Fundamental Design Principles for Embedded Systems: The Architectural Style of the Cross-Domain Architecture GENESYS

R. Obermaisser, C. El Salloum, B. Huber, H. Kopetz
Vienna University of Technology
{ro,salloum,huberb,hk}@vmars.tuwien.ac.at

Abstract— **The GENESYS (Generic Embedded System) project is a European research project that aims to develop a cross-domain architecture for embedded systems. The requirements and constraints for such an architecture are documented in the ARTEMIS strategic research agenda in the form of seven key challenges. This paper presents the architectural style of GENESYS by listing the key architectural principles, such as: strict component orientation, separation of computation from communication, availability of a common time, hierarchical system structure, adherence to message passing, state awareness, fault isolation and integrated resource management. This paper explains how these architectural principles contribute to solve the seven key challenges in the ARTEMIS strategic research agenda.**

I. INTRODUCTION

Over the past thirty years the field of embedded systems has evolved in a bottom-up fashion in many different application domains. Driven by the ever improving price/performance ratio of microelectronic devices, many mechanical, hydraulic and electronic-analogue control systems have been replaced by embedded digital computer systems, resulting in control systems of lower cost, increased reliability and enhanced functionality. Examples are micro-controller based control systems in many appliances, computer-based engine control systems in cars, digital television and mobile phones, just to name a few. Since many additional benefits can be accrued if stand alone embedded control systems are interconnected, domain-specific real-time communication protocols and standards have been developed within the different domains in the next phase: examples are CAN for the automotive domain, AFDX for the aerospace domain, many different version of field-buses, such as Profibus, for the industrial domain or Firewire for multimedia applications. We are now in the situation that distinct application domains are coming together: Multimedia systems have entered the automobile and are expected to play an important role in obstacle recognition systems, the cross-domain technology of the Internet is being interfaced to many embedded control systems, giving rise to an 'Internet of things'. This convergence of the application domains is mainly driven by the following three forces:

i. *Interoperability:* There is an increasing need to link embedded subsystems from different domains: For example, in an industrial control system where smart cameras replace mechanical sensors, multimedia systems and control systems must be interconnected to provide a real-time service.
ii. *Technology Constraints:* The further miniaturization of semiconductor devices makes it very expensive to develop the masks for a new high-performance System-on-Chip. It is thus expedient to develop generic SoCs that can be deployed in many different application domains.
iii. *Human Resources:* If every domain has its own peculiar technical solutions concerning operating systems, communication protocols and standards, it is very difficult to make best use of the limited pool of human resources, since the transfer of an expert from one application domain to another requires extensive retraining efforts that cut into the productivity.

In recognition of these developments, the European Embedded Systems Industry in cooperation with the European Union has established the new European Technology Platform ARTEMIS with the mission to develop a cross-domain architecture and design methodology for embedded systems. In its Strategic Research Agenda [1] p. 16-17, the ambitious ARTEMIS vision and targets for the year 2016 are communicated: ARTEMIS is expected to deliver a cross-domain embedded system architecture that supports the cross-sectoral reusability of embedded system components, achieves a significant reduction in the development costs and the development cycle for new embedded applications, and provides a framework in order to manage the expected increase in the complexity of embedded systems functions without an undue increase in their development efforts.

In order to support an early first step towards the ARTEMIS goals, the FP 7 program of the EU has called for proposals for a cross-domain candidate architecture for ARTEMIS. As a consequence, the consortium GENESYS (Generic Embedded Systems) of 21 partners from industry, research institutes and academia has been formed to develop a blueprint of such an architecture candidate. This paper presents the first results of this research effort to develop a cross domain European Embedded System Architecture.

A computer system architecture is a framework that guides a designer and constrains a design such that the evolving hardware-software artifact is meeting a set of established requirements and possesses desirable properties. It follows that architecture design has to start with capturing and elaborating on the requirements and properties that are the desiderata of the looked-for artifacts. Architecture design can thus be structured into the following three distinct phases:

i. *Requirements Capture:* In a two year effort (from 2006 to 2007) a group of experts from industry and academia has captured the requirement and constraints for a cross-domain embedded system architecture [2]. The more than one-hundred requirements have been grouped under the headings composability, networking, security, robustness, diagnosis and maintenance, integrated resource management, and evolvability. The requirement capture phase has been completed by the end of the year 2007.

ii. *Definition of an Architectural Style:* During this phase, the architectural principles that characterize the architecture must be established. A principle is an accepted statement about some fundamental insight in a domain of discourse. Principles establish the basis for the formulation of operational rules. Design principles are thus the starting point for the derivation of detailed operational design rules and services. The Architecture principles of GENESYS have been established in the first half of 2008.

iii. *Definition of Architectural Services:* In this phase the established principles are operationalized in the form of a specification of detailed architectural services. This phase is ongoing and a first version of architectural services will be available by the end of 2008.

The rest of the paper is structured as follows. Section II describes the seven ARTEMIS challenges, which are the drivers for the presented architectural style. The focus of Section III are the principles of the architectural style. Each principle is explained and the implications for the ensuing system are explained. Section IV discusses the positive effects of the architectural principles regarding the handling of the ARTEMIS challenges. The paper finishes with a conclusion in Section V.

II. ARTEMIS CHALLENGES

The European Technology Platform (ETP) on embedded systems, ARTEMIS, has proposed a Strategic Research Agenda (SRA) that outlines the most important areas requiring research for embedded systems [2].

1. *Composability*. Architectures must support the constructive composition of large systems out of components and subsystems without uncontrolled side effects.
2. *Networking*. Subsystems have to communicate with each other and with the external environment at multiple levels (e.g., network-on-chip, cluster-level, local-area networks), observing given timing and dependability constraints.
3. *Security*. Novel approaches are required to deal with malicious attacks from intruders, to maintain the confidentiality of sensitive data and to protect intellectual property.
4. *Robustness*. This challenge comprises the provision of an acceptable level of service despite the occurrence of transient and permanent hardware faults, design faults, imprecise specifications, and accidental operational faults.

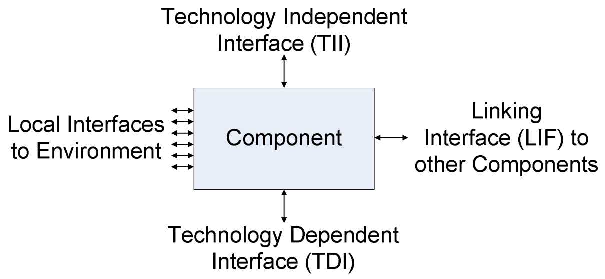
Technology Independent Interface (TII)
Local Interfaces to Environment
Component
Linking Interface (LIF) to other Components
Technology Dependent Interface (TDI)

Figure 1: Component Interfaces

5. *Diagnosis and Maintenance*. Instantiations derived from the architectural style should support the reliable identification of failed subsystems in order to autonomously recover from transient failures and guide untrained users in the replacement of defective subsystems with permanent failures.
6. *Integrated Resource Management*. Architectures need to support an integral management of resources (e.g., power, time, communication bandwidth, memory).
7. *Evolvability*. The instantiations derived from the architectural style shall support the evolution of systems, driven by novel hitherto unknown user requirements, and the need to integrate new technological capabilities.

III. ARCHITECTURAL PRINCIPLES

This section describes the principles of the GENESYS cross-domain architectural style. The principles are accepted statements about fundamental insights in the design of embedded system architectures and facilitate the solving of the seven ARTEMIS challenges.

A. *Strict Component Orientation*

The GENESYS architectural style follows a strict component orientation from the hardware/software point of view. A component is a self-contained subsystem that can be independently developed and used as a building block in the design of a larger system. A component is a replaceable part of a system that encapsulates the implementation and exposes a set of interfaces. Each component serves as a stable intermediate form that exhibits aggregating properties, when integrated with other components to an ensuing system. As expressed in [3], '*complex systems will evolve from simple systems much more rapidly if there are stable intermediate forms than if there are not.*' A component can have a complex internal structure that is neither visible, nor of concern, to the user of the component at the architecture level. Thereby, a component offers an appropriate unit of abstraction for the design.

What needs to be known about a component at the architecture level is captured in the specification of the component's interfaces. Figure 1 depicts the component interface structure. We distinguish between the *Linking Interface* (LIF), the *Technology Independent Interface* (TII), the *Technology-Dependent Interface* (TDI), and the *local interfaces*.

The LIF is the interface for the integration of components. The LIF of a component abstracts over the internal structure and accumulates relevant information of the other interfaces of the component. The LIFs need to be technology independent in the sense that the LIF does not incorporate implementation details of a component. A technology independent LIF ensures that different implementations of computational components can be integrated (e.g., general purpose CPU, FPGA, ASIC).

Local interfaces are interfaces to the component environment (e.g., using sensors and actuators) or to other subsystems (i.e., in case of a gateway). The local interfaces (e.g., technology-specific interface to transducer) of a component are not constrained by the LIF specification. However, the specification of the LIF includes those properties of the local environment that are of relevance for the integration.

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 Section III.F). The TII is agnostic about the component implementation.

Depending on the ability to reconfigure a component via its TII, we distinguish between soft components and hard components. The functionality of a *hard component* is determined by the hardware and cannot be modified via the TII. In a *soft component*, the functionality is determined by software on a CPU or an FPGA. Hence, the functionality of soft components can be modified during the lifetime of the component. Actual components can also be a mixture of soft and hard components, i.e., supporting partial modification of the functionality. Soft components are important for dynamic reconfiguration in order to allow systems to evolve and adapt to a changing context. However, functionality implemented in a hard component can have superior non functional properties (e.g., energy efficiency, silicon area). In addition, mechanisms are required to protect the system from malicious users that try to use soft components for malicious purposes.

Since the LIF abstracts over the component implementation, the difference between hard and soft components is not visible at the LIF.

The *Technology-Dependent Interface* (TDI) of a component is used to provide a view into the internals of the concrete component implementation for a maintenance technician. 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.

B. *Strict Separation of Communication and Computation*

An important principle is the *consequent separation of (computational) components from the communication infrastructure*. The communication infrastructure provides the means to integrate the components to the overall system. Through the separation of concerns between communication and computation, the design complexity is reduced by allowing the design and analysis of the two parts in isolation. This separation also enables to exploit the locality of changes: As long as the behavior at the interface of the communication infrastructure remains unchanged, components are unaffected by modifications of the internal implementation. Furthermore, components can be reused independently from the communication infrastructure. Likewise, a generic communication infrastructure can be used with different computational components.

The communication infrastructure of the architecture is based on the paradigm of message passing. In this paradigm, the basic interaction mechanism among components is the unidirectional exchange of messages. Every message has a specific identifiable sender and one or more receivers. In case of multiple receivers, the communication infrastructure employs multicasting to deliver the message. Multicasting is required in many real-time applications, e.g., for fault-tolerance by active redundancy or in control loops. In particular, multicast supports the non-intrusive observation of the component interactions by an independent fault containment region.

The message is a concept that combines the temporal and the value domain. The temporal properties of a message include information concerning the temporal order, the interarrival time (e.g., periodic, sporadic, aperiodic recurrence), or the latencies. Messages provide inherent consistency (i.e., atomic delivery of the message contents) and can be used for synchronization. Therefore, the message paradigm provides a higher level of abstraction compared to bus interfaces.

A single well-designed message passing service also provides a simple interface of a component to its environment. The message concept is a universal model, since on top of a basic message passing service it is possible to realize other communication paradigms such as a shared memory (e.g., [4], [5]) or higher communication protocols (e.g., Controller Area Network [6]).

C. *Availability of a Common Time*

Another general principle of the GENESYS cross-domain architectural style is the provision of a common time base to all components. A common time base allows the temporal coordination of distributed activities (e.g., synchronized messages), as well as to interrelate timestamps assigned at different components.

In an ensemble of clocks, each with its own oscillation mechanism and starting with the same initial value, the clocks will diverge over time due to differences in the physical oscillation mechanisms—the differences in the rates of the clocks. In order to keep the clocks in approximate synchrony, we must resynchronize the clocks periodically. In an ensemble of good synchronized clocks (a clock which has failed is a bad clock) we call the maximum duration between the respective ticks of any two good clocks – as measured by the reference clock – the precision of the ensemble. The precision of the common time base depends on the integration level. For example, the precision on a chip will be significantly better than the precision in an open system of loosely coupled devices. In addition, the common time can be synchronized with an external time source (e.g., GPS).

In safety-critical systems the common time base must be fault-tolerant, since the correct provision of safety-related services can depend on the availability of the common time base. Therefore, the GENESYS architectural style has the option to support the use of fault-tolerant clock synchronization algorithms that have been developed for different types of fault assumptions.

D. *Hierarchical System Structure*

The presented architectural style introduces three different integration levels: the *chip-level*, the *device-level*, and the *system-level* (see Figure 2). At the system level *open* and *closed systems* are distinguished. At each level, the architecture provides *core services* and *optional services*.

The reason for the introduction of these integration levels is that the service characteristics of the three levels are substantially different, e.g., the bandwidth in a network-on-chip (NoC) is orders of magnitude cheaper than the bandwidth at the system level (e.g., WLAN). Furthermore, temporal guarantees can only be given in a closed system.

A system consists of devices, each of which is a spatially and logically self-contained apparatus (e.g., ECU in a car, mobile phone, DVD player, smart transducer with a standardized network interface, …). The LIFs of a device serve for the connection with other devices (e.g., wireless connection between a mobile phone and a music player, wirebound connection between ECUs in a car). A LIF is of relevance for the architecture and must be compliant to the GENESYS architectural style. The interconnection of devices can occur in an open environment or closed environment. In an *open environment* the integration of devices always occurs dynamically during the system operation without a priori knowledge concerning the participating devices. A closed system is a system where all devices are known a priori. In a *closed environment*, the integration of devices can be static or dynamic.

The precise form of the local interfaces, the TDI and the TII is not relevant for the system integration, since the semantic properties of these interfaces are captured in the semantic specification of the LIF [7].

If a device has an internal structure that is relevant from the point of view of the architecture, then the device can be further decomposed into a set of chips that interact via inter-chip LIFs. The inter-chip LIFs are established using off-chip communication systems such as PCI or SPI.

If a chip is an MPSoC that has an internal structure with relevance from the point of view of the architecture, then the chip can be decomposed into a set of IP cores. The IP cores communicate with each other using inter-IP core LIFs via networks-on-a-chip.

The connection between integration levels occurs via *gateway components*. For example, at the device-level a chip

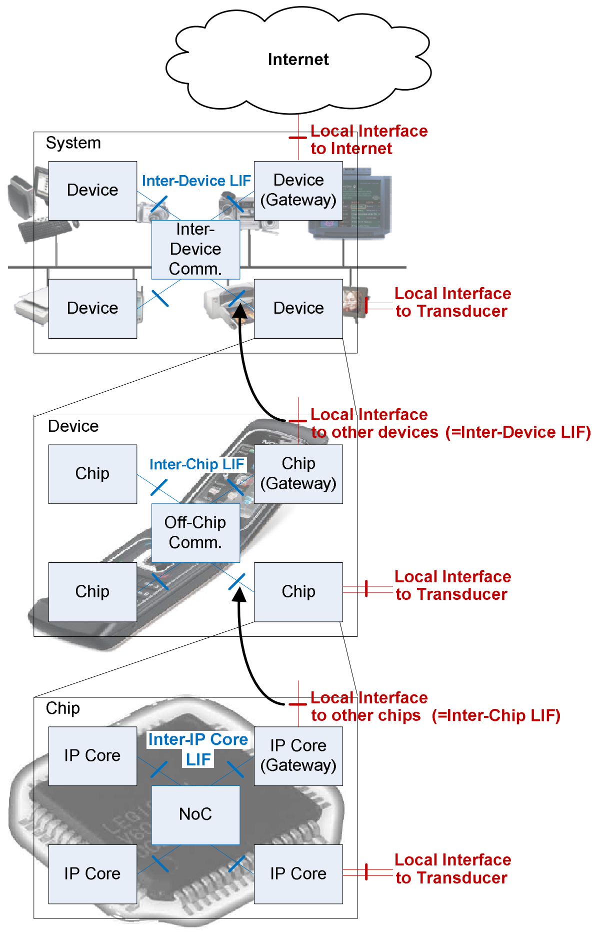
Internet
Local Interface to Internet
System
Device
Inter-Device LIF
Device (Gateway)
Inter-Device Comm.
Device
Device
Local Interface to Transducer
Local Interface to other devices (=Inter-Device LIF)
Device
Chip
Inter-Chip LIF
Chip (Gateway)
Off-Chip Comm.
Chip
Chip
Local Interface to Transducer
Local Interface to other chips (=Inter-Chip LIF)
Chip
IP Core
Inter-IP Core LIF
IP Core (Gateway)
NoC
IP Core
IP Core
Local Interface to Transducer

Figure 2: Integration Levels of GENESYS

can serve as a gateway component by mapping the inter-chip LIFs to an inter-device LIF. Such a chip bridges the gap between the device and system level. What is important is that when integrating chips at the device-level, only the inter-chip LIF of this gateway component needs to be considered, while the inter-device LIF becomes a local interface. Adversely, when integrating devices at the system level, only the inter-device LIF of the gateway component is relevant and the inter-chip LIF acts as a local interface.

E. *Sporadic, Periodic and Streaming Communication*

The GENESYS architectural style recognizes that different application subsystems can have different requirements concerning the underlying communication infrastructure. For this reason, the following three communication modes are provided by the architectural services of GENESYS:

i. *Periodic messages*: Periodic messages provide a predictable time-guaranteed communication service. The instants for the transmissions of periodic messages are specified by a communication schedule, which prevents any collisions between messages a priori. For periodic messages, the communication infrastructure guarantees temporal properties such as latency,

latency jitter, bandwidth, and message order. The relation between the transfer time and the length of messages is deterministic. Periodic messages are well-suited for cyclic applications, e.g., control applications.

ii. *Sporadic messages:* Sporadic messages are characterized by bandwidth, sender queue length, and receiver queue length and provide a flexible communication service. The transmission of sporadic messages can be triggered by any significant event, i.e., not necessarily by time events. Due to the potential unpredictability of the events that trigger sporadic messages, this communication mode offers weaker temporal guarantees, e.g., best-effort communication or probabilistic statements about the bandwidth or latency.

iii. *Synchronized data streams*: Data streams enable the transmission and on-the-fly processing of streaming data. Many applications require the temporal alignment of data streams (e.g., synchronized of video and audio sources like lip sync) or multiple video streams (e.g., as input for pattern recognition). For this purpose, data streams from multiple sources can be synchronized.

F. *State Awareness*

Shrinking geometries, lower power voltages and higher frequencies result in a significant increase of transient failure rates.

Most of the transient faults corrupt the internal state of a component but have no permanent effect on the component hardware. Thus, the robustness of the overall system can be significantly increased if the state of a component that has failed due to a transient fault is automatically recovered and if, in a subsequent step, the component is reintegrated into the system. State aware design is required for making the state of a component explicit which helps significantly for state recovery.

In the GENESYS architectural style we assume that each component regularly reaches a so-called *reintegration point*, at which the component state is clearly defined and can be restored. The planning of these reintegration points is part of the system design phase. The state at a reintegration point contains all information about the past history of the component that is relevant for the component's future behavior. A good candidate for a reintegration point is a periodically recurring point in time where the component state is minimal. The state at the reintegration point is called the *ground state* and the period with which the ground state is reached is called the *ground cycle*. In order to simplify component restart, the ground state can be externalized at the component's LIF in the form of a *ground state message*. The ground state (e.g., stored at replicas) can be relayed back to the component when performing a restart. We speak of a *restart message* to denote the message that carries the externalized ground state to a restarting component.

Externalizing a component's state via a ground state message is not only beneficial for recovery, but enables also online diagnosis without any probe effect and simplifies testing and debugging.

G. *Fault Isolation*

Fault isolation is an important issue when dealing with both *physical faults* and *design faults*. As mentioned in the previous section, increasing transient failure rates will occur with respect to physical faults. Design faults are associated with more and more complex designs, which result from the increasing functionality expected from embedded systems.

In the GENESYS architectural style, fault containment is based on the clear definition of *fault containment regions* (FCRs). An FCR operates correctly regardless of any arbitrary logical or electrical fault outside the region [8]. Conversely a fault in a fault-containment region cannot cause hardware/software outside the region to fail.

The structuring rules of the GENESYS cross-domain architectural style guide designers in such a way that components represent fault containment regions. Components are physical entities (i.e., IP cores, chips, or devices) and should be aligned with the logical entities (i.e., the *application jobs* which are the basic unit of work in the GENESYS architectural style). This alignment between physical and logical entities is the key mechanism for the establishment of clear fault containment regions.

In order to establish fault containment regions, the GENESYS architectural style foresees the realization of *temporal* and *spatial partitioning* [9]. Temporal partitioning ensures that a component cannot affect the guaranteed availability of communication resources (e.g., time of availability, duration or jitter of availability) to other components. Spatial partitioning ensures that a third component cannot affect the integrity of information exchanged between any other two components.

An aspect of particular importance is fault isolation with respect to intentional faults. In the near future, devices belonging to different application domains will interact with each other in order to provide emergent services and added value to the user (e.g., a mobile phone may act as a key for a car). This increased connectivity and extensibility implies that security is a new dimension in the design of safety-critical systems. Therefore, the fault isolation mechanisms of the GENESYS architectural style aim at tolerating perturbations such as accidental faults (e.g., transient or permanent hardware faults caused by radiation, quantum-mechanical effects, electromagnetic interference or aging) as well as malicious attacks aiming at violating desired security properties.

H. *Resource Awareness and Control*

A common characteristic of embedded systems across application domains is the need to dynamically adapt to the (changing) context. For this purpose, integrated resource management, which follows a holistic view on different resources (e.g., power, energy, communication bandwidth, memory) is established as an architectural principle for GENESYS. In many cases, trade-offs between different resources exist that need to be considered. For example, high computing performance typically conflicts with low power operation. When assigning platform resources to the application, it is thus a challenge to optimize for a low power solution, while satisfying the performance constraints.

Applications can have multiple application quality levels, which are selected dynamically based on the user preferences and platform resource availability.

Since a single resource allocation strategy cannot be expected to fit for all applications and domains, different resource allocation strategies must be supported. For instance, resource management has to be supported for safety-critical applications, non safety-critical applications, and mixed criticality systems with both safety-critical and non safety-critical subsystems. Therefore, the GENESYS architectural style supports hard resource guarantees or enhanced service modes in safety-related subsystems, while sharing of chips and devices among safety-critical and non-critical services. In addition, predictability of the service availability after reconfiguration is required. In particular, the effects of reconfiguration to critical services should be known beforehand and reconfiguration activities should not disrupt the behavior of subsystems that are not subject to the reconfiguration activities.

For many applications power/energy awareness and control is a key requirement. The strict component orientation of the GENESYS architectural style facilitates this in multiple ways: Firstly, the power/energy consumption for one application can be transparently and nearly independently altered without disturbing the operation of other applications (e.g. by reducing supply voltage and clock frequency of one component while keep other components unmodified). Secondly, the strict component orientation eases the gradual change of component implementations to different implementation technologies (e.g., software on CPU, FPGA, ASIC). This is important, because different implementation technologies can differ with respect to power/energy efficiency by multiple orders of magnitude. Thirdly, in operation the GENESYS architectural style enables the collection, storing and provision of information about the energy, thermal and power conditions and capabilities (e.g., energy supply state, current power usage and estimated usage time, peak power) of individual components. Thus, allowing for creating a structured, holistic view on the resource usage.

Resource control in the context of GENESYS also encompasses the transparent relocation of application functionality to different components (e.g., to overcome resource shortages or to raise availability in case of permanent faults). This is facilitated by the establishment of separate namespaces for logical and physical system structuring to provide location transparency for applications. Location transparency requires that the identifier of an entity does not necessarily reflect the physical location of the entity or the user. For this reason, we perform a clear distinction between addressing and identification. In particular, with respect to the message-based communication all communication primitives are identical regardless of the physical position of the communication partners.

I. Waistline structure of services

A central issue in the GENESYS architectural style is the provision of standardized, validated and certified architectural services (e.g., communication, diagnostic, or fault-tolerance services) that facilitate the development of embedded real-time applications. These architectural services separate the application functionality from the underlying platform technology in order to reduce design complexity and to facilitate design reuse, which corresponds to the concept of platform-based design [10].

The GENESYS architectural style is intended to be used across multiple heterogeneous application domains, with highly specific and potentially mutually contradicting requirements with respect to the architectural services. Examples for different requirements are the desired properties of communication protocols: Safety-critical applications (e.g., steer-by-wire systems) that have to deliver a given service within a guaranteed time bound require communication services that are highly deterministic in the temporal domain (e.g., time-triggered protocols). In contrast to safety-critical applications, many non-safety-critical systems typically have to be optimized for average load in order to optimize cost (e.g., the comfort electronic system of car) and thus require a highly flexible protocol that adheres to the event-triggered communication paradigm.

Consequently, the set of provided architectural services has to be *extendable* and *configurable*. Extendability is an important factor since it is not possible to anticipate how applications and their requirements will evolve in the future. Configurability is required to facilitate the construction of resource-efficient systems. Therefore, the architecture should be adaptable in a way, that any particular instantiation contains only those services that are required in the actual system.

In order to support the flexible integration of architectural services for heterogeneous application domains, the GENESYS architectural style introduces a *waistline structure of services*, which is based on a generic and minimal set of mandatory, validated and verified *core services*. The core services constitute the waist of the architecture. On top of this waist a broad range of *optional services* can be realized which can be tailored to the requirements of specific application domains and refine or extend the core services.

IV. Impact on ARTEMIS Challenges

This section discusses the implications of the presented architectural principles on the seven ARTEMIS challenges. Figure 3 lists the benefits regarding the ARTEMIS challenges, which will be discussed in the following.

A. Composability

A major requirement for the constructive composition of large systems out of components is the precise specification of the component's interfaces. In the case of real-time systems these interfaces have to be specified in the *value domain* as well as in the *temporal domain*. The precise specification of interfaces is supported by multiple design prin-

Architectural Principle	ARTEMIS Challenge						
	Composability	Networking	Security	Robustness	Diagnosis	Integrated RM	Evolvability
strict component orientation	**	*	*	**	**	**	**
separation of communication and computation	**	**	*	**	**	*	**
availability of a common time	**	*	*	*	*		
hierarchical system structure	**		*	*	*	*	*
periodic,sporadic and streaming communication	**	**	*	*	*	*	*
state awareness	*		*	**	**	*	*
fault isolation	**		**	**	**		*
resource awareness and control	*	*		*	*	**	*
waistline structure of services	**	**	*	*	*		**

Figure 3: Principles and ARTEMIS Challenges

ciples. The principle *strict component orientation* leads to systems that are decomposed into nearly autonomous components with clearly defined interfaces. Furthermore, this principle significantly reduces the complexity when integrating components into a larger system by limiting the interactions among components to occur exclusively via messages exchanged over the component's LIFs. The principles *separation of communication and computation* and *periodic, sporadic and streaming communication* identify the interface to the communication infrastructure. The *availability of a common time* enables the temporal specification of component interfaces in a way that the interactions of components in a distributed system can be temporally coordinated.

When composing a large system out of many components – which are potentially developed by different suppliers and may belong to different criticality levels – it has to be assured that a faulty component can affect only other components that depend via the input on this component but not the entire system. This property is supported by the principle *fault isolation*, which assures that a faulty component can affect other components exclusively by providing faulty input.

The principle *waistline structure of services* introduces layering as a means of complexity management with respect to services. For example, if an optional service is built on a core service, only the interface to the underlying core service is relevant but its implementation is exchangeable.

The principle *hierarchical system structure* introduces abstraction as a means of complexity management. At the chip-level, IP cores are composed to build a chip. At the device-level chips are composed to build a device and the internal structure of a chip can be neglected. The components that are composed at the system-level are devices. Again, the internal structure of a device is not relevant at the system level.

B. *Networking*

The networking challenge requires the support for communication with different communication protocols at multiple levels (e.g., network-on-chip, board-level, cluster-level, local-area networks, wide-area networks).

A key principle of the architectural style, which targets this challenge, are the three *communication modes* introduced in Section III.E. These communication modes are universal in the sense that any communication protocols can be layered on top of the three communication modes. Event-triggered communication protocols, such as CAN [6] or IP can be layered on top of the sporadic message transport service. Protocol-specific interaction patterns can be constructed with sequences of sporadic messages. For example, a client/server interaction can be realized with a request and a reply message.

The periodic communication mode allows to establish time-triggered communication, e.g., as provided by protocols such as FlexRay [11], TTP [12] or TTCAN [13]. Finally, the streaming communication is a communication abstraction that fits to streaming protocols such as TCP/IP.

The principle of a *waistline structure* helps in the networking challenge, because using optional services different protocols can be layered on top of the three communication modes that are supported by the core services. With distinct optional services, different protocols (or protocol stacks) can coexist in a single system. For example, a legacy application using CAN communication can coexist with safety-critical time-triggered applications and a multimedia application using TCP/IP communication.

Another strong point for the networking challenge is the *separation of communication and computation*. Thereby the implementation of the communication infrastructure can be modified without implications on the computational components.

C. *Security*

The architectural principles *strict component orientation*, *separation of communication and computation* and *fault isolation* ensure that components constitute FCRs by design and that the interactions between components are restricted to occur exclusively via messages exchanged over their LIFs. Out of these architectural properties, a security feature emerges which is called *intrusion containment*. The temporal and spatial partitioning properties of the architecture draw a boundary with respect to potential consecutive attacks after a component has been compromised. A compromised component can influence any other component exclusively by means of the exchange of malicious messages (e.g., a piece of malicious code in one component cannot overwrite memory locations in any other component). Thus, security considerations can focus on the messages exchanged over the component's LIFs, which significantly reduces the validation effort of a system's security properties.

In the case of closed systems, the temporal partitioning of the communication subsystem ensures that a component cannot affect the guaranteed availability of communication resources (e.g., time of availability, duration or jitter of availability) to other components, and thus helps in repelling denial-of-service (DOS) attacks.

The *availability of a common time* throughout the system enables a component to interpret timestamps that have been generated by another component. These timestamps can be integrated into signed messages or other signed data structures in order to prevent *replay attacks*.

The principle of *state awareness* ensures that the state of a component can be made explicit. An explicit notion of state can be used for monitoring and analysis in order to implement security relevant features like *intrusion detection*.

D. Robustness

The architectural style facilitates robustness in the presence of different types of faults, including transient and permanent hardware and software faults.

Firstly, each component represents a clear fault containment region due to the *strict component orientation* anpd *fault isolation* of the architectural style. With a given assumption coverage, each component operates correctly regardless of any hardware or software fault outside the component.

Due to the consequent *separation of communication and computation*, the communication infrastructure can also be designed in such a way that its correct operation is not affected by faulty components. The communication system can operate autonomously, thereby realizing a so-called temporal firewall [14].

Based on the fault containment, the architectural style enables error containment. Fault-tolerance for hardware faults can occur using active redundancy. The waistline architecture permits optional services for voting (e.g., in a Triple Modular Redundancy (TMR) configuration), thereby performing active redundancy transparently to the applications. Furthermore, the global time, the clear notion of state, and the communication modes support determinism as a basis for voting on a bit-by-bit basis. In all communication modes, collisions between messages can be avoided at design time, thus eliminating the potential for indeterminism through race conditions. For example, the TT-SoC architecture [15] is an MPSoC architecture that follows the presented principles at the chip-level. This architecture employs a network-on-a-chip without dynamic arbitration in order to ensure determinism.

The *state awareness* is a key principle for the robustness w.r.t. transient faults. The externalized state of replicas can be used to recover from a transient fault using a reset of the component. In addition, the detection of transient faults can occur by monitoring the externalized state or comparing the state of replicas.

E. Diagnosis and Maintenance

The GENESYS architectural style supports diagnosis and maintenance in several ways. Firstly, due to the explicit interaction between components solely via their LIFs, a systematic diagnostic service can monitor the correct behavior of a component by building assertions on the exchanged messages. That means without the need to look inside a component. This way, a separation of concerns between diagnosis and application behavior can be performed.

Another essential diagnostic principle is to ensure the traceability of system behavior. This principle involves the establishment of a view of consistent distributed system state despite independence of components. By externalizing and regularly distributing the ground state of components as stated in Section 3.6 a diagnostic service is supported in establishing a view of a consistent distributed system state.

With respect to maintenance, the architectural principle of fault isolation helps the maintenance engineer to identify the component on which a maintenance action has to be carried out (e.g., new software to resolve a design fault, replacement of a hardware part to resolve a permanent hardware fault). Fault isolation suppresses the propagation of faults and thus improves the traceability of the origin of the fault.

F. Integrated Resource Management

To tackle the challenge of integrated resource management, the support for *resource awareness and control* is incorporated as architectural principle into the GENESYS architectural style. The necessity of a self-contained LIF specification of each component (cf. Section 3.1) makes the resource requirements of individual applications (e.g. communication requirements, power requirements) explicit. This enables a resource manager to create a holistic view of the resource requirements of the overall system.

This *strict component orientation* increases also the independence of resource management decisions. In particular on the lowest integration level – the chip level – fine-grained resource allocation and control can be achieved by a strict mapping of single application jobs to IP cores. This also reduces the complexity of the resource management decisions, since, e.g., the reduction of resources granted to one IP core, has only a direct impact on a single job.

One explicitly mentioned challenge in the ARTEMIS SRA is the efficiency of future embedded systems in regard to low operating and standby power, in particular for battery operated devices. A strategy for tackling this challenge is the temporary shut-down of particular components whenever they are not required. The identification of a ground state and its periodic externalization as stated in the GENESYS architectural principles helps in speeding-up the recovery from standby and ensuring that the state of component is not lost.

G. *Evolvability*

Every successful system must adapt to its changing context and will evolve as new requirements emerge and new implementation technologies become available.

The principle of *strict component orientation* improves evolvability, because the technology-independent LIF makes it possible to replace the implementation of a component without modifications of the LIF. Thus, technological changes can be accommodated without inducing effects at the system level.

Furthermore, modifications of the communication infrastructure and modifications of the computational components do not affect each other as long as the specified behavior at the interfaces remains unchanged. This is a consequence of the principle '*separation of communication and computation*'.

Evolvability of the architecture services is supported by the *waistline structure of services*. Additional optional services can refine and extend existing services. Furthermore, new stacks of architectural services can be established for new types of applications.

V. Conclusion

The European industry recognizes the seven ARTEMIS challenges as the enabler for more advanced embedded systems in many application domains including automotive, avionic, industrial control, consumer electronic and mobile systems. This paper presents fundamental architectural principles that provide a major step forward towards the solving of these challenges. Each of the principles of the architectural style is not only explained, but the paper also describes how the architectural principles help to solve the ARTEMIS challenges.

Acknowledgment

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 are warmly acknowledged.

References

[1] ARTEMIS, "Strategic Research Agenda - First Edition," 2006.

[2] ARTEMIS, "Reference Designs and Architectures. Constraints and Requirements. Report Version 13," 2006.

[3] H. Simon, The Sciences of the Artificial: MIT Press, 1996.

[4] K. Li and P. Hudak, "Memory coherence in shared virtual memory systems," ACM Transaction of Computer Systems, vol. 7, pp. 321-359, 1989.

[5] B. Fleisch and R. H. Katz, "Mirage: A coherent distributed shared memory design," in 12th ACM Symposium on Operating Systems, 1989, pp. 211-223.

[6] Robert Bosch GmbH, "CAN Specification, Version 2.0," 1991.

[7] H. Kopetz and N. Suri, "Compositional design of RT systems: A conceptual basis for specification of linking interfaces.," in Proc. of the Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2003, pp. 51–60.

[8] J.H. Lala and R. E. Harper, "Architectural principles for safety-critical real-time applications," Proc. of the IEEE, vol. 82, pp. 25–40, 1994.

[9] J. Rushby, "Partitioning for Avionics Architectures: Requirements, Mechanisms, and Assurance," NASA Langley Research Cen-ter1999.

[10] A. Sangiovanni-Vincentelli, "Defining platform-based design," EEDesign of EETimes, 2002.

[11] FlexRay Consortium, "FlexRay Communications System Protocol Specification Version 2.1," 2005.

[12] H. Kopetz and G. Grünsteidl, "TTP - A protocol for fault-tolerant real-time systems," Computer, vol. 27, pp. 14-23, 1994.

[13] T. Fuhrer, B. Müller, W. Dieterle, F. Hartwich, and R. Hugel, "Time-triggered CAN - TTCAN: Time-triggered communication on CAN," in 6th International CAN Conference (ICC6), 2000.

[14] H. Kopetz and R. Nossal, "Temporal firewalls in large distributed real-time systems," in 6th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS '97), 1997.

[15] R. Obermaisser, C. E. Salloum, B. Huber, and H. Kopetz: "The Time-Triggered System-on-a-Chip Architecture," in IEEE International Symposium on Industrial Electronics, 2008.