Virtual Gateways in the DECOS Integrated Architecture

R. Obermaisser, P. Peti, H. Kopetz
Vienna University of Technology, Austria
email: {ro,php,hk}@vmars.tuwien.ac.at

***Abstract*— The DECOS architecture aims at combining the advantages of federated and integrated systems. The DECOS architecture divides the overall system into a set of nearly-independent distributed application subsystems, which share the node computers and the physical network of a single distributed computer system. Through restricting each application subsystem's communication activities to a corresponding encapsulated virtual network realized as an overlay network on a physical time-triggered backbone, the integrated DECOS architecture supports fault isolation, complexity management, and the independent development of application subsystems.**

This paper provides a solution to the controlled export and import of information between distributed application subsystems. We give the designer the ability to coordinate application services and exploit redundancy in the system to either improve reliability or reduce resource duplication. We introduce virtual gateways for the coupling of virtual networks by the selective redirection of messages. Virtual gateways not only resolve property mismatches between distributed application subsystems, but also preserve encapsulation. We capture the essential properties of each application subsystem in a message description based on timed automata and use this description as a parameterization of generic architectural gateway services.

***Index Terms*— Integrated Architecture, Gateway, Error Containment, Timed Automata, Virtual Network**

I. Introduction

One can distinguish two classes of systems for distributed applications, namely *federated* and *integrated systems*. In a federated system, each application subsystem has its own dedicated computer system, while an integrated system is characterized by the integration of multiple application subsystems within a single distributed computer system. Federated systems have been preferred for ultra-dependable applications due to the natural separation of application subsystems, which facilitates fault-isolation and complexity management.

Integrated systems, on the other hand, promise massive cost savings through the reduction of resource duplication. In addition, integrated systems permit an optimal interplay of application subsystems, reliability improvements with respect to wiring and connectors, and overcome limitations for spare components and redundancy management. An ideal future system architecture would thus *combine the complexity management advantages of the federated approach, but would also realize the functional integration and hardware benefits of an integrated system* [1, p. 32]. The challenge is to devise an integrated architecture that provides a framework with generic architectural services for integrating multiple application subsystems within a single, distributed computer system, while retaining the error containment and complexity management benefits of federated systems.

The DECOS integrated architecture [2] is an integrated system architecture that aims at fulfilling this requirement. The DECOS integrated architecture divides the overall system into a set of encapsulated Distributed Application Subsystems (DASs), each with dedicated computational and communication resources. The communication resources for the different DASs are provided through *virtual networks* [3], which are realized as overlay networks on top of a time-triggered physical network. Each virtual network forms the communication infrastructure of a DAS and runs a communication protocol tailored to the needs of the respective DAS. A virtual network exhibits specified temporal properties, which are independent from the communication activities in other virtual networks. The partitioning of the overall system into DASs with encapsulated virtual networks helps in managing complexity, because a DAS can be understood independently from other DASs. In addition, this strategy facilitates independent development, since each DAS can become the responsibility of a corresponding vendor. Furthermore, error containment between DASs results from the fact that a message failure in one DAS cannot propagate into another DAS.

Even with a strict separation of DASs, the integrated DECOS architecture would already permit a considerable reduction in the numbers of components and wiring through the sharing of components and networks among DASs. However, controlled interactions between DASs are required for unleashing the full advantages of the integrated approach.

On the one hand, the quality of control of a real-time computer system can be improved when different

control functions are coordinated to achieve a tactic behavior. For example, in order to improve fuel efficiency in airplanes, a coordinated strategy between autopilot and autothrottles is required. If such a strategy is not realized, i.e. autopilot and autothrottles are designed as totally independent systems, autothrottle speed control constantly upsets the autopilot flight path control and vice versa [4, chap. 6-3]. In the automotive industry, an example of the coordination of different application subsystems for improving the quality of service with respect to passenger safety is the passive safety mechanism (Pre-Safe) of the Mercedes S-class [5]. The Pre-Safe system tensions seat-belts, realigns seats to a safer position, and closes an open sun roof when sensors detect possibly hazardous situations. The system correlates information of existing car dynamics sensors in order to determine hazardous situations such as skidding, emergency braking, or avoidance maneuvers.

Secondly, in a large real-time computer system, different DASs will typically depend on the same or similar sensory inputs and computations. By adapting encapsulation to allow for the exporting and importing of information between DASs, one DAS can use services (e.g., acquisition of sensory information or computations) in the other DASs and does not need to provide the services on its own. For example, in an automotive system the speed sensors from the factory installed Antilock Braking System (ABS) can be exploited to estimate the car's heading for the navigation system during periods of GPS unavailability [6]. The redundant sensors can be eliminated in one of the DASs leading to reduced resource consumption and hardware cost. Alternatively, redundancy can be exploited to improve the reliability of the sensory information. Even sensory information from different physical entities can be exploited by sensor fusion [7].

This paper presents a solution for controlled interactions between DASs in order to achieve functional integration, reliability benefits, and reduced resource consumption. As part of the integrated architecture that is developed in the Dependable Embedded Components and Systems (DECOS) EU Framework Programme 6 [2], we introduce virtual gateways as generic architectural services for the controlled coupling of the virtual networks of DASs. A virtual gateway selectively redirects information between DASs. By making only a part of the information of one DAS visible to another DAS, one can abstract from the detailed behavior of other DASs when trying to understand a DAS. In addition, gateways perform error detection to control the forwarding of information and prevent the propagation of timing message failures. Furthermore, virtual gateways address operational differences and incoherent naming of interconnected DASs. Virtual gateways resolve these mismatches based on a formal description of the properties of each DAS.

Virtual gateways permit tactic coordination and exploitation of redundancy without having to fuse different control functions into a single DAS. Consequently, complexity benefits, fault isolation, and independent development are not compromised by the mutual coordination of control functions.

The paper is structured as follows. Section II gives a short overview of the DECOS integrated architecture that employs virtual gateways for the coupling of DASs. The concept of virtual gateways is described in more detail in Section III. We describe the operation of a virtual gateways and present a solution as part of the integrated DECOS architecture. Section IV discusses the establishment of virtual gateways at the component level. We sketch the application of virtual gateways in Section V as part of an exemplary automotive system. The paper finishes with a conclusion in Section VI.

II. DECOS Integrated Architecture

The DECOS architecture [2] offers a framework for the development of distributed embedded real-time systems integrating multiple DASs with different levels of criticality and different requirements concerning the underlying platform. Structuring rules guide the designer in the decomposition of the overall system both at a functional level and for the transformation to the physical level. In addition, the DECOS integrated architecture aims at offering to system designers generic architectural services, which provide a validated stable baseline for the development of applications.

A. Functional System Structuring

For the provision of application services at the controlled object interface, the services of a real-time computer system are divided into a set of nearly-independent DASs. Each DAS is further decomposed into smaller units called *jobs*. A job is the basic unit of work and exploits a *virtual network* [3] in order to exchange messages with other jobs and work towards a common goal. A *virtual network* is the encapsulated communication system of a DAS. All communication activities of a virtual network are private to the DAS, i.e. transmissions and receptions of messages can only occur by jobs of the DAS unless a message is explicitly exported or imported by a gateway. Furthermore, a virtual network exhibits predefined temporal properties that are independent from other virtual networks.

A *port* is the access point between a job and the virtual network of the DAS the job belongs to. Depending on the data direction, one can distinguish input ports and output ports. In addition, we classify ports into state ports and

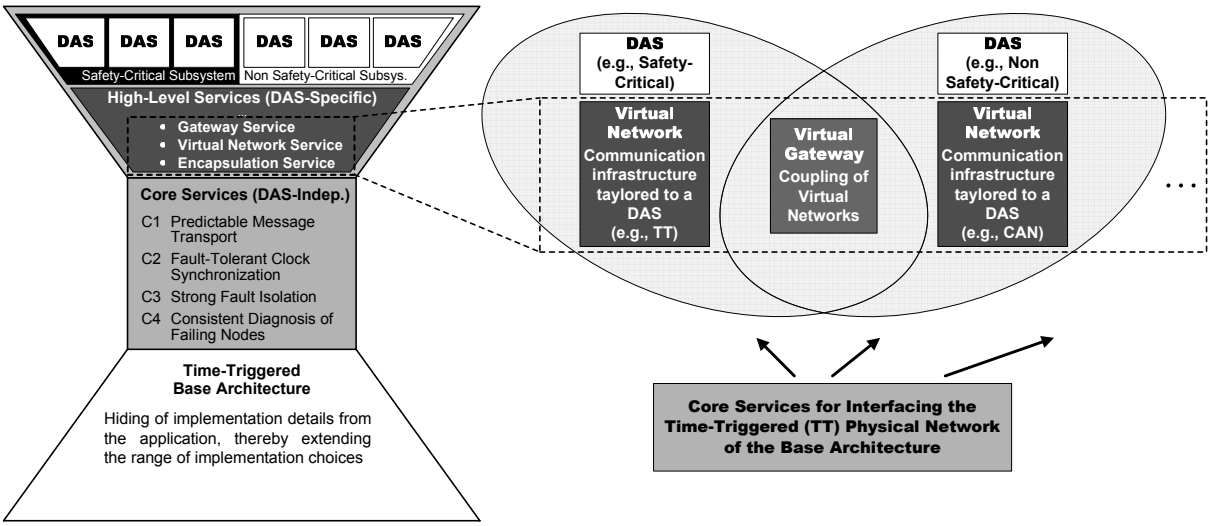

Fig. 1. DECOS Integrated System Architecture with Gateways

event ports depending on the information semantics of send or received message.

A *state port* aims at messages with *state semantics* [8]. Information with state semantics contains the absolute value of a real-time entity (e.g., temperature in the environment is 41 degrees Celsius). Since applications are often only interested in the most recent value of a real-time entity, a state port contains a memory element that is overwritten with newer state values whenever a message arrives at the port (i.e. update in place).

An *event port* supports *event semantics* [8], i.e. information about the change in value of a real-time entity associated with a particular event. Messages containing event information transport relative values (e.g., increase of the temperature in the environment by 2 degrees). In order to reconstruct the current state of a real-time entity from messages with event semantics, messages are queued at an event port in order to process every message exactly-once. The loss of a single message with event information could affect state synchronization between a sender and a receiver.

B. Physical System Structuring

During the development of an integrated system the functional elements must be mapped to the physical building blocks of the platform. In the DECOS architecture these building blocks are the *time-triggered physical core network*, *components* and *partitions*. The components are part of a distributed computer system that is interconnected by the *time-triggered physical core network*. A *component* is a self-contained computational element with its own hardware (processor, memory, communication interface, and interface to the controlled object) and software (application programs, operating system) [9]. Components are the target of job allocation and provide encapsulated execution environments denoted as *partitions* for jobs. Each partition prevents temporal interference (e.g., stealing processor time) and spatial interference [10] (e.g., overwriting data structures) between jobs. In the DECOS architecture, a component can host multiple partitions and host jobs that can belong to different DASs.

C. Architectural Services

Generic architectural services separate the application functionality from the underlying platform technology in order to facilitate reuse and reduce design complexity. This strategy corresponds to the concept of platform-based design [11], which proposes the introduction of abstraction layers, which facilitate refinements into subsequent abstraction layers in the design flow.

The DECOS architectural services depicted in Figure 1 are such an abstraction layer. The specification of the architectural services hides the details of the underlying platform, while providing all information required for ensuring the functional and meta-functional (dependability, timeliness) requirements in the design of a safety-critical real-time application. The architectural services serve as a validated stable baseline that reduces application development efforts and facilitates reuse, because applications build on an architectural service interface that can be established on top of numerous platform technologies.

In order to maximize the number of platforms and applications that can be covered, the DECOS architectural service interface distinguishes a minimal set of *core services* and an open-ended number of *high-level services* that build on top of the core services. The core services include predictable time-triggered message transport, fault tolerant clock synchronization,

strong fault isolation, and consistent diagnosis of failing components through a membership service. The small number of core services eases a thorough validation (e.g., permitting a formal verification), which is crucial for preventing common mode failures as all high-level services and consequently all applications build on the core services. Any architecture that provides these core services can be used as a base architecture [12] for the DECOS integrated distributed architecture. An example of a suitable base architecture is the Time-Triggered Architecture (TTA) [13].

Based on the core services, the DECOS integrated architecture realizes high-level architectural services, which are DAS-specific and constitute the interface for the jobs to the underlying platform. Among the high-level services are gateway services, virtual network services, and encapsulation services. On top of the time-triggered physical network, different kinds of virtual networks are established and each type of virtual network can exhibit multiple instantiations (see Figure 1). Gateway services selectively redirect messages between virtual networks and resolve differences with respect to operational properties and naming. The encapsulation services control the visibility of exchanged messages and ensure spatial and temporal partitioning for virtual networks in order to obtain error containment.

D. Fault Hypothesis

In the integrated system architecture, we perform a differentiation of Fault Containment Regions (FCRs) for hardware and software faults [14]. For hardware faults, we regard a complete component as a FCR, because a component will be implemented as a System-on-a-Chip (SoC) and contains shared physical resources (e.g., processor, power supply). The failure mode of a hardware FCR is assumed to be arbitrary. The failure frequency in case of permanent hardware failures is in the order of 100 FIT [15]. In case of transient failures a significantly higher failure frequency in the order of 1000-10000 hours is assumed.

For software faults, we regard a job as a FCR. The failure mode of a job is a violation of the port specification in either the time or value domain. In case of a failure in the value domain, the content of a message does not conform to its specification, while in case of a timing failure, the send instant of the message is incorrect.

E. Interface Specification

A port is a message interface, where the term *message* denotes a category of frames that are intended for the exchange through the communication system and characterized by common syntactic, temporal and semantic properties. We call a member of such a category of frames, which is sent at a particular point in time, a *message instance*.

We assume that each message is identified by a *message name*, which belongs to a corresponding namespace. In the integrated architecture, each DAS's virtual network possesses such a namespace. While the namespace supports the discrimination of different messages, the namespace does not support to distinguish between message instances. The message name can either by defined via the point in time at which the message is sent (i.e. an implicit message name [8]) or be part of the message content (i.e. an explicit message name).

The syntactic and temporal properties of messages exchanged via a virtual network are captured by an *operational specification* [9], which consists of a syntactic and a temporal specification. The *syntactic specification* forms larger information units (e.g., string, floating point number) out of bits. The *temporal specification* constrains the points in time, the temporal variability (jitter), and the ordering of transmissions and receptions of message instances.

For expressing temporal specifications, we distinguish between time-triggered and event-triggered control [8]. Time-triggered control occurs when the transmission of message instances takes place at predetermined, global points in time and is triggered solely by the progression of time. In contrast, event-triggered control involves on-demand disseminations of message instances at a priori unknown points in time. In general, only probabilistic knowledge about the time of event-triggered message exchanges is specified via interarrival and service time distributions [16], [17]. This knowledge can be available as *local constraints* between message instances within the same message or as *global constraints* between the instances of different messages.

Due to the respective advantages of event-triggered and time-triggered control paradigms [18], the integrated DECOS architecture supports both paradigms through the virtual network services. Time-triggered virtual networks aim at safety-critical DASs, where the benefits with respect to predictability help in managing the complexity of fault-tolerance and corresponding formal dependability models, as required for the establishment of ultra-high reliability [19]. The predetermined points in time of the periodic message transmissions allow error detection and establishing of membership information. Redundancy can be established transparently to applications [20], i.e. without any modification of the function and timing of application systems. A time-triggered system also supports replica determinism [21], which is essential for establishing fault-tolerance through active redundancy.

In non safety-critical (soft real-time) DASs, however,

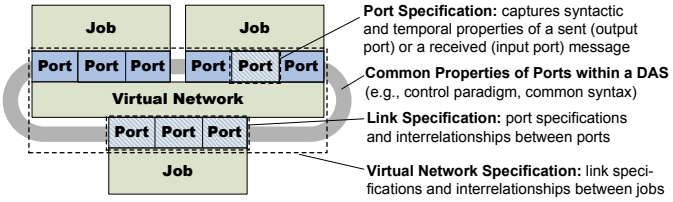

Fig. 2. Jobs of a Distributed Application Subsystem exchange messages via ports.

event-triggered control paradigm may be preferred due to higher flexibility and resource efficiency. Event-triggered virtual networks support dynamic resource allocation strategies and resource sharing. In event-triggered virtual networks the provision of resources can be biased towards average demands, thus allowing timing failures to occur during worst-case scenarios in favor of more cost-effective solutions. If the correlation between the resource usages of different jobs is known, resources can be multiplexed between different jobs while providing probabilistic guarantees for communication latencies and sufficiency of buffering capacities.

The operational specification of a DAS in the integrated DECOS architecture occurs at three levels (see Figure 2):

- **Port Specification:** A port is dedicated to the transmission or reception of message instances of a single message. Each port has a statically defined data direction, thus forming either an input port or an output port. The port specification captures the syntactic and temporal properties of the message instances of the received (in case of an input port) or the sent (in case of an output port) message. Only those temporal properties are part of the port specification, which are defined for the port in isolation, i.e. independently from other ports (i.e. local constraints). An example for temporal properties contained in the port specification are message interarrival or service times in an event-triggered port or a specification of absolute global points in time for a time-triggered port.
  Furthermore, the port specification captures the direction of the control flow relative to the data flow between sender and receiver. In an *information push behavior* [22], data and control flow have the same orientation, i.e. the information transfer occurs via the sender's request. An *information pull behavior* starts an information transfer via the receiver's request, data and control flow have complementary orientations.
  Based on the concepts of information push and pull, we can employ a finer differentiation of interaction types at a port by considering the sender's and receiver's access to the communication system. When the sender passes a message to the communication system, we can distinguish between the case that this process occurs at the request of the sender (denoted as *sender-push*) or through a control signal from the communication system (denoted as *sender-pull*). In analogy, we can distinguish between the case that the communication system passes messages to the receiver at the receiver's request (denoted as *receiver-pull*) or through a control signal from the communication system (denoted as *receiver-push*).
  With this classification of interaction types, we can refine our concept of ports. The notion of an input port is further differentiated into a push input port (receiver-push) and a pull input port (receiver-pull). The notion of an output port is differentiated into a push output port (sender-push) and a pull output port (sender-pull).
- **Link Specification:** The link of a job consists of the ports provided to the job. The link specification contains the respective port specifications and additional temporal properties that can be defined only with respect to multiple ports of the job (i.e. global constraints). An example for the additional temporal properties would be a statement for the latency between the reception of a request message at an input port and the transmission of the corresponding reply message at an output port of the job.
- **Virtual Network Specification:** The virtual network specification consists of all link specifications in the DAS and those temporal properties that can be defined only with respect to ports of more than one job. For example, consider the effect of bandwidth multiplexing between jobs, in which case the communication activities at ports of one job may influence the temporal properties at ports of other jobs. In this case, the virtual network specification expresses the durations of a message transmissions and the variability of these durations (transmission jitter) in the relation to the behavior of the jobs at their output ports.

III. Role of Virtual Gateways

A virtual gateway in the integrated DECOS architecture interconnects two virtual networks of two respective DASs by forwarding information contained in the messages received at the input ports of one virtual network onto the output ports towards the other virtual network (and vice versa in case of a bidirectional gateway). As depicted in Figure 3 a virtual gateway possesses to each of the two interconnected virtual networks a link that consists of a set of ports. The ports are specified by a corresponding port specification that captures the semantic and operational properties of the messages

exchanged via the port. The port specifications are part of the link specification, which also contains properties that can be defined only over multiple ports of the link.

In general, the semantic and operational properties of the input ports at one virtual network can be different to the semantic and operational properties of the output ports at the other virtual network. The resulting *property mismatch* is resolved by the gateway by performing transformations on the information passing through the gateway. A *property mismatch* is a disagreement among connected interfaces in one or more of their properties [23]. Consequently, the first purpose of a gateway is *property transformation*.

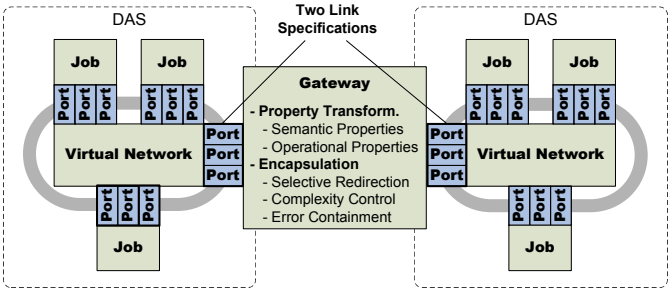

Fig. 3. Purposes of a Gateway

The second purpose of the gateway is *encapsulation*. In general, only a fraction of the information exchanged at one virtual network will be required by jobs connected to the virtual network at the other side of the gateway. By restricting the redirection through the gateway to the information actually required by the jobs of the other DAS, the gateway not only improves resource efficiency by saving bandwidth of unnecessary messages, but also facilitates complexity control. A further aspect of the encapsulation performed by the gateway is error containment, which depends on the gateway's ability for detecting message failures and preventing propagation of faulty messages through the gateway.

In the DECOS integrated architecture, we sharply distinguish between architecture level and application level. Based on this differentiation, we can identify two choices for the construction of a gateway. A *hidden gateway* performs the interconnection of virtual networks at the *architecture level*. Generic architectural services – although parameterized by the application requirements – are transparent to the jobs at the application level. A *visible gateway*, on the other hand, performs the interconnection at the *application level*. A so-called *gateway job* possesses ports to two virtual networks.

In the remainder of the paper, we focus on hidden gateways. The major strength of a hidden gateway is a simplification of the application software. Functionality is moved to the architecture level, where it is developed and validated once and for all. In contrast, a visible gateway enables the designer to resolve mismatches that elude a generic architectural solution. Property mismatches at the semantic level will usually fall into this category.

A. Property Transformation

Since a virtual gateway couples two virtual networks that build on top of the same physical network, property mismatches occur only at the operational and semantic levels.

1) Semantic Level: A semantic mismatch between two virtual networks occurs, if the meaning of messages interpreted by the jobs connected to one virtual network does not align with the meaning intended by the sender jobs connected to the other virtual network. A semantic mismatch impairs the achievement of the intended effect of a message, such as causing the receiver to carry out an action, informing the receiver about a particular fact, or requesting from the receiver to send back specific information [24].

A semantic mismatch is typically resolved with an application-level gateway. In most cases, the highly application-specific nature of a semantic mismatch prevents the construction of generic application services. Nevertheless, we want to highlight event and state semantics, as well as incoherent naming as particular cases of semantic mismatches in port specifications, for which the integrated architecture provides support with hidden gateways. Naming is denoted as incoherent, if different entities are assigned the same name in different parts of a system. The integrated architecture supports incoherent naming in different DASs by providing a separate name space for each DAS. At gateways between DASs, however, this naming incoherence must be resolved. When different entities are assigned the same name in the two DASs, a message carrying this entity has to be renamed. Furthermore, when the same entity possesses different names in the two DASs, then the gateway has to change the message name assigned by the producing DAS to the message name of the consuming DAS.

2) Operational Level: The need for resolving property mismatches at the operational level arises from differences in the operational specifications of ports and links.

The resolving of property mismatches can occur in a generic manner through hidden gateways. For syntactic transformations, the gateway requires a description of the syntactic format (i.e. the data types) of the messages passing through the gateway and rules for transforming the different syntactic transformations into each other. Generic transformation rules are possible due to widely-used standards for data types and interface definition languages (e.g., CORBA IDL [25]) for the construction of

more complex data types that are hierarchical compound from elementary data types, such as integers or floating point numbers.

The transformation of different temporal specifications is simpler, when the interacting DASs exhibit the same operational specification style. For example, if both DASs are time-triggered, then a priori knowledge about the points in time of message transmissions and receptions is available via the port specifications to the gateway. When the interacting DASs operate with different periods or phase-shift relationships of the time-triggered communication schedules, the gateway needs to buffer messages. The forwarding and buffering of messages can be performed according to a schedule that is fixed at design time.

If the DASs interconnected by the gateway exhibit different operational specification styles, the gateway requires additional buffering functionality for the exchange of information between virtual networks with varying rigidity of temporal specifications. For example, such a scenario occurs, if one DAS operates time-triggered and the second DAS operates event-triggered.

B. Encapsulation

In the integrated DECOS architecture, a DAS along with the corresponding communication resources (virtual networks) and computational resources (partitions in components) is encapsulated and interactions between DASs are limited to the exchange of messages via precisely specified hidden gateways.

1) Selective Redirection: By performing a selective redirection of information, the virtual gateway acts as a forwarder [26]. Selective redirection occurs when filtering mechanisms are applied in order to decide on whether information is forwarded or blocked by a gateway. This decision requires a filtering specification in the temporal and value domain that can be evaluated on the interface state [23] of the gateway. In the value domain, the gateway checks message contents with user data and control information such as message names or message types. In the temporal domain, the gateway monitors the temporal patterns of messages exchanges.

2) Complexity Control: The selective forwarding of information with filtering capabilities is a key element to control complexity, i.e. limiting the mental effort required to understand a DAS. In general, only a subset of the information exchanged by jobs is required outside the DAS. The controlled export of information through the virtual gateway improves the understandability of DASs, because for analyzing a DAS only those messages must be considered that are produced by the jobs of the DAS or pass through gateways into the DAS.

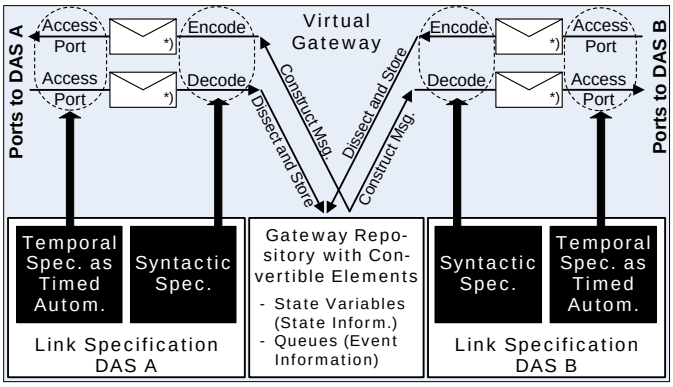

Fig. 4. Operation of a Virtual Gateway

3) Error Containment: Although an FCR can restrict the immediate impact of a fault, fault effects manifested as erroneous data can propagate across FCR boundaries. For this reason virtual gateways must also provide error containment [27] to avoid error propagation by the flow of erroneous messages. A virtual gateway supports error containment, when the selective redirection of information is controlled by error detection mechanisms.

In the DECOS integrated architecture, virtual gateways perform error containment in the temporal domain based on temporal specifications at the port and link level. Section IV will present a solution for virtual gateways that exploits these temporal specifications for blocking timing messages failures.

IV. Realization of Hidden Virtual Gateways

A hidden virtual gateway is an architectural service for the redirection of information between two virtual networks with the necessary property transformations. In this section, we will focus on operational property transformations, resolving of incoherent naming, and the conversion between event and state semantics. The highly application-specific nature of general semantic property transformations usually precludes a generic solution at the architecture level.

The operation of a virtual gateway is illustrated in Figure 4. For each of the two interconnected virtual networks, the virtual gateway possesses an link specification that denotes the knowledge concerning the syntactic and temporal properties of messages. The virtual gateway parses incoming messages based on the link specification of the corresponding DAS. It, then, dissects each message into *convertible elements* and stores these convertible elements in a real-time database denoted as the *gateway repository*. This real-time database with convertible elements comprises the central data structure of the virtual gateway and will be described in the following. Finally, the virtual gateway encodes information from

the gateway repository into outgoing messages according to the syntactic part of the link specification of the second DAS. The temporal part of the link specification determines the control mechanism and the timing for the interaction with the ports to a virtual network.

For each message handled by the gateway, there is a dedicated port and thus a dedicated message buffer for holding message instances of this message. For messages with state semantics exchanged via state ports, the message buffer is a state variable that stores a single message instance only. For messages with event semantics exchanged via event ports, a message buffer is a queue that can hold a statically defined number of message instances to accommodate for temporary intervals of time with imbalances of message interarrival and service times. The determination of the queue sizes is derived from the relationships between message interarrival and service times, e.g., as expressed via a probabilistic model [16].

A. Gateway Repository

In general, only a part of a message will be subject to selective redirection by the gateway. Therefore, the virtual gateway regards a message as a compound structure that contains smaller units of information. A part of a message that needs to be subdivided no further by the virtual gateway is denoted as a *convertible element*. A convertible element is thus an atomic unit of information that is stored in the virtual gateway and packed into a message as a whole. Each message consists of one or more convertible elements.

Redirection of information through the gateway occurs when messages of the two virtual networks that are interconnected by the gateway share common convertible elements. Note, however, that the messages at the two virtual networks need not consist of the exact same set of convertible elements. Although, both a message in virtual network A and a message in virtual network B contain a particular convertible element, the other convertible elements of the two messages can be different from each other.

As a consequence, whenever the virtual gateway is redirecting information from virtual network A to virtual network B, the virtual gateway must first dissect the messages received from virtual network A into convertible elements and recombine these convertible elements into messages for virtual network B.

The virtual gateway buffers convertible elements, because the message exchanges at the two sides of a gateway are temporally decoupled by the virtual gateway and the necessary convertible elements for constructing a particular message might arrive at different points in time. Since convertible elements carry temporally accurate real-time images exchanged between jobs, this buffer forms a real-time database [8]. We denote this real-time database of the virtual gateway as the *gateway repository*.

Information Semantics	Convertible Element Data	Meta Information	
		Variable	Description
Convertible Element with State Semantics		d^1_{acc} b^1_{req} t^1_{update}	temporal accuracy interval update request point in time of last update
Convertible Element with State Semantics		d^2_{acc} b^2_{req} t^2_{update}	temporal accuracy interval update request point in time of last update
⋮			
Convertible Element with Event Semantics		b^{k+1}_{req}	request for convertible element instance
Convertible Element with Event Semantics		b^{k+2}_{req}	request for convertible element instance
⋮			

Fig. 5. Gateway repository contains convertible elements with state semantics and event semantics.

For the storing of convertible elements in the gateway repository, the information semantics of convertible elements are taken into account (see Figure 5). For convertible elements with state semantics, the repository contains state variables that are overwritten whenever a new version of the convertible element arrives (update in place). Convertible elements with event semantics, on the other hand, are stored in queues. Since event semantics represent relative information, every convertible element must be processed exactly-once by a receiver to support state synchronization between a sender job and the receiver jobs.

In addition to the data of the convertible elements, the virtual gateway also stores meta information about convertible elements. For each convertible elements with state semantics, the temporal accuracy interval d_{acc} is a static attribute that determines when the validity of a real-time image is lost through the progression of time. The point in time of the most recent update t_{update} is a dynamic attribute associated with each convertible element with state semantics. The purpose of t_{update} and d_{acc} is to ensure that only temporally accurate real-time images are forwarded by the gateway. A real-time image stored as a convertible element with state semantics in the gateway repository is temporally accurate, if the following conditions holds,

$$t_{\text{update}} + d_{\text{acc}} < t_{\text{now}} \tag{1}$$

where t_{now} is the current point in time. For convertible elements with event semantics, no meta information with respect to temporal accuracy is stored. In order to maintain state synchronization between a sender and its receivers, convertible element instances need to be processed exactly once and independently of whether

the state variable, which is constructed from the value changes in the convertible element, will be temporally accurate after the incorporation of the value change in a particular convertible element instance.

Furthermore, for the interaction with event-triggered virtual networks the meta information of a convertible element includes a boolean request variable b_{req}. For convertible elements with both state and event semantics, the respective boolean request variable denotes whether a convertible element instance needs to be transferred into the gateway repository. By setting the request variable, the gateway side sending messages to an event-triggered virtual network can request convertible element instances from the other virtual network. The gateway side receiving messages from an event-triggered virtual network can initiate receptions conditionally, based on the value of the request variable.

B. Link Specification

In order to perform selective redirection and property transformations of information between two virtual networks, the virtual gateway requires a link specification of the message interfaces to each of the two virtual networks. This specification enables the virtual gateway to perform the following actions:

- **Port interactions:** interact with a virtual network to send and receive messages.
- **Construction and dissection of messages:** divide a message into its convertible elements and construct a message out of convertible elements.
- **Syntax transformation:** perform a recombination of convertible elements received via messages from one virtual network into the syntactic structure of messages for the second virtual network.
- **Conversion of information semantics:** transform convertible elements with event semantics into convertible elements with state semantics and vice versa.

The link specification consists of three parts, namely a syntactic part, a temporal part, and transfer semantics. The *syntactic part* provides a separate syntactic description for each message on the virtual network that is to be sent or received by the gateway. The *temporal part* is a set of deterministic timed automata that express the protocol for interacting with the ports to a particular virtual network. The automata specify the control patterns (e.g., request-reply interactions), the sequence of message exchanges, and the temporal constraints for the transmission and reception of messages (e.g., message interarrival times). The *transfer semantics* specify the information semantics of convertible elements and provides rules for the conversion of convertible elements between state and event semantics.

Since a virtual gateway interconnects two virtual networks, a virtual gateway is always equipped with two link specifications. The two link specifications exhibit a common set of common convertible elements that are part of messages in both network segments. Each such convertible element is produced by jobs at one virtual network and consumed by one or more jobs at the second one.

```
<linkspec>
<DAS>X-by-wire</DAS>
<message name="msgSlidingRoof">
<element name="Name" key="yes" conv="no">
 <field name="id">
  <type length=16>Integer</type>
  <value>731</value>
 </field>
</element>
<element name="MovementEvent" key="no" conv="yes">
 <field name="ValueChange"><type length=16>Integer</type></field>
 <field name="EventTime"><type length=16>Timestamp</type></field>
</element>
<element name="FullClosure" key="no" conv="no">
 <field name="Trigger"><type>Boolean</type></field>
</element>
</message>

<timedautomaton name="msgSlidingRoofReception">
  <location name="statePassive"/>
  <location name="stateActive"/>
  <location name="stateError"/>
  <init name="statePassive"/> <error name="stateError"/>
  <transition>
    <source name="statePassive"/><target name="stateActive"/>
    <label type="guard">x&gt;=Tmin</label></transition>
  <transition>
    <source name="stateActive"/><target name="statePassive"/>
    <label type="guard">x&lt;Tmax m?</label>
    <label type="assignment">x:=0</label>
   </transition>
  <transition>
    <source name="stateActive"/><target name="stateError"/>
    <label type="guard">x&gt;=Tmax</label>
  </transition>
  <transition>
    <source name="statePassive"/><target name="stateError"/>
    <label type="guard">m?</label>
  </transition>
  <transition>
    <source name="statePassive"/><target name="statePassive"/>
    <label type="guard">x&lt;Tmin , ~m?</label>
  </transition>
  <transition>
    <source name="stateActive"/><target name="stateActive"/>
    <label type="guard">x&lt;Tmax , ~m?</label>
  </transition>
</timedautomaton>

<transfersemantics>
<element name="MovementState">
 <field name="StateValue" init=0 semantics="State">
    StateValue=StateValue+ValueChange
 </field>
 <field name="ObservationTime" semantics="State">
    ObservationTime=EventTime
 </field>
</element>
</transfersemantics>
</linkspec>
```

Fig. 6. XML example of a link specification with a syntactic part (structure of each message), a temporal part (deterministic timed automata), and transfer semantics.

We have chosen Extensible Markup Language (XML) [28] for expressing link specifications, because of the wide use of XML and the availability of parsers. The XML syntax for an exemplary link specification in an automotive application can be found in Figure 6. The syntactic part of the link specification defines the structure of a message `msgSlidingRoof`, which contains event information about the movement of a car's sliding

roof. The field `ValueChange` denotes the change of the opening of the sliding roof in percent that has occurred at time `EventTime`. In addition, the element `FullClosure` informs whether the user has issued the command for a full closure of the sliding roof.

We assume that the movement of the sliding roof needs to be forwarded to a second virtual network by the gateway, e.g., since the data provided by the comfort DAS shall be display by instruments that are accessed by a second DAS. Consequently, the element `MovementEvent` is marked as a convertible element. We also assume that the second DAS expects state semantics, thus requiring a conversion of the event semantics of `ValueChange` into state semantics. This conversion is specified via a convertible element `MovementState` that is computed out of `MovementEvent`.

The temporal part of the link specification specifies a deterministic timed automaton supporting the push reception of the message `msgSlidingRoof` with predefined interarrival and service times.

1) Syntactic Part: The *syntactic part* for a message denotes the message structure. It defines the *elements* of a message with each element being a structure of further elements or fields. A *field* is a variable that possesses a known type (e.g., integer, string, floating point number) and is subdivided no further, i.e. it is considered as atomic at the virtual gateway. A field is *static*, if the value of field is time-invariant, while the value of a *dynamic field* can change over time.

For each element, the link specification specifies whether the element is a convertible element. Convertible elements are subject to selective redirection by the gateway and stored in the gateway repository. All other elements are only of local interest to a virtual network and discarded by the virtual gateway.

Each message requires a *message name* that provides an identification by which the message is demarcated from other messages on the virtual network. The message name is statically defined subset of a message's elements. The message name consists of static fields – not necessarily at the beginning of a message - and is defined via the syntactic part of the link specification. A message name is unique for each message of a virtual network and allows the virtual gateway to determine for each message instance the corresponding message entry in the link specification.

2) Temporal Part: The temporal part is specified by a deterministic timed automaton [29], i.e. a state transition graph annotated with timing constraints. We employ a notation similar to the syntax introduced in [30]. We introduce a special state denoted as "error state". The reaching of this state represents a violation of the temporal specification and gives the gateway the ability to perform error handling (e.g., restart of the gateway service).

In addition to guard and assignment labels, we provide labels for port interactions in order to exchange messages between virtual networks and the gateway repository:

- **Guard.** A guard expresses a condition on clock and state variables that must be fulfilled so an edge can be taken. A guard is a conjunction of expressions of the form $x \circ y$, where $\circ$ is a binary operator (e.g., $\leq, =$). x and y are clock variables or state variables.
- **Assignment.** An assignment ($x := n$) permits to set the values of a clock or state variable x to a new value n.
- **Port interactions.** The sending of a message m at an output port is expressed as $m!$, while $m?$ denotes the receiving of the message at an input port. An edge with a label $m!$ for a transmission represents a guard. The edge can only be taken, if all convertible elements for the construction of the message are available in the repository. A convertible element with state semantics needs to be temporal accurate (see Condition (1)), while a convertible element with event semantics must possess a non empty queue. If the convertible elements are unavailable, the boolean request variables b_{req} of the missing convertible elements are set. In case the edge is actually taken, the gateway constructs the message m out of convertible elements from the repository and transfers the message m to the output port. Similarly, the reception of a message $m?$ also functions as a guard to enable conditional state changes. An edge with this label can only be taken, if an incoming message is present at the respective input port. If the edge is taken, the message m is dissected into convertible elements that are transferred to the repository.

In addition to the clock and state variables defined for a particular timed automaton, the clock variable t_{now} captures the current point in time. Furthermore, two functions *horizon(m)* and *requ(m)* can be used in guards and assignments.

For a message m with state semantics, the *temporal accuracy horizon*, which is formally denoted as *horizon(m)*, yields the length of the remaining interval of time during which all constituting convertible elements of message m are temporally accurate. For this purpose, the gateway takes into account the temporal accuracy intervals and the points in time of the most recent updates of the constituting convertible elements. If message m consists of a set of convertible elements $C_m = \{c_1, \ldots, c_k\}$, then *horizon*$(m)$ is defined as

$$\textit{horizon}(m) = \min_{c \in C_m} \left(t^c_{\text{update}} + d^c_{\text{acc}} - t_{\text{now}}\right) \qquad (2)$$

By computing the temporal accuracy horizon of a message m as the minimum of the durations of remaining temporal accuracy of convertible elements, it is ensured that all real-time images transported in the message m are temporally accurate until $t_{\text{now}} + \textit{horizon(m)}$. For the gateway side sending the message m to a virtual network, the *horizon(m)* must be at least as large as the remaining time (e.g., communication delay, operating system delays at the receiver) until the point in time of use of the real-time image transported via m. By exploiting *horizon(m)*, the timed automaton can be used for the detection of conditions in which a real-time image would have been invalidated by the progression of time when it is used by a receiver job. Furthermore, in combination with event-triggered control the temporal accuracy horizon allows to determine the need for requesting messages in order to perform an update of the gateway repository.

The function *requ(m)* yields true, if there is a constituting element of message m for which an update has been requested.

$$\textit{requ}(m) = \begin{cases} \text{true} & \exists c \in C_m \text{ with } b^c_{req} = \text{true} \\ \text{fase} & \text{otherwise} \end{cases} \quad (3)$$

This function allows to construct timed automata for event-triggered message receptions according to the information pull principle. For this purpose, messages are requested via the virtual network conditionally, namely based on the result of the function *requ(m)*.

In the following, we will explain how to use these automata for expressing temporal specification for input and output ports employing time-triggered control or event-triggered control with either the information push or information pull principle. These examples are meant to illustrate the capabilities of generic gateway services with temporal specifications provided by deterministic timed automata.

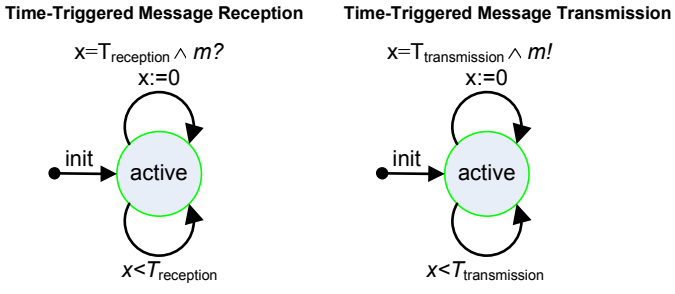

Fig. 7. Deterministic timed automata for time-triggered message reception and transmission.

Deterministic timed automata for time-triggered message transmissions and receptions are depicted in Figure 7. The automata contain edges with labels for port interactions ($m?$ and $m!$) that can be periodically taken with a period $T_{\text{reception}}$ or $T_{\text{transmission}}$. The automata do not contain error states, because a proper time-triggered communication schedule for the respective virtual network ensures temporal correctness by design.

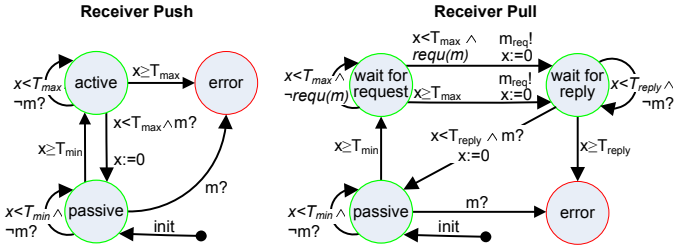

Fig. 8. Deterministic timed automata for event-triggered message receptions with interarrival times $[T_{\min}, T_{\max}]$.

Figure 8 contains exemplary timed automata for a receiver-pull and a receiver-push activity. The timed automata specify event-triggered sporadic message receptions with an upper bound on the interarrival time $T_{\max}$ and a lower bound on the interarrival time $T_{\min}$.

In the push scenario in Figure 8, the initial state is the "passive state", which enforces the minimum interarrival time $T_{\min}$. Any message reception within the passive state enables a transition to the "error state". After the minimum interarrival time has passed, the automaton progresses into the "active state". When a message arrives before the maximum interarrival time has passed, the automaton transits back to "passive state". If a message fails to arrive, the automaton progresses into the "error state" after the maximum interarrival time.

In contrast to the push scenario, in the pull scenario in Figure 8 a message m only arrives after a request message m_{req} has been sent. The corresponding automaton also contains a "passive state", which enforces the minimum interarrival time $T_{\min}$. Since any reception of a message m violates the temporal specification and does not belong to a request message, a transition to the "error state" is taken. After $T_{\min}$ has passed, the automaton remains in a state "wait for request" until the gateway repository signals the need for a more recent message instance via *requ(m)* or a request needs to be sent to avert a violation of the maximum message interarrival time. The next state "wait for reply" is left after the reply message m belonging to m_{req} is received. In case this reply fails to arrive within a timeout T_{reply}, a transition to the "error state" occurs.

Figure 9 depicts exemplary timed automata for event-triggered message transmissions. The initial state of the automaton for the sender-push activity is the "passive state". A transition to the "active state" can only be taken after the minimum interarrival time $T_{\min}$ has elapsed. From the "active state", the return to "passive state" occurs as soon as the transmission guard $m!$ fires. In

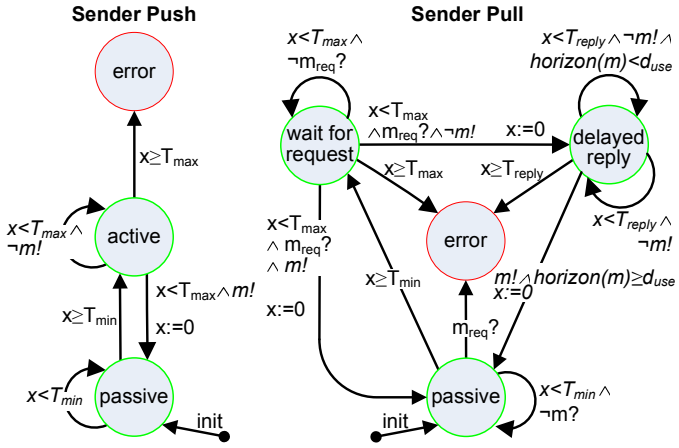

Fig. 9. Deterministic timed automata for event-triggered message transmission with interarrival times $[T_{\min}, T_{\max}]$.

case this guard blocks for the maximum interarrival time $T_{\max}$, a transition into the "error state" takes place.

The automaton for the sender-pull activity in Figure 9 accepts a request message from the virtual network and responds with a reply message as soon as the convertible elements for the construction of the reply message are available, i.e. guard $m!$ fires. The temporal properties that are checked include minimum and maximum interarrival times of request messages, and the maximum response time T_{reply} for reply messages. A violation of these properties causes a transition into the "error state". In addition, the transmission of message m only occurs, if the temporal accuracy horizon of m is sufficient to ensure the temporal accuracy of the transported real-time images at the point in time of use ($\mathit{horizon}(m) < d_{\text{use}}$). The duration d_{use} incorporates all delays, such as the communication system delay and the operating system delay at the receiver, before the real-time image is actually used.

The presented examples have demonstrated how timed automata are used by the virtual gateways to ensure error containment in the temporal domain. Error containment in the value domain is the responsibility of the application (e.g., through replication and comparing redundant computations). The timed automata encode the temporal specification of a link to a virtual network. Hence, these automata are constructed in such a way that all execution paths represent correct behaviors. In case the execution on such a path would not possible, e.g., since temporally accurate real-time convertible elements for the construction of a message are unavailable, a transition to the "error state" is performed. The "error state" permits a reaction to a detected error, such as storing of contextual information about the error for analysis, or a restart of the timed automaton.

3) Transfer Semantic: The transfer semantic defines a conversion for the information semantics of a convertible element. For this purpose, the transfer semantic provides an expression for computing either event information from successive state information values or state information from a sequence of event information values. The respective convertible element is equipped with two message buffers, both a state variable and a queue.

For the conversion from event to state information, the expression of the transfer semantic defines an accumulation of event related value changes. An example for such an expression is contained in Figure 6, in which information about the movement of the sliding roof is converted from event to state semantics. The computed state semantics value leads to an update in place of the convertible element's state variable.

The conversion from state to event semantics, on the other hand, occurs via an expression that derives an event related state change from the most recent and the previous value of a state variable. In case these two state values differ, the event related state change is stored in the convertible element's queue.

C. Invocation and Activities of the Virtual Gateway

The virtual gateway services are implicitly synchronized with the virtual network services. The memory buffers associated with the ports that comprise the interface between the virtual network and gateway services are accessed alternately by these two services, controlled by the progression of time on a global sparse time base [31]. In the sparse time model the continuum of time is partitioned into an infinite sequence of alternating durations of activity and silence. Thereby, the occurrence of significant events is restricted to the activity intervals of a globally synchronized action lattice. The sparse time base allows to generate a consistent temporal order on the basis of time-stamps [8]. During the silence intervals of the action lattice the sparse time base provides a consistent distributed state, where the notation of state is used as introduced in system theory [32, p. 45] as a dividing line between past and future.

Every port of the gateway is associated with a corresponding port state, which is the state of the gateway as seen from the port. The notation of *port state* is based on the concept of interface state [23]. For a state port, the port state comprises a state variable, while a message queue is the port state of an event port. As depicted in Figure 10, the update of the port state through the virtual network services occurs during the activity intervals of the global sparse time base. A message reception from a virtual network results in the update the state variable associated with a state input port or the insertion of an event message into a message queue of an event input port. At an event output port, message transmissions via a virtual network results in the removal of message from

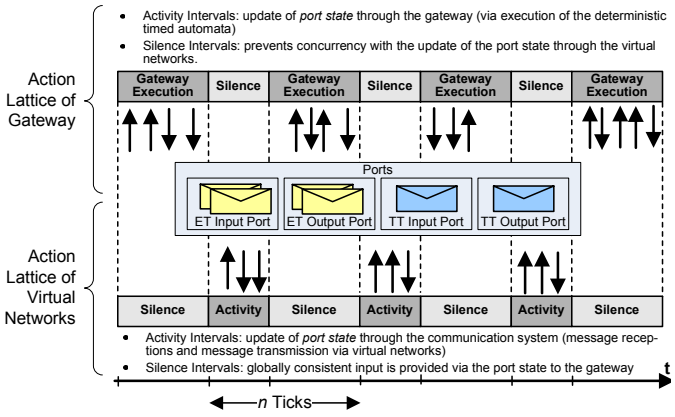

Fig. 10. Execution of timed automata occurs on a global sparse time base

an outgoing message queue. In the silence intervals of the sparse time base, globally consistent input is provided via the port state to the gateway.

In these silence intervals of the virtual network services, the gateway service accesses the ports via the execution of the deterministic timed automata. Thereby, each automaton can start its execution with a globally consistent port state. At each activation, the automata associated with the two interacting DASs are executed. As depicted in Figure 11, the labels associated with the transitions of the automata cause interactions both with ports and the gateway repository.

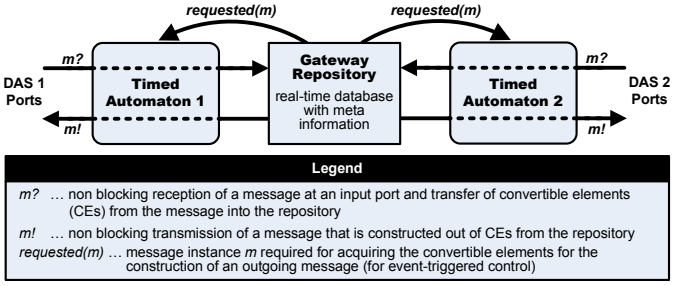

Fig. 11. Gateway Operation. *Timed automata control the flow of information (indicated through arrows) to and from the repository.*

Upon each activation time progresses by n ticks, which is reflected by all clock variables maintained in automata. For each automaton, the gateway proceeds with evaluating guards and performing state changes with edges for which all guards are being fulfilled. In case no edge can be taken with the current values of the variables (including clock variables), the execution of the respective automaton is finished for this activation cycle. During the execution of the automata, the reading and updating of the port state is triggered through the execution of transitions with port interaction labels. Whenever an edge with a port interaction label $m!$ is taken, a message instance is constructed out of convertible elements from the repository and written into the output port. Similarly, the taking of an edge with an interaction label $m?$ causes the reading of a message instance from the respective input port and the dissection into convertible elements that are transferred into the repository.

During the activity interval of the virtual network services, on the other hand, the gateway services are inactive in order to prevent concurrent updates of the port state between the gateway and virtual network services.

In addition to the implicit synchronization between the virtual network and gateway services, the execution of services on a global sparse time base allows the construction of replica determinate [21] gateways as required for building a fault-tolerant system with active redundancy. By providing each automaton with a globally consistent port state as input, gateway functionality can be provided redundantly at different components with replica determinism.

V. Application

This section describes the application of virtual gateways in a real-world automotive example. We elaborate on a typical automotive electronic infrastructure and map these federated clusters onto the introduced integrated architecture. Thereby the need for a coupling of individual DASs emerges. The virtual gateways establish means that allow for the interconnection of DASs with different requirements in the integrated architecture. Based on this automotive example, we highlight the various ways of utilizing gateways to integrate legacy subsystems into the integrated architecture and to allow for improved quality of control and resource efficiency.

To give an impression of the complexity and the amount of electronics in today's luxury cars take for example the Volkswagen Phaeton or the BMW 7 series cars. The Phaeton can have up to 61 Electronic Control Units (ECUs), the BMW 7 up to 75 ECUs depending on the customers requested extra equipment [33], [34]. The distributed ECUs are interconnected via communication networks with different protocols (e.g., Controller Area Network (CAN) [35], Media Orientated System Transport (MOST) [36]), physical layers, bandwidths (10 kbps–500 kbps), and dependability requirements.

Such a typical automotive network is depicted in Figure 12. Multiple federated clusters are connected via a central gateway allowing data exchange and access to the On-Board Diagnosis (OBD) systems of each ECU. The comfort clusters as well as the powertrain cluster are typically implemented via the CAN protocol, while the multimedia cluster in luxury cars is frequently based on the MOST protocol.

Each of these clusters consists of nodes that are dedicated to a single job. In order to improve the functionality

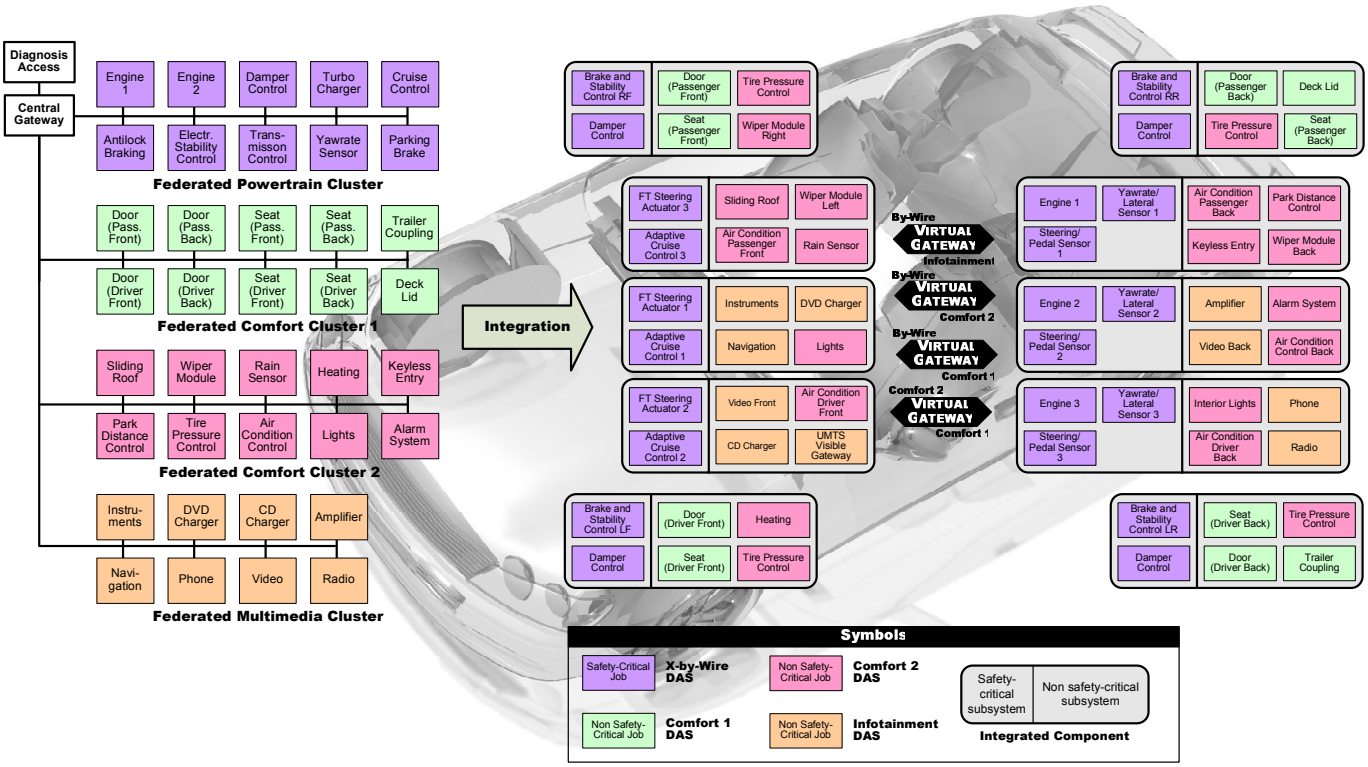

Fig. 12. Typical Network of State-of-the-Art Car as Federated and Integrated System

of the car, additional ECUs have to be added to the clusters. However, this trend of increasing the number of ECUs is coming to its limits, because of complexity, wiring, space and weight restrictions.

A mapping of the above described electronic infrastructure onto the DECOS integrated architecture is shown in Figure 12. In order to emphasize the suitability of the proposed DECOS architecture for safety-critical applications, we exchanged the powertrain DAS with a drive-by-wire DAS.

As depicted in Figure 12 several virtual gateways are employed to allow the controlled export and import of information between the different DASs. For example, a virtual gateway between the by-wire DAS and the infotainment DAS allows to forward data from the yaw rate sensors to the navigation system (as proposed in [6]), thus reducing sensor duplication. Furthermore, the virtual gateway between the by-wire DAS and the comfort DAS permits to improve quality-of-control, e.g., by tensioning seat-belts and realigns seats as defined in [5].

VI. Conclusion

Virtual gateways in the integrated DECOS architecture permit the exchange of information between Distributed Application Subsystems (DASs), while preserving encapsulation. Virtual gateways are are a generic architectural service that is parameterized with an message description that consists of a syntactic and a temporal specification as a timed automaton. The messages description determines the messages or parts of messages that are forwarded by the gateway. This selective redirection of information is crucial for controlling the mental effort for understanding a DAS, because it allows the hiding of the internal details of a DAS from other DASs. The messages description also enables the virtual gateway to resolve mismatches of operational specifications, resolve incoherent naming and perform transformations between event and state semantics. We have demonstrated the ability to capture important classes of time-triggered and event-triggered transmission and reception behaviors. In addition, the paper has demonstrated how the timed automata can be employed to detect violations of temporal specifications in order to prevent error propagation via timing message failures.

VII. Future Work

The solution presented in this paper supports different message formats through the recombination of convertible elements. As future work we plan to address different syntactic representations of convertible elements in the interconnected virtual networks. Within the messages transported at the two virtual networks interconnected by the gateway, convertible elements may be encoded in different syntactic representations. Therefore, the virtual

gateway should recode convertible elements from the syntactic specification of the source virtual network to the syntactic specification of the destination virtual network.

ACKNOWLEDGMENTS

This work has been supported in part by the European IST project ARTIST2 under project No. IST-004527 and the European IST project DECOS under project No. IST-511764.

REFERENCES

[1] R. Hammett. Flight-critical distributed systems: design considerations [avionics]. *IEEE Aerospace and Electronic Systems Magazine*, 18(6):30–36, June 2003.

[2] H. Kopetz, R. Obermaisser, P. Peti, and N. Suri. From a federated to an integrated architecture for dependable embedded real-time systems. Technical Report 22, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2004.

[3] R. Obermaisser, P. Peti, and H. Kopetz. Virtual networks in an integrated time-triggered architecture. Research report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2004.

[4] L.E. Wallace. *Airborne Trailblazer*. National Aeronautics and Space Administration, NASA History Office, Washington, D.C., 1994.

[5] S. Birch. Pre-safe headlines S-Class revisions. *Automotive Engineering International*, pages 15–18, January 2003.

[6] C.R. Carlson and J.C. Gerdes. Practical position and yaw rate estimation with GPS and differential wheelspeeds. In *Proceedings of 6th International Symposium on Advanced Vehicle Control*, pages 481–486, Hiroshima Japan, September 2002.

[7] Wilfried Elmenreich. *Sensor Fusion in Time-Triggered Systems*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2002.

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

[9] H. Kopetz and N. Suri. Compositional design of RT systems: A conceptual basis for specification of linking interfaces. In *Proceedings of Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 51–60, May 2003.

[10] J. Rushby. Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999. Also to be issued by the FAA.

[11] A. Sangiovanni-Vincentelli. Defining platform-based design. *EEDesign of EETimes*, February 2002.

[12] J. Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, Computer Science Laboratory, SRI International, September 2001.

[13] H. Kopetz and G. Bauer. The time-triggered architecture. *IEEE Special Issue on Modeling and Design of Embedded Software*, January 2003.

[14] A. Avizienis, J.C. Laprie, and B. Randell. Fundamental concepts of dependability. Research Report 01-145, LAAS-CNRS, Toulouse, France, April 2001.

[15] B. Pauli, A. Meyna, and P. Heitmann. Reliability of electronic components and control units in motor vehicle applications. In *VDI Berichte 1415, Electronic Systems for Vehicles*, pages 1009–1024. Verein Deutscher Ingenieure, 1998.

[16] L. Kleinrock. *Queuing Systems Volume I: Theory*. John Wiley and Sons, New York, 1975.

[17] J.P. Lehoczky. Real-time queueing theory. In *Proceedings of 17th IEEE Real-Time Systems Symposium*, pages 186–195, December 1996.

[18] R. Obermaisser. *Event-Triggered and Time-Triggered Control Paradigms – An Integrated Architecture*. Real-Time Systems Series. Kluwer Academic Publishers, November 2004.

[19] N. Suri, C.J. Walter, and M.M. Hugue. *Advances In Ultra-Dependable Distributed Systems*, chapter 1. IEEE Computer Society Press, 10662 Los Vaqueros Circle, P.O. Box 3014, Los Alamitos, CA 90720-1264, 1995.

[20] G. Bauer and H. Kopetz. Transparent redundancy in the time-triggered architecture. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000), NY, USA*, pages 5–13, June 2000.

[21] S. Poledna. *Fault-Tolerant Real-Time Systems: The Problem of Replica Determinism*. Kluwer Academic Publishers, November 1995.

[22] R. DeLine. *Resolving Packaging Mismatch*. PhD thesis, Carnegie Mellon University, Computer Science Department, Pittsburgh, June 1999.

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

[24] C. Linn. Semantic reliability in distributed systems: ontology issues and system engineering. In *Proceedings of IEEE/WIC International Conference on Web Intelligence*, pages 292–300, October 2003.

[25] Object Management Group. *The Common Object Request Broker: Architecture and Specification*, July 2002.

[26] D.S. Dodge. Gateways - 101. In *Proceedings of the Military Communications Conference*, volume 1, pages 532–538. IEEE, October 2001.

[27] J.H. Lala and R.E. Harper. Architectural principles for safety-critical real-time applications. *Proceedings of the IEEE*, 82:25–40, January 1994.

[28] World Wide Web Consortium (W3C). *Extensible Markup Language (XML) 1.0 (Second Edition)*, October 2000. Editors: T. Bray, J. Paoli, C.M. Sperberg-McQueen, and E. Maler.

[29] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.

[30] K.G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a nutshell. *International Journal on Software Tools for Technology Transfera*, pages 134–152, October 1997. Springer-Verlag.

[31] H. Kopetz. Sparse time versus dense time in distributed real-time systems. In *Proceedings of 12th International Conference on Distributed Computing Systems*, Japan, June 1992.

[32] M. D. Mesarovic and Y. Takahara. *Abstract Systems Theory*, chapter 3. Springer-Verlag, 1989.

[33] The Hansen Report on Automotive Electronics, November 2002. Portsmouth NH USA, `www.hansenreport.com`.

[34] A. Deicke. The electrical/electronic diagnostic concept of the new 7 series. In *Convergence International Congress & Exposition On Transportation Electronics*, Detroit, MI, USA, October 2002. SAE.

[35] Robert Bosch Gmbh, Stuttgart, Germany. *CAN Specification, Version 2.0*, 1991.

[36] MOST Cooperation, Karlsruhe, Germany. *MOST Specification Version 2.2*, November 2002.