Specification and Execution of Gateways in Integrated Architectures

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

Abstract

The DECOS integrated 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. 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 a 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 an interface specification based on timed automata and use this description as a parameterization of generic architectural gateway services.

1. Introduction

The DECOS integrated architecture [10] is a system architecture that aims at combining the benefits of integrated and federated architectures. The DECOS 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* [15], 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, the DECOS architecture is designed to provide error containment by preventing the propagation of message failure between DASs [10].

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 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. 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 [3].

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 [5]. 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.

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 [10], 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 resolve operational differences and incoherent naming of interconnected DASs based on a formal description of the properties of each DAS.

This work has been supported by the European IST project DECOS under project No. IST-511764.

The specification of a virtual gateway defines the redirection of messages between two virtual networks along with the necessary property transformations (e.g., syntactic, temporal, message names). The gateway specification comprises interface specifications of the interfaces to the two interconnected virtual networks. Based on the service specification model of linking interfaces [**?**], we distinguish between an operational specification (syntax, temporal properties, interface state) and a semantic specification of an interface.

For the temporal specification, we employ an extension of deterministic timed automata [2] with corresponding execution semantics. Thereby, timed automata can form the input to automatic code generation and be executed as part of an architectural service, which offers a generic framework for the realization of virtual gateways. The interface state specification comprises the definition of the *gateway repository*, which is a real-time database that is maintained by the gateway and contains state variables and event queues updated by the contents of received messages. The gateway repository decouples the two sides of the gateway.

The paper is structured as follows. Section 2 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 3. After describing the operation of a virtual gateway in Section 4, we explain the specification of a virtual gateway in Section 5. The execution of the architectural gateway services is the focus of Section 6. Finally, we discuss the ability of virtual gateways to resolve property mismatches and perform selective redirection in Section 7.

2. DECOS Integrated Architecture

The DECOS architecture [10] 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. The DECOS architecture aims at offering to system designers generic architectural services, which provide a validated stable baseline for the development of applications.

2.1. 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* [15] 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 event ports depending on the information semantics [9] of send or received message.

2.2. Architectural Services

The DECOS architecture distinguishes a minimal set of *core services* and an open-ended number of *high-level services*. The core services include predictable time-triggered message transport, clock synchronization, strong fault isolation, and a membership service. 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. 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.

2.3. 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 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. The message name can either by defined via the point in time at which the message is sent (i.e. an implicit message name [9]) 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* [**?**], 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 [9]. Time-triggered control occurs when the transmission of message instances takes place at predetermined, global points in 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 [7, 12]. Due to the respective advantages of event-triggered and time-triggered control paradigms [**?**], the integrated DECOS architecture supports both paradigms through the virtual network services.

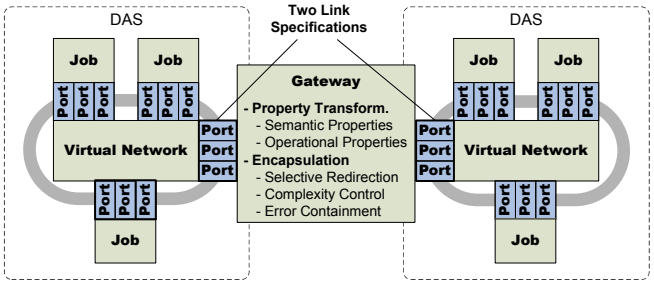


Figure 1. Purposes of a Gateway

The operational specification of a DAS in the integrated DECOS architecture occurs at the following levels:

- **Port Specification:** 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. 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.

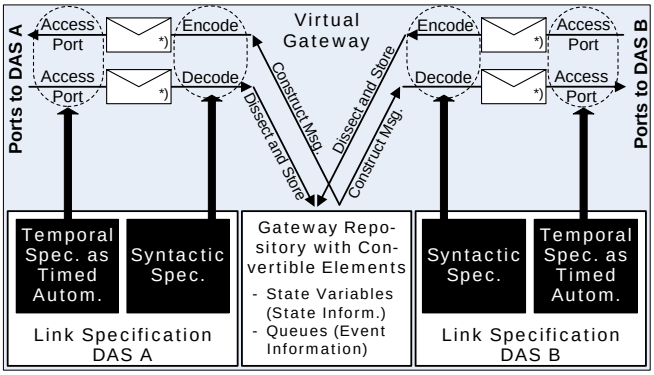
  Furthermore, the port specification captures the direction of the control flow relative to the data flow between sender and receiver. In an *information push behavior* [6], the information transfer occurs via the sender's request. An *information pull behavior* starts an information transfer via the receiver's request.

- **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. 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.

3. Role of Virtual Gateways

A virtual gateway in the integrated DECOS architecture interconnects two virtual networks [15] 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. As depicted in Figure 1, 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 a disagreement among connected interfaces in one or more of their properties [4]. Consequently, the first purpose of a gateway is *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.



*) message buffer: queue for event information, state variable for state information

Figure 2. Operation of a Virtual Gateway

This paper focuses on resolving property mismatches at the operational level, which arise from differences in the operational specifications of ports and links. For syntactic transformations, the gateway employs 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 (see Section 5).

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.

4. 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. We 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 2. For each of the two interconnected virtual networks, the virtual gateway possesses a link specification that denotes the knowledge concerning the syntactic and temporal properties of messages. The virtual gateway sends and receives messages at a virtual network according to the temporal part of the link specification. In addition, the virtual gateway parses received messages based on the syntactic part of link specification of the corresponding DAS.

In general, only a part of a message will be subject to selective redirection by the gateway. Therefore, the 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 gateway is denoted as a *convertible element*. A convertible element is thus an atomic unit of information that is stored in the 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 networks that are interconnected by the gateway share common convertible elements. In this case, the virtual gateway recombines convertible elements received from one virtual network into outgoing messages according to the syntactic part of the link specification of the second virtual network and sends these messages based on the protocol defined by the temporal part of the link specification of the second virtual network.

For this purpose, the gateway dissects each message into *convertible elements* and stores these convertible elements in a central data structure maintained in the gateway. Since convertible elements carry temporally accurate real-time images exchanged between jobs, this data structure forms a real-time database [9]. We denote this real-time database as the *gateway repository*. The gateway repository temporally decouples the two sides of the gateway and allows the convertible elements that are necessary for constructing a particular message to arrive at different points in time.

For storing convertible elements in the gateway repository, the information semantics [9] of convertible elements are taken into account. 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: the temporal accuracy interval d^k_{acc}, the point in time of the most recent update t^k_{update}, a request variable b_{req}, and the number of enqueued messages n_k (1 for a convertible element with state semantics).

For each convertible element c_k with state semantics, the temporal accuracy interval d^k_{acc} is a static attribute that is derived from the dynamics of the controlled object [9]. The temporal accuracy interval 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^k_{update} is a dynamic attribute associated with each convertible element c_k with state semantics. The purpose of t^k_{update} and d^k_{acc} is to ensure that only temporally accurate real-time images are forwarded by the gateway. A real-time image stored as convertible element c_k with state semantics in the gateway repository is temporally accurate, if the following conditions holds,

$$t^k_{\text{update}} + d^k_{\text{acc}} < t_{\text{now}} \tag{1}$$

where t_{now} is the current point in time. For convertible elements with event semantics, the temporal accuracy interval is ∞, because each convertible element instance needs to be processed exactly once in order to maintain state synchronization between a sender and its receivers. Convertible element instances need to be processed 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.

In addition to the meta information about the convertible elements in the gateway repository, the gateway maintains meta information for each message m at an input port. In analogy to convertible elements, d^m_{acc} is the temporal accuracy interval of message m and t^m_{update} is the point in time of the most recent update. This informations enables the gateway to determine whether a message in an input port is temporally accurate at a particular point time.

5. Gateway 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:** interaction with a virtual network to send and receive messages.
- **Construction and dissection of messages:** division of a message into its convertible elements and construction of a message out of convertible elements.
- **Syntax transformation:** 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:** transformation of convertible elements with event semantics into convertible elements with state semantics and vice versa.

The link specification consists of three parts, namely a *syntactic specification*, a *temporal specification*, and *transfer semantics*. The syntactic specification defines the structure of the messages that are sent or received by the gateway. In particular, the syntactic specification defines for each message the constituting convertible elements that need to be stored in the gateway repository. The temporal specification is a set of deterministic timed automata that express the protocol for

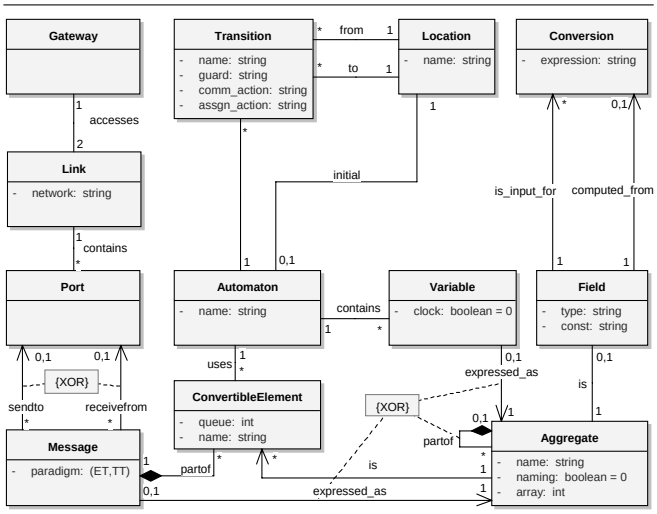


Figure 3. Gateway Specification

interacting with the ports to a particular 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 provide rules for the conversion of convertible elements between state and event semantics.

Figure 3 shows a UML class diagram for the specification of a gateway. A Gateway accesses two Links to the networks that are interconnected by the gateway. At each of these Links the gateway sends and/or receives Messages according to a protocol specified by deterministic timed automata. In the following, we will explain the specification of the structure of Messages and Variables (syntactic specification), the specification of the protocol for the transmission and reception of messages (temporal specification), and the conversion between event and state semantics (transfer semantics).

5.1. Temporal Specification

The temporal specification is performed with deterministic timed automata [2], i.e. state transition graphs annotated with timing constraints. In the UML model in Figure 3, the classes Transition and Location are used for defining deterministic timed gateway automata. The vertices of the graph are called *locations* and interconnected by edges called *transitions*. The taking of a transition is instantaneous, whereas time can elapse within a location. We employ a notation based on guard and assignment action labels (attributes *guard*, *assgn_action*, and *comm_action* of Transition).

For formally expressing a *timed gateway automaton*, we use a set of integer-valued clocks X, a set of variables V, a set of convertible elements C, and a set of messages M. Clocks and variables can be assigned a new value when a transition is taken. In addition, clocks and variables can be used to construct boolean expressions. Such a boolean expression is called a *guard*, when it is assigned to a transition and becomes a necessary prerequisite for the transition to be taken. In addition, an *action* can be assigned to a transition for associating operations with an edge. An action consists of *assignment actions* and *communication actions*. An assignment action assigns new values to variables and clocks. Communication actions express interactions with virtual networks and with the gateway repository (see Figure 4). Transitions are always atomic, i.e. transition-induced value changes of variables (through assignment and communication actions) and clocks (through assignment actions) only take place when the guard of a transition evaluates to true.

We capture convertible elements in the gateway repository through the set $C = \{\mathbb{N} \cup \infty\} \times \mathbb{N} \times \{\mathbb{T}, \mathbb{F}\} \times \mathbb{N}$. A convertible element is a tuple $c_k = \langle d^k_{\text{acc}}, n^k, b^k_{\text{req}}, t^k_{\text{update}} \rangle$, where the elements of this tuple represent the meta information introduced in Section 4. The convertible elements in C are part of the messages in M. The set M consists of the messages that are sent (M_s) and received (M_r) at the respective link, i.e. $M = M_s \cup M_r$. Each message $m \subseteq C$ is a set of convertible elements.

The sending of a message to a virtual network is expressed as $m_s!$ ($m_s \in M_s$) and comprises the transfer of message m_s to an output port. The contents of m_s are determined by the values of a set of convertible elements mapped into variables of the gateway automaton. Adversely, the receiving of a message m_r from an input port (expressed as $m_r?$, $m_r \in M_r$) leads to an update of a corresponding set of variables, which represent the convertible elements of m_r. Furthermore, a communication action can specify the transfer of a convertible element c_r from a variable of the gateway automaton into the gateway repository, which is denoted as *push*(c_r) ($c_r \in m_r, m_r \in M_r$). Adversely, *pull*(c_s) ($c_s \in m_s$, $m_s \in M_s$) expresses a communication action for the transfer of a convertible element c_s from the gateway repository into a variable of the gateway automaton.

Consequently, for a message $m_s \in M_s$, the permitted operations in a communication action are pull(c_s) ($c_s \in m_s$) for getting a constituting convertible element from the gateway repository, and $m_s!$ for sending message m_s at the output port. For a message $m_r \in M_r$, the permitted operations are push(c_r) ($c_r \in m_r$) for putting a constituting convertible element into the gateway repository and $m_r?$ for receiving message m_r at the input port.

The operations $m_r?$ and pull(c_s) are sensitive to the availability of messages and convertible elements in ports or the gateway repository respectively. In case of state semantics, availability means temporal accuracy of convertible elements or messages, while for event semantics availability means a non empty queue containing instances of the convertible element or message. For this reason, we define for a message m the temporal accuracy $d_{\text{acc}}(m)$ and the number of enqueued instances $n(m)$ as a function of the respective values of the constituting convertible elements.

The function $n(m)$ returns for a message m the minimum number of enqueued constituting convertible elements.

$$n(m) = \min_{c_k \in m} (n^k) \tag{2}$$

The *temporal accuracy horizon* $d_{\text{acc}}(m)$ of message m is the length of the remaining interval of time during which all constituting convertible elements of message m are tempo-

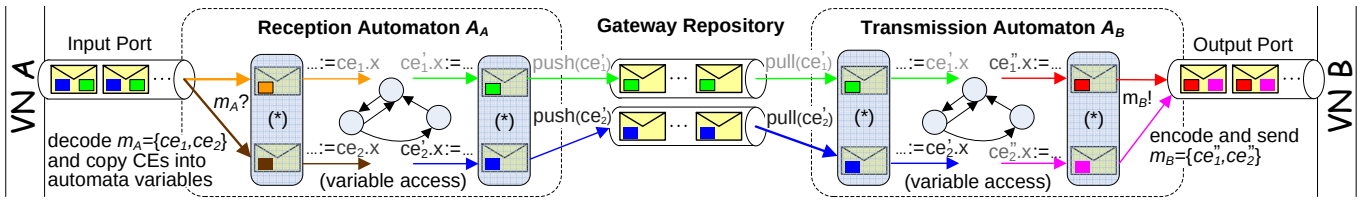


Figure 4. Convertible elements are mapped as variables (denoted by (*)) into the gateway automata

rally 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. $d_{\text{acc}}(m)$ is defined as

$$d_{\text{acc}}(m) = \begin{cases} \min\limits_{c_k \in m} \left(t^k_{\text{update}} + d^k_{\text{acc}} - t_{\text{now}}\right) & \text{if } m \in M_s \\ d^m_{\text{acc}} & \text{otherwise} \end{cases}$$

By computing the temporal accuracy horizon of a message $m_s \in M_s$ 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_s are temporally accurate until $t_{\text{now}}+d_{\text{acc}}(m_s)$. For the gateway side sending the message m_s to a network, the $d_{\text{acc}}(m_s)$ must be at least as large as the remaining time (e.g., operating system delays at the sender, communication delay, operating system delay at the receiver) until the point in time of use of the real-time image transported via m_s. By exploiting $d_{\text{acc}}(m_s)$, 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, at a gateway side employing 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.

We also extend the meta information about request from convertible elements to messages. The function $b_{\text{req}}(m)$ is true, if there is a constituting convertible element of message m for which an update has been requested.

$$b_{\text{req}}(m) = \begin{cases} \text{true} & \exists c_k \in m \text{ with } b^k_{\text{req}} = \mathbb{T} \\ \text{false} & \text{otherwise} \end{cases} \tag{3}$$

This function allows to construct timed automata for event-triggered message receptions according to the information pull principle. For this purpose, transmissions and receptions at a gateway side occur conditionally, namely based on the result of the function $b_{\text{req}}(m)$.

In order to formally define the gateway timed automaton, we specify what constraints are allowed as the enabling conditions called *guards* of a timed gateway automaton. We define the set $\Phi(X, V, M, C) : (X, V, M, C) \mapsto (\mathbb{T}, \mathbb{F})$ of guards for a set of clocks X, and variables V via the following grammar:

$$\begin{aligned} \varphi := \quad & \text{avail}(m) \mid \text{avail}(c) \mid b_{\text{req}}(m) \mid d_{\text{acc}}(m) \circ z \mid \\ & x \circ v \mid x \circ c \mid v \circ c \mid v \circ v' \mid \neg\varphi_1 \mid \varphi_1 \wedge \varphi_2 \end{aligned} \tag{4}$$

where x is a clock in X, z is a constant in $\mathbb{N}$, v and v' are variables in V, m is a message in M, and c is a convertible element in C. $\circ$ is a binary operator ($\leq, <, =, >, \geq$). avail(m) is short for $(d_{\text{acc}}(m) > 0) \wedge (n(m) > 0)$. avail$(m)$ determines whether the constituting convertible elements with event information are present in the queues of the gateway repository or the input port, and that the constituting convertible elements with state information are temporally accurate. The satisfaction of avail(m) is a prerequisite for transferring convertible elements of m into variables of the automaton via $m?$ or pull(c) ($c \in m$).

The set of *actions* $\Theta(X, V, M) : (X, V, M) \mapsto (X, V, M)$ is defined via the following grammar:

$$\vartheta := \; x := z \mid v := z \mid v := v' \mid \text{req}(m) \mid \vartheta_1; \vartheta_2, \tag{5}$$

where x is a clock in X, z is a constant in $\mathbb{N}$, v is a variable in V, and m is a message in M. The operation req(m) causes the setting of the request flags of all constituting convertible elements of m, i.e. $\forall c \in m : \; b^c_{\text{req}} := \mathbb{T}$.

Based on the above definitions, we define a *timed gateway automaton* A as a tuple $\langle L, L^0, X, V, M, C, I, E\rangle$. L is a finite set of locations and $L^0 \in L$ is the initial location. X is a finite set of clocks, V is a finite set of variables, C is a finite set of convertible elements, and M is a finite set of messages. I is a mapping that assigns to each location an invariant as a constraint in $\Phi(X, V, M, C)$ ($I : L \mapsto \Phi(X, V, M, C)$).

$E \subseteq L^2 \times \Phi(X, V, M, C) \times \Lambda$ is a set of transitions. A transition $\langle l_1, l_2, \varphi, \lambda\rangle$ represents an edge between location l_1 and location l_2 with the guard φ. The transition can only be taken, when the guard φ evaluates to $\mathbb{T}$.

$\Lambda = \Theta(X, V) \times 2^M \times 2^M \times 2^C \times 2^C$ is the set of actions. An action $\lambda \in \Lambda$ is a tuple $\langle \vartheta, m_s, m_r, c_w, c_r\rangle$. ϑ is a set of assignment operations to be performed with the transition. The set m_s defines which messages in M are to be sent at the transition (i.e. transferred to output ports, which is expressed as $m!$), while m_r defines the messages in M that are received at the transition (i.e. transferred from the gateway repository to variables of A, which is expressed as $m?$). The set c_w contains the convertible elements in C that are written from variables of the gateway automaton into the gateway repository (i.e. push(c_k), $c_k \in c_w$) when the transition is taken. The set c_r contains the convertible elements in C that are read from gateway repository and written into variables of the gateway automaton (i.e. pull(c_k), $c_k \in c_w$) when the transition is taken.

Since we require the timed gateway automaton to be deterministic, the constraint $I(l_k)$ of a location l_k must be:

$$I(l_k) = \neg \bigvee \varphi_i \text{ where } e_i \in E = \langle l_{1,i}, l_{2,i}, \varphi_i, \lambda_i\rangle \wedge l_{1,i} = l_k$$

Furthermore, the nonZenoness condition [1] must be satisfied. The nonZenoness condition requires that there is no in-

finite sequence of transitions without any progression of time in between.

Examples. In the following, we will explain how to use these automata for expressing temporal specifications for input and output ports with either a unidirectional or a request/reply pattern. These examples are meant to illustrate the capabilities of generic gateway services with temporal specifications provided by deterministic timed automata.

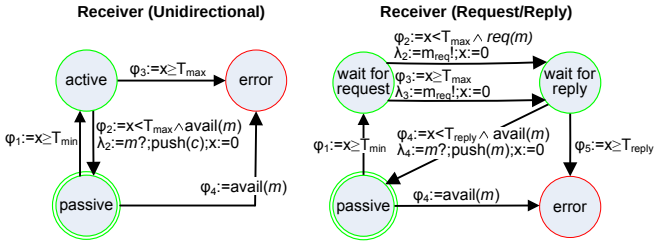


Figure 5. Message receptions.

Figure 5 contains exemplary timed gateway automata for 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 unidirectional reception scenario in Figure 5, 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". The "error state" is a special state of the automaton that enables the gateway to react to errors. The reaching of this state represents a violation of the temporal link specification and gives the gateway the ability to perform error handling. The error handling comprises the sending of a message indicating a symptom to the diagnostic subsystem [17] of the DECOS architecture and the restart of the gateway by returning to the initial 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 unidirectional scenario, in the request/reply scenario in Figure 5 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 *req(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.

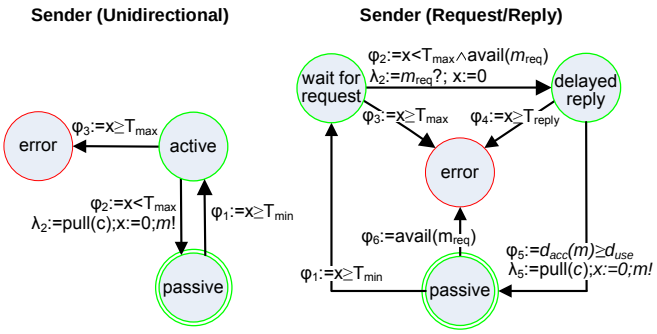


Figure 6. Message transmission.

Figure 6 depicts exemplary timed automata for event-triggered message transmissions. The initial state of the automaton for the unidirectional sender 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 case this guard blocks for the maximum interarrival time T_{max}, a transition into the "error state" takes place.

The automaton for the request/reply sender activity in Figure 6 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 ($d_{\text{acc}}(m) < d_{\text{use}}$). 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.

5.2. Syntactic Specification

The *syntactic specification* (see UML class diagram in Figure 3) defines the structure of Messages, Convertible Elements and Variables in terms of smaller structural elements denoted as Aggregates and Fields. An Aggregate allows the grouping of multiple structural elements (Aggregates and Fields). An Aggregate possesses a unique name and can represent an array with the integer attribute array denoting the array size. An Aggregate that is subdivided no further and considered atomic at the gateway is called a Field. A Field possesses a type (e.g., integer, string, floating point number). A Field is static, if the value of the field is time-invariant (value defined via the optional attribute const), while the value of a dynamic field can change over time.

Timed automata can contain several Variables. If a Variable is a clock variable (Boolean attribute *clock*), then the Variable must consist only a single Field of integer type. Each clock variable possesses a corresponding

epoch, which is initialized to 0 at the startup of the gateway. Furthermore, the epoch of a clock variable can be modified during the execution of the timed automaton (see below) through assignment operations of an automaton as defined in the temporal part of the link specification. The value of the clock variable, which is automatically incremented, denotes the number of ticks of the global time base since the epoch at a granularity of 2^{-24} seconds [**?**].

In contrast to the Variables of the automata, each Message also requires a *message name* that provides an identification by which the message is demarcated from other messages on the network. The message name is a statically defined part of a message. The message name consists only 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 network and allows the gateway to determine for each message instance the corresponding message entry in the link specification. The boolean attribute *naming* denotes whether an Aggregate is part of the message name. If an Aggregate is part of the message, then all constituting Aggregates and Fields are also regarded as parts of the message name.

Furthermore, each Aggregate of a Message can represent a Convertible Element for one or more automata. If an Aggregate is a Convertible Element, then all constituting Aggregates and Fields are also part of the Convertible Element. Convertible Elements are subject to selective redirection by the gateway and stored in the gateway repository. All other Aggregates and Fields in a Message are only of local interest to a network and discarded by the gateway. A Convertible Element possesses a unique name (attribute *name*) and either a corresponding temporal accuracy interval (attribute *d_acc*) in case of state semantics or a queue length (attribute *queue*) in case of event semantics.

5.3. Specification of Transfer Semantic

The transfer semantic defines a conversion for the information semantics of a convertible element (class Conversion in the UML class diagram in Figure 3). 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. 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.

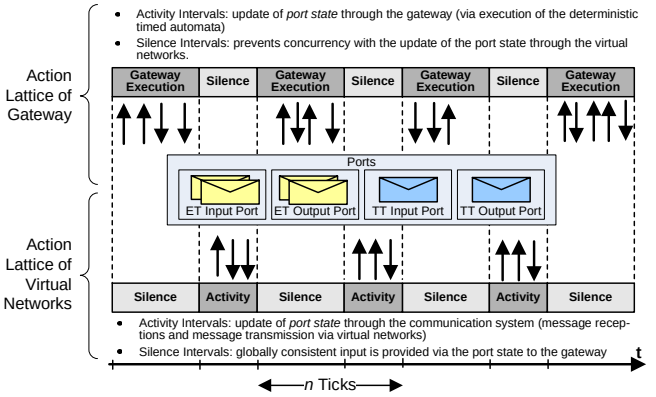


Figure 7. Execution on sparse time base

6. Execution of Virtual Gateway Service

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 [**?**]. 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 [9]. 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 [13, 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 [4]. For a port with state information, the port state comprises a state variable, while a message queue is the port state of an event port. As depicted in Figure 7, 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 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. The labels associated with the transitions of the automata cause interactions both with ports and the gateway repository.

`clocks`	X
`variables`	V
`messages`	M
`convertible elements`	C
`current location`	l

$T = T_{activation} - n$
`while` ($T < T_{activation}$)
  `while` ($\exists \langle l_1, l_2, \varphi, \lambda \rangle \in E$ with
    $\varphi = \mathbb{T} \wedge l = l_1 \wedge \lambda = \langle \vartheta, m_s, m_r, c_w, c_r \rangle$)
    $V \xrightarrow{m_s}$ `network` // send messages
    $V \xleftarrow{m_r}$ `network` // receive messages
    $V \xrightarrow{c_w}$ `repository` // push CE
    $V \xleftarrow{c_r}$ `repository` // pull CE
    $(X, V, M) := \vartheta(X, V, M)$ // execute action
    $l := l_2$ // new location
  `end`
  $T = T + 1$ // advance time
  $\forall x \in X : x = x + 1$
`end`

Table 1. Execution step (*n* ticks)

For each automaton, the gateway proceeds with evaluating guards and performing state changes with edges for which all guards are being fulfilled (see Table 1). In case no edge can be taken with the current values of the variables (including clock variables), time progresses by a tick, which is reflected by all clock variables maintained in the automaton. If time has progressed by n ticks and no guard is enabled, then 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 [**?**] 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.

7. Discussion and Conclusion

Virtual gateways in the integrated DECOS architecture permit the exchange of information between Distributed Application Subsystems (DASs), while preserving encapsulation. Virtual gateways are a generic architectural service that is parameterized with link specifications, each including a temporal specification as a timed gateway automaton. The link specification 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 link specification also enables the virtual gateway to resolve mismatches of operational specifications, resolve incoherent naming and perform transformations between event and state semantics.

7.1. Executability

For the specification of gateways, we use timed gateway automata with communication actions. The timed gateway automata serve as a parameterization of a generic architectural gateway service. The presented timed automata are nonZeno and deterministic. In order to ensure determinism, timed gateway automata impose restrictions on guards, invariants, and the points in time when variables can be modified through messages from the network.

1. **Guards.** Guards of transitions must ensure that no two guards are enabled simultaneously.
2. **Invariant.** The invariant of a location is the negated disjunction of the guards of all transitions originating from the location. Hence, it is ensured that a transition from a location is enabled, when the location's invariant is violated.
3. **Variables.** The variables that map convertible elements of messages into a gateway automaton are updated on a global sparse time base.

Thereby, the execution semantics of the timed gateway automata are fully specified. Since the gateway automata possess an unambiguously defined execution, they can form input to code generation and can be executed. Restrictions 1 and 2 ensure deterministic local execution semantics. Restriction 3 is a key element for a deterministic distributed execution, which permits to realized fault-tolerance through replicated gateways and exact voting (i.e. replica determinism [**?**]).

7.2. Property Transformation

A virtual gateway uses two timed gateway automata, each specifying the protocol for the transmission and reception of messages at a link of a respective virtual network. The gateway resolves different *temporal link specifications* by executing the first gateway automaton to receive messages and the second automaton to send messages. The two gateway automata in execution exchange information via the gateway repository, which is a real-time database that decouples the two sides of the gateway.

As long as the temporal behavior defined by one gateway automaton ensures temporal accuracy of convertible elements with state information in the gateway repository, the design of the second gateway automaton does not depend on detailed knowledge about the communication protocol at the other gateway side. In case of event information, bounded service and interarrival times of convertible elements are an example for a mechanism to abstract from the communication protocol at the other gateway side. Furthermore, the request variables (see meta information in the gateway repository in Section 4) allow to exchange control information between the two gateway sides in case on demand communication activities are required.

The user's ability to abstract from the communication protocol at the adjacent gateway side not only reduces the mental effort for designing an actual instance of the virtual gateway service, but also allows to devise universal gateway automata in the sense that an automaton of a particular virtual network can be reused regardless of the virtual network protocol of the other gateway side.

Different *syntactical link specifications* are resolved by splitting received message into smaller structural elements called convertible elements and recombining convertible elements from one or more messages into outgoing messages destined to the virtual network of the other gateway side. In addition, the syntax of convertible element can be converted either through the reception automaton or the transmission automaton. The first strategy permits to apply an abstract syntax for storing convertible elements, while the latter approach prevents superfluous conversions in case both gateway sides share the convertible element syntax.

The resolving of *semantical mismatches* includes the conversion between information semantics (state and event information) through conversion rules associated with convertible elements. Secondly, the gateway resolves naming incoherences in two virtual networks based on the link specifications, which contains a mapping between the name of a message and its constituting convertible elements.

7.3. Selective Redirection

The gateway specification not only defines which messages are subject to selective redirection, but also captures the convertible elements contained within each message that need to be written to the gateway repository and forwarded to the second virtual network. In addition, virtual gateways provide request variables as part of the meta information associated with convertible elements in the gateway repository. Using request variables, a transmission automaton can send messages only in response to a prior request from the virtual network.

In addition, the timed gateway 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.

References

[1] M. Abadi and L. Lamport. An old-fashioned recipe for real time. *ACM Trans. Program. Lang. Syst.*, 16(5):1543–1571, 1994.

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

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

[4] C. Jones et al. Final version of the DSoS conceptual model. *DSoS Project (IST-1999-11585)*, Dec. 2002.

[5] C. Carlson and J. Gerdes. Practical position and yaw rate estimation with GPS and differential wheelspeeds. In *Proc. of 6th Int. Symposium on Advanced Vehicle Control*, 2002.

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

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

[8] 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.

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

[10] 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, 2004.

[11] H. Kopetz and N. Suri. Compositional design of RT systems: A conceptual basis for specification of linking interfaces. In *Proc. of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, 2003.

[12] J. Lehoczky. Real-time queueing theory. In *Proc. of 17th IEEE Real-Time Systems Symposium*, Dec. 1996.

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

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

[15] R. Obermaisser, P. Peti, and H. Kopetz. Virtual networks in an integrated time-triggered architecture. In *Proc. of 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems*, 2004.

[16] OMG. Smart Transducers Interface. Specification ptc/2002-05-01, Object Management Group, May 2002. Available at `http://www.omg.org/`.

[17] P. Peti, R. Obermaisser, and H. Kopetz. Out-of-norm assertions. In *Proc. of IEEE Real-Time and Embedded Technology and Applications Symposium*, 2005.

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