reuse and integration of prevalidated hardware/software components. The temporal firewall concept [10], developed in the context of the time-triggered architecture (TTA), supports such a component validation by precisely specifying the value and temporal properties of the component interfaces at the architecture design level. These interface specifications establish the precondition and post condition for the component validation both in the value domain and in the temporal domain. It is thus possible to validate a component independently, i.e., outside the system context. Practical experience with this concept have demonstrated the significant benefits of this approach [7].

6.2 Worst Case Execution Time

The temporal firewalls of a component, specified at the architecture design level, identify the deadlines the component must meet under all specified operational conditions. During component design is must be demonstrated, that these deadline will never be missed. A necessary prerequisite for this temporal validation is knowledge about a tight upper bound of the worst case execution time (WCET) of all time-critical process inside a component [19]. This WCET analysis is an important research area in the field of real-time software the results of this research will have implications on other software areas, e.g., algorithm design, compiler design, and operating system design. For example, many of today's compiler try to optimize the average execution time, even if is this implies extending the WCET. In hard real-time systems a small WCET is of utmost importance, while the average execution time is of minor concern.

6.3 Simulation

Large real-time systems require a closed loop simulation in the laboratory to demonstrate that the system provides the intended services. In many cases, these simulations will be real-time simulations, where the effects of the real-time control system will be validated with respect to a real-time simulation model of the controlled object. In such an environment it is possible to study and validate the fault-tolerance properties of the control system, since it is possible to inject faults and to generate operational situations that will only occur rarely in the real environment (rare events). Such a rare-event simulation is also absolutely necessary to validate the peak-load performance of a high-dependability system. Today the methodological support for closed loop real-time simulation and fault injection of large systems is already an area of utmost industrial interest and this interest is likely to increase over the next ten years.

6.4 Formal Verification

A safety case is the accumulation of evidence from different sources that establishes the rational basis for the decision that a safety critical computer system is safe to deploy. The formal analysis of critical algorithms that are used in the system can form a convincing argument in the safety case [16]. The present formal validation technologies have already achieved a level of maturity that allows them to contribute to the validation of safety critical systems. In the future, the contributions of these formal methods are expected to increase.

7. CONCLUSION

It is dangerous to write a predictive paper about technology trends if the period of prediction is small enough that it is possible to confront the author--sometime in the future--with a comparison of prediction versus reality. Nevertheless I have engaged in this endeavor and tried to

outline a rough road map of the software engineering issues in real-time systems over the period of the next ten years. To me, the technological developments in the field of the computer hardware and the demands of new high-dependability applications will dramatically change the environment of the real-time software engineer. In my opinion, the most dramatic changes will be in the fields of composable architectures and systematic validation of distributed fault-tolerant real-time systems.

ACKNOWLEDGMENTS

This work has been supported, in part, by the IST project DSOS.

REFERENCES

[1] ARINC (1992). Software Considerations in Airborne Systems and Equipment Certification. ARINC, Annapolis, Maryland.

[2] Caro, D. (1998). What Does Real Time Mean Anyway. Instrument Society of America, http://www.isa.org/journals/ic/octfloor/html.

[3] Deirauer, P. and B. Woolever (1998). Understanding Smart Devices. *Industrial Computing*. Vol. pp. 47-50.

[4] Fabre, J. C., F. Salles, et al. (1999). Assessment of COTS Microkernels by Fault Injection. *Seventh Internatinal Working Conference on Dependable Computing*, San Jose, Cal. IEEE Press. pp. 19-38.

[5] Frank, R. (1995). *Understanding Smart Sensors*. London. Artech House.

[6] Garlan, D. (2000) *Software Architecture: A Roadmap.* in this volume.

[7] Hedenetz, B. and R. Belschner (1998). "Brake by Wire" without Mechanical Backup by Using a TTP Communication Network. *SAE World Congress*, Detroit Michigan. SAE Press, Warrendale, PA, USA.

[8] Kopetz, H. (1997). *Real-Time Systems, Design Principles for Distributed Embedded Applications; ISBN: 0-7923-9894-7, Third printing 1999*. Boston. Kluwer Academic Publishers.

[9] Kopetz, H. (1998). Component-Based Design of Large Distributed Real-Time Systems. *Control Engineering Practice–A Journal of IFAC, Pergamon Press*. Vol. **6**. pp. 53-60.

[10] Kopetz, H. and R. Nosssal (1997). Temporal Firewalls in Large Distributed Real-Time Systems. *Proceedings of IEEE Workshop on Future Trends in Distributed Computing*, Tunis, Tunesia. IEEE Press. pp. 310-315.

[11] Kopetz, H. and T. Thurner (1998). TTP--A new approach to solving the interoperability problem of independently developed ECUs. *SAE Congress 1998*, Detroit, USA. SAE Press 981107. pp. 1-7.

[12] Littlewood, B. and L. Strigini (1995). Validation of Ultradependability for Software Based Systems. *Predictably Dependable Computing Systems* B. Randell, J. L. Laprie, H. Kopetz and B. Littlewood Ed. Heidelberg. Springer Verlag. pp. 473-493.

[13] Lutz, R.L. (2000). *Software Engineering for Safety: A Roadmap.* In this volume

[14] OMG, 9. (1998). Real-Time CORBA. Object Management Group, Framingham, Mass.

[15] Poledna, S. (1994). Replica Determinism in Fault-Tolerant Real-Time Systems. Kluwer Academic Publishers, Boston.

[16] Rushby, J. (1993). Formal Methods and the Certification of Critical Systems. Computer Science Lab, SRI.

[17] Selic, B. (1999). Turning Clockwise: Using UML in the Real-Time Domain. *Comm. ACM*. Vol.42, No. 10, Oct. 1999. pp.46-54

[18] SIA (1997). National Roadmap for Semiconductors. Semiconductor Industry Association, http://notes.sematech.org/ntrs/Rdmpmen.

[19] Wilhelm, R. (1999). Special Issue on Timing Analysis and Validation for Real-Time Systems. *Real-Time Systems, Kluwer Academic Publishers.* Vol. **17**. pp. 127-287.