Chapter 4
Real-Time Model

Overview The objective of this chapter is to introduce the reader to a cross-domain architecture model of the behavior of a real-time system. This model will be used throughout the rest of the book. The model is based on three basic concepts, the concept of a *computational component*, the concept of *state*, and the concept of a *message*. Large systems can be built by the recursive composition of components that communicate by the exchange of messages. Components can be reused on the basis of their interface specification without having to understand the component internals. Concerns about the understandability have been of utmost importance in the development of this model.

The chapter is structured as follows. In Sect. 4.1 we give a broad outline of the model, describing the essential characteristics of a *component* and a *message*. Related components that work towards a joint objective are grouped into *clusters*. The differences between *temporal control* and *logical control* are explained. The following section elaborates on the close relationship between *real-time* and the *state of a component*. The importance of a well-defined *ground state* for the dynamic reintegration of a component is highlighted. Section 4.3 refines the message concept and introduces the notions of event-triggered messages, time-triggered messages, and data streams. Section 4.4 presents the four interfaces of a component, two operational interfaces and two control interfaces. Section 4.5 deals with the concept of a *gateway component* that links two clusters that adhere to different *architectural styles*. Section 4.6 deals with the specification of the linking interface of a component. The linking interface is the most important interface of a component. It is relevant for the integration of a component within a cluster and contains all the information that is needed for the use of a component. The linking interface specifications consists of three parts: (1) the transport specification that contains the information for the transport of the messages, (2) the operational specification that is concerned with interoperability of components and the establishment of the message variables, and (3) the meta-level specification that assigns meaning to the message variables. Points to consider when composing a set of components to build *systems of subsystems* or *system of systems* are discussed in Sect. 4.6. In this section the four principles of composability are introduced and the notion of a multilevel system is explained.

H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, Real-Time Systems Series, DOI 10.1007/978-1-4419-8237-7_4,

4.1 Model Outline

Viewed from the perspective of an outside observer, a real-time (RT) system can be decomposed into three communicating subsystems: a *controlled object* (the *physical* subsystem, the behavior of which is governed by the laws of physics), a *"distributed" computer subsystem* (the *cyber system*, the behavior of which is governed by the programs that are executed on digital computers), and a *human user or operator*. The distributed computer system consists of computational nodes that interact by the exchange of messages. A *computational node* can host one or more *computational components*.

4.1.1 Components and Messages

We call the process of executing an algorithm by a processing unit a *computation or task*. Computations are performed by *components*. In our model, a component is a self-contained hardware/software unit that interacts with its environment exclusively by the exchange of messages. We call the *timed sequence of output messages* that a component produces at an interface with its environment the *behavior* of the component at that interface. The *intended behavior* of a component is called its *service*. An unintended behavior is called a *failure*. The internal structure of a component, whether complex or simple, is neither visible, nor of concern, to a user of a component.

A component consists of a *design* (e.g., the software) and an *embodiment* (e.g., the hardware, including a processing unit, memory, and an I/O interface). A *real-time component* contains a real-time clock and is thus aware of the progression of real-time. After *power-up,* a component enters a *ready-for-start* state to wait for a triggering signal that indicates the start of execution of the component's computations. Whenever the triggering signal occurs, the component starts its predefined computations at the *start instant*. In then reads input messages and its internal state, produces output messages and an updated internal state, and so on until it terminates its computation – if ever – at a *termination instant*. It then enters the *ready-for-start* state again to wait for the next triggering signal. In a cyclic system, the real-time clock produces a triggering signal at the start of the next cycle.

An important principle of our model is the *consequent separation of the computational components* from the *communication infrastructure* in a distributed computer system. The communication infrastructure provides for the transport of unidirectional messages from a sending component to one or more receiving components (*multicasting*) within a given interval of real-time. *Unidirectionality* of messages supports the unidirectional reasoning structure of *causal chains* and eliminates any dependency of the sender on the receiver(s). This property of *sender independence* is of utmost importance in the design of fault-tolerant systems, because it avoids error back-propagation from a faulty receiving component to a correct sending component by design.

Multicasting is required for the following reasons:

1. Multicasting supports the *non-intrusive observation* of component interactions by an independent observer component, thus making the interactions of components perceptually accessible and removing the barrier to understanding that has its origin in *hidden interactions* (see Sect. 2.1.3).
2. Multicasting is required for the implementation of fault-tolerance by active redundancy, where each single message has to be sent to a set of replicated components.

A message is sent at a *send instant* and arrives at the receiver(s) at some later instant, the *receive instant*. The message-paradigm combines the temporal-control and the value aspect of an interaction into a single concept. The temporal properties of a message include information about the send instants, the temporal order, the inter-arrival time of messages (e.g., periodic, sporadic, aperiodic recurrence), and the latency of the message transport. Messages can be used to synchronize a sender and a receiver. A message contains a data-field that holds a data structure that is transported from the sender to the receiver. The communication infrastructure is *agnostic* about the contents of the data field. The message concept supports *data atomicity* (i.e., atomic delivery of the complete data structure contained in a message). A single well-designed message-passing service provides a simple interface of a component to other components inside and outside a node and to the environment of a component. It facilitates encapsulation, reconfiguration, and the recovery of component services.

4.1.2 Cluster of Components

A cluster is a *set of related components* that have been grouped together in order to achieve a common objective (Fig. 4.1). In addition to the set of components,

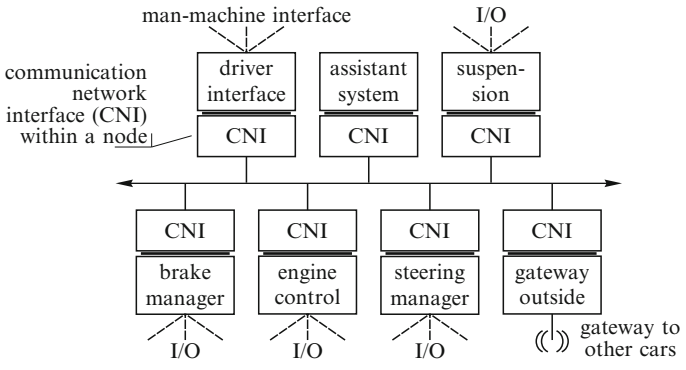

Fig. 4.1 Example of an *in-car cluster*

a cluster must contain an intra-cluster communication system that provides for the transport of messages among the components of the cluster. The components that form a computational cluster agree on a common *architectural style* (see the last paragraphs in Sect. 2.2.4).

Example: Figure 4.1 depicts an example of a computational cluster within a car. This cluster consists of a computational component, the *assistant system*, and gateway components to the man–machine interface (the driver), the physical subsystems of the car, and a gateway to other cars via a wire-less vehicle-to-vehicle communication link.

4.1.3 Temporal Control Versus Logical Control

Let us revisit the rolling mill example of Fig. 1.9 of Chap. 1 and specify a relation between measured variables that must be monitored by an alarm-monitoring task in the MMI component. Assume that the pressures p_1, p_2, and p_3, between the roles of the three drives are measured by the three controller components of Fig. 1.9. The measurements are sent to the man–machine interface (MMI) component for checking the following alarm condition:

when $((p_1 < p_2) \wedge (p_2 < p_3))$
then *everything ok*
else *raise pressure alarm*;

This looks like a reasonable specification at the user level. Whenever the pressure between the rolls does not satisfy the specified condition, a pressure alarm must be raised.

During the refinement of this specification by a system architect, four different tasks (three *measurement* tasks in the three control nodes and one *alarm-monitoring* task in the MMI node of Fig. 1.9) must be designed. The following questions concerning the temporal activation of these tasks arise:

1. What is the maximum tolerable time interval between the occurrence of the alarm condition in the controlled object and the raising of the alarm at the MMI? Because the communication among the components takes a finite amount of time, some time intervals are unavoidable!
2. What are the maximum tolerable time intervals between the three pressure measurements at the three different control nodes? If these time intervals are not properly controlled, false alarms will be generated or important alarms will be missed.
3. When and how often do we have to activate the pressure measurement tasks at the three control nodes?
4. When do we have to activate the alarm-monitoring task at the alarm-monitoring component (the MMI component in Fig. 1.9)?

Because these questions are not answered by the given specification, it is evident that this specification lacks precise information concerning the architectural requirements in the temporal domain. The temporal dimension is buried in the ill-specified

semantics of the **when** statement. In this example, the **when** statement is intended to serve two purposes. It is specifying

1. the point in time when the alarm condition must be raised, and
2. the conditions in the value domain that must be monitored.

It thus intermingles two separate issues, the behavior in the time domain and the behavior in the value domain. A clean distinction between these two issues requires a careful definition of the concepts of *temporal control* and *logical control*.

Temporal control is concerned with determining the instants in the domain of real time when computations must be performed, i.e., when tasks must be activated. These instants are derived from the dynamics of the application. In the above example, the decision regarding the instants at which the pressure measuring tasks and the alarm-monitoring task must be activated is an issue of temporal control. Temporal control is related to the progression of *real-time*.

Logical control is concerned with the control flow *within* a task that is determined by the given task structure and the particular input data, in order to realize the desired computation. In the above example, the evaluation of the branch condition and the selection of one of the two alternatives is an example of *logical control*. The time interval needed for the execution of a task that performs the logical control is determined by the frequency of the oscillator that drives the processing unit – we call this time-base the *execution time*. The execution time is determined by the given implementation and will change if we replace the given processor by a faster one.

Since *temporal control* is related to *real time*, while *logical control* is related to *execution time*, a careful distinction must be made between these two types of control (see also Sect. 8.3.4). A good design will separate these two control issues in order to decouple the reasoning about temporal constraints dictated by the application, from the reasoning about logical issues inside the algorithmic part of a program. Synchronous real-time languages, such as LUSTRE [Hal92], ESTEREL [Ber85], and SL [Bou96] distinguish cleanly between logical control and temporal control. In these languages, the progression of real-time is partitioned into an (infinite) sequence of intervals of specified real-time duration, which we call steps. Each step begins with a tick of a real-time clock that starts a computational task (logical control). The computational model assumes that a task, once activated by the tick of a real-time clock (temporal control), finishes its computation *quasi immediately*. Practically this means that a task must terminate its executions before the next triggering signal (the next tick of the real-time clock) initiates the next execution of the task.

The periodic finite state machine (PFSM) model [Kop07] extends the classic FSM, which is concerned with *logical control*, by introducing a new dimension for the progression of a global sparse time to cover *temporal control* issues.

If the issues of temporal control and logical control are intermingled in a program segment, then it is not possible to determine the worst-case execution time (WCET – see Sect. 10.2) of this program segment without analyzing the behavior of the environment of this program segment. This is a violation of the design principle *Separation of Concerns* (see Sect. 2.5).

Example: A *semaphore wait statement* is a temporal control statement. If a *semaphore wait statement* is contained in a program segment that also includes logical control (algorithmic) statements, then the temporal behavior of this program segment depends on both, the progress of *execution time* and the progress of *real-time* (see also Sects. 9.2 and 10.2).

4.1.4 Event-Triggered Control Versus Time-Triggered Control

In Sect. 4.1.1, we introduced the notion of a *triggering signal*, i.e., a control signal that indicates the instant when an activity should start in the temporal domain. What are the possible origins of such a triggering signal? The triggering signal can be associated either with the occurrence of a *significant event* – we call this *event-triggered control* – or with the arrival of a *specified instant* on the time line – we call this *time-triggered control*.

The significant events that form the basis of event-triggered control can be the arrival of a particular message, the completion of an activity inside a component, the occurrence of an external interrupt, or the execution of a *send message statement* by the application software. Although the occurrences of significant events are normally sporadic, there should be a minimal real-time interval between two successive events so that an overload of the communication system and the receiver of the events can be avoided. We call such an event-stream, for which a minimum inter-arrival time between events is maintained, a *rate-controlled* event stream.

Time-triggered control signals are derived from the progression of the global time that is available in every component. Time-triggered control signals are normally cyclic. A cycle can be characterized by its *period*, i.e., the real-time interval between two successive *cycle starts*, and by its *phase*, that is the interval between the start of the period, expressed in the global time, and the *cycle start* (see also Sect. 3.3.4). We assume that a cycle is associated with every time-triggered activity.

4.2 Component State

The concept of *state* of a component is introduced in order to separate *past behavior* from *future behavior* of a real-time component. The concept of state requires a clear distinction between past events and future events, i.e., there must be a consistent temporal order among the events of significance (refer to Sect. 3.3.2).

4.2.1 Definition of State

The notion of *state* is widely used in the computer science literature, albeit sometimes with meanings that are different from the meaning of *state* that is useful

in a real-time system context. In order to clarify the situation, we follow the precise definition of Mesarovic [Mes89, p. 45], which is the basis for our elaborations:

> The state enables the determination of a future output solely on the basis of the future input and the state the system is in. In other words, the state enables a "decoupling" of the past from the present and future. The state embodies all past history of a system. Knowing the state "supplants" knowledge of the past. ... Apparently, for this role to be meaningful, the notion of past and future must be relevant for the system considered.

The sparse time model introduced in Sect. 3.3.2 makes it possible to establish the consistent system-wide separation of the past from the future that is necessary to define a *consistent system state* in a distributed real-time computer system.

4.2.2 The Pocket Calculator Example

Let us look at the familiar example of a pocket calculator to investigate the concept of *state* in more detail. An operand, i.e., a number of keyboard digits, must be entered into the calculator before the selected operator, e.g., a key for the trigonometric function *sine*, can be pressed to initiate the computation of the selected function. After the computation terminates, the result is displayed at the calculator display. If we consider the computation to be an atomic operation and observe the system immediately before or after the execution of this atomic operation, the internal state of this simple calculator device is empty at the points of observation

Let us now observe the pocket calculator (Fig. 4.2) during the interval between the *start* of the computation and the *end* of the computation. If the internals of the device can be observed, then a number of intermediate results that are stored in the local memory of the pocket calculator can be traced during the series expansion of the *sine* function. If the computation is interrupted at an instant between the instants *start* and *end*, the contents of the program counter and all memory cells that hold the intermediate results form the *state at this instant* of interruption. After the end instant of the computation, the contents of these intermediate memory cells are no longer relevant, and the state is empty again. Figure 4.3 depicts a typical *expansion* and *contraction* of the state during a computation.

Let us now analyze the state (sometimes also called history state or h-state) of a pocket calculator used to sum up a set of numbers. When entering a new number, the sum of the previously entered numbers must be stored in the device. If we interrupt the work after having added a subset of numbers and continue the addition

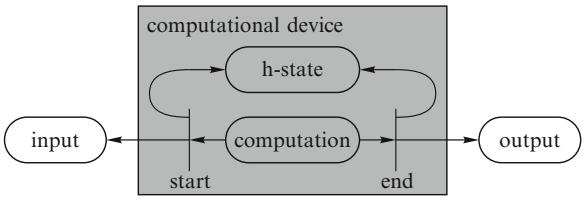

Fig. 4.2 Model of a pocket calculator

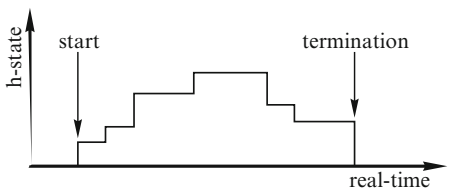

Fig. 4.3 Expansion and contraction of the *state* during a computation

with a new calculator, we first have to input the intermediate result of the previously added numbers. At the user level, the state consists of the intermediate result of the previous additions. At the end of the operation, we receive the final result and clear the memory of the calculator. The state is empty again.

From this simple example we can conclude that the size of the state of a system depends on the *instant of observation* of the system. If the granularity of observations is increased, and if the observation points are selected immediately before or after an atomic operation at the chosen level of abstraction, then, the size of the state can be reduced.

The state at any instant of interruption is contained in the contents of the program counter and all *state variables* that must be loaded into a *virgin* hardware device to resume the operation from the instant of interruption onward. If an interruption is caused by a failure of a component and we must reintegrate the component into a running system, then the size of the state that must be reloaded into the repaired component is of concern.

If our hardware device is a programmable computer, we must first load the software, i.e., operating system, the set of application programs, and the initial values for all state variables, into a *virgin* hardware device before we can start a computation. We call the *totality of software* that has to be loaded into a virgin hardware device the *core image* or the *job*. Normally, the job is a data structure that is static, i.e., it is not changed during the execution of the software. In some embedded hardware devices the job is stored in a ROM (read-only memory) and thus the *software becomes literally a part of the hardware*.

4.2.3 Ground State

In order to facilitate the dynamic reintegration of a component into a running system, it is necessary to design periodic *reintegration instants* into the behavior, where the size of a component's state at the reintegration instant contained is a small set of well-defined application specific *state variables*. We call the state at the reintegration instant the *ground state* (*g-state*) of a component and the temporal interval between two reintegration points the g*round cycle*.

The ground state at the reintegration point is stored in a declared *g-state data structure*. Designing a minimal ground state data structure is the result of an explicit design effort that involves a semantic analysis of the given application. The designer

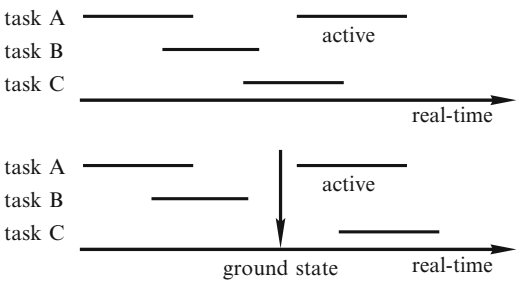

Fig. 4.4 Task executions: without (above), and with (below) ground state

has to find periodic instants where there is a maximum decoupling of future behavior from past behavior. This is relatively easy in *cyclic applications*, such as in control applications and multimedia applications. In these applications, a natural reintegration instant is immediately after the termination of one cycle and before the beginning of the next cycle. Design techniques for the minimization of the ground state are discussed in Sect. 6.6.

At the reintegration instant, no task may be active and all communication channels must be flushed, i.e., there are no messages in transit [Ahu90]. Consider a node that contains a number of concurrently executing tasks that exchange messages among each other and with the environment of the node. Let us choose a level of abstraction that considers the execution of a task as an *atomic action*. If the execution of the tasks is asynchronous, then, the situation depicted in the upper part of Fig. 4.4, can arise; at every instant, there is at least one active task, thus implying that there is no instant when the ground state of the node can be defined.

In the lower part of Fig. 4.4, there is an instant when no task is active and when all channels are empty, i.e., when the system is in the *g-state*. If a node is in the *g-state*, then the entire state that is essential for the future operation of the node is contained in the declared ground state data structure.

Example: Consider the relation between the size of the g-state and the duration of the ground (g) cycle in the design of a clock. If the g-cycle is 24 h and the start of a new day is the reintegration instant, then the g-state is empty. If every complete hour is a reintegration instant, then the g-state contains 5 bits (to identify one out of 24 h per day). If every complete minute is a reintegration instant, then the g-state is 11 bits (to identify one of 1,440 min per day). If every complete second is a reintegration instant, then the g-state is 17 bits (to identify one out of 86,400 s per day). It depends on the application characteristics to determine which one of the above alternatives is optimal for a given application. If the clock is an alarm clock that can store up to five alarms and the accuracy of an alarm is 5 min, then the g-state for every alarm is 9 bits (8 bit for the alarm and 1 bit to denote whether the alarm is *on* or *off*). If we assume that the reintegration cycle is 1 s and 5 alarms must be supported, then the g state message in this simple example is 62 bits in lengths. This g-state can be stored in an 8-byte data structure. In the restart message, the time field must be corrected such that it contains the precise time value at the restart instant.

Table 4.1 shows that g-*state recovery* is substantially different from *checkpoint recovery* that is used to establish a consistent state after a failure in a non-real-time data-intensive system.

Table 4.1 Comparison of g-state recovery and checkpoint recovery

	G-state recovery	Checkpoint recovery
Data selection	Application specific small data set that is essential for the future operation of the system.	All data elements that have been modified since the start of the computation.
Data modification	G-state data is modified to establish consistency between the g-state and the state of the environment at the future reintegration instant. Rollback of the environment is not possible in a real-time system.	No modification of checkpoint data. Consistency is established by rolling the (data) environment back to the instant when the checkpoint data was captured.

4.2.4 Database Components

We call a component where the number of *dynamic data elements*, i.e., data elements that are modified by the computations, is too large for storing them in a single ground state message a *database component*. The data elements that are contained in a database component can be either part of the *state* or *archival data*.

The term *archival data* refers to data that has been collected for archival purposes and does and not have a direct influence on the future behavior of the component. Archival data is needed to record the history of production process variables in order to be able to analyze a production process at some later time line. It is a good practice to send archival data to a remote storage site for archival data as soon as possible.

Example: The legendary *black box* in an airplane contains archival data. One could send these data immediately after collection via a satellite link to a storage site on the ground in order to avoid the problems of having to recover a black box after an accident.

4.3 The Message Concept

The concept of a *message* is the third basic concept of our model. A *message* is an atomic data structure that is formed for the purpose of communication, i.e., data transmission and synchronization among components.

4.3.1 Message Structure

The concept of a *message* is related to the concept of a *letter* in the postal system. A message consists of a header, a data field, and a trailer. The header, corresponding to the envelope of a letter, contains the *port address* of the receiver (the mailbox) where the message must be delivered, information about how the message must be

handled (e.g., a registered letter), and may contain the address of the sender. The data field contains the application specific data of the message, corresponding to the content of a letter. The trailer, corresponding to the signature in a letter, contains information that allows the receiver to detect whether the data contained in the message is uncorrupted and authentic. There are different types of trailers in use: the most common trailer is a CRC-field that allows the receiver to determine whether the data field has been corrupted during transport. A message may also contain an *electronic signature* in the trailer that makes it possible to determine whether the *authenticated contents* of the message have not been *altered* (see Sect. 6.2). The notion of *atomicity* implies that a message is delivered either in its entirety or not at all. If a message is corrupted or only parts of the message arrive at the receiver's site, the whole message is discarded.

The temporal dimension of the message concept relates to the instants when a message is sent by the sender and received by the receiver, and consequently how long the message has been in transit. We call the interval between the *send instant* and the *receive instant* the *transport delay*. A second aspect of the temporal dimension relates to the rate of message production by the sender and message consumption by the receiver. If the sending rate is constrained, then we speak about *a rate-constrained* message system. In case the sender's rate is not constrained, the sender may overload the transport capacity of the communication system (we call this *congestion*) or the processing capacity of the receiver. In case the receiver cannot keep up with the message production rate of the sender, the receiver can send a control message to the sender telling the sender to slow down (*back pressure flow control*). Alternatively, the receiver or the communication system may simply discard messages that exceed its processing capacity.

4.3.2 Event Information Versus State Information

The state of a dynamic system changes as real-time progresses. Let us assume that we periodically observe the state variables of a system with a duration d between two successive observation instants. If we observe that the value of all state variables is the same in two successive observations, then we infer that no *event*, i.e., *change of state,* has occurred in the last observation interval d. This conclusion is only valid, if the dynamics of the system is slow compared to our observation interval d (refer to *Shannon's theorem* [Jer77]). If two successive observations of the values of some state variables differ, then we conclude that at least one event has occurred in the last observation interval d. We can report about the occurrence of an event, i.e., a change of state, in two different ways: either by sending a single message containing *event information* or by sending a sequence of messages containing *state information*.

We talk about *event information* if the information conveys the *difference in values* of the previous state observation and the current state observation.

The instant of the current (later) observation is postulated to be the instant of event occurrence. This assumption is not fully accurate, since the event may have occurred at any instant during the last interval of duration d. We can reduce this temporal observation error of an event by making the interval d smaller, but we cannot fully eliminate the temporal uncertainty about the observation of events. This holds true even if we use the interrupt system of a processor to report about an event. The input signal that relays the occurrence of an interrupt is not sensed continuously by the processor, but only after the termination of the execution of an instruction. This delay is introduced in order to reduce the amount of *processor state* that has to be saved and restored in the processor in order to be able to continue the interrupted task after the interrupt has been served. As outlined in Sect. 4.2.3, the state is minimal immediately before or after the execution of an atomic operation – in this case, the execution of a complete instruction by a processor.

If the precise timing of an event is critical, we can provide a separate dedicated hardware device to time-stamp the observed state-change of the interrupt line immediately and thus reduce the temporal observation error to a value that is in the order of magnitude of the cycle time of the hardware. Such hardware devices are introduced to achieve precise clock synchronization in distributed systems, where the precision of the distributed clocking system must be in the nanosecond range.

Example: The IEEE 1588 standard for clock synchronization suggests the implementation of a separate hardware device to precisely capture the arrival instant of a clock synchronization message.

We talk about *state information* if the information conveys the *values* of the current state variables. If the data field of a message contains *state information*, it is up to the receiver to compare two successive state observations and determine whether an event has occurred or not. The temporal uncertainty about the event occurrence is the same as above.

4.3.3 Event-Triggered Message

A message is called *event-triggered (ET)* if the triggering signal for sending the message is derived from the occurrence of a significant event, such as the execution of a *send message* command by the application software.

ET messages are well suited to transport *event information*. Since an event refers to a *unique* change of state, the receiver must consume every single event message and it is not allowed to duplicate an event message. We say that an event message must adhere to the *exactly-once* semantics. The event message model is the standard model of message transmission that is followed in most non-real time systems.

Example: The event message *valve must be closed by* 5° means that the new intended position of the valve equals the current position plus 5°. If this event message is lost or duplicated, then the image of the state of the valve position in the computer will differ from

the actual state of the valve position in the environment by 5°. This error can be corrected by *state alignment*, i.e., the (full) state of the intended valve position is sent to the valve.

In an event-triggered system, error detection is in the responsibility of the sender who must receive an explicit acknowledgment message from the receiver telling the sender that the message has arrived correctly. The receiver cannot perform error detection, because the receiver cannot distinguish between *no activity by the sender* and *loss of message*. Thus the control flow must be *bidirectional* control flow, even if the data flow is only *unidirectional*. The sender must be *time-aware*, because it must decide within a finite interval of real time that the communication has failed. This is one reason why we cannot build fault-tolerant system that are unaware of the progression of *real time*.

4.3.4 Time-Triggered Message

A message is called *time-triggered (TT)* if the triggering signal for sending the message is derived from the progression of real-time. There is a *cycle*, characterized by its *period* and *phase*, assigned to every time-triggered message before the system starts operating. At the instant of cycle start, the transmission of the message is initiated automatically by the operating system. There is no *send message command* necessary in TT message transmission.

TT messages are well suited to transport *state information*. A TT message that contains state information is called a *state message*. Since a new version of a state observation normally replaces the existing older version, it is reasonable that a new state message *updates-in-place* the older message. On reading, a state message is *not consumed;* it remains in the memory until it is updated by a new version. The semantics of state messages is similar to the semantics of a *program variable* that can be read many times without consuming it. Since there are no queues involved in state message transmissions, queue overflow is no issue. Based on the *a priori* known cycle of state messages, the *receiver can perform error detection autonomously* to detect the loss of a state message. State messages support the *principle of independence* (refer to Sect. 2.5) since sender and receiver can operate at different (independent) rates and there is no means for a receiver to influence the sender.

Example: A temperature sensor observes the state of a temperature sensor in the environment every second. A state-message is well suited to transport this observation to a user that stores this observation in a program variable named *temperature*. The user-program can read this variable *temperature* whenever it needs to refer to the current temperature of the environment, knowing that the value stored in this variable is *up to date* to within about 2 s. If a single state message is lost, then for one cycle the value stored in this variable is *up to date* to only within about 3 s. Since, in a time-triggered system, the communication system knows *a priori* when a new state message must arrive, it can associate a *flag* with the variable temperature to inform the user if the variable temperature has been properly updated in the last cycle.

4.4 Component Interfaces

Let us assume that the design of a large component-based system is partitioned into two distinct design phases, *architecture design* and *component design* (see also Sect. 11.2 on *system design*). At the end of the architecture design phase, a *platform independent model* (PIM) of a system is available. The PIM is an executable model that partitions the system into clusters and components and contains the precise interface specification (in the domains of value and time) of the linking interfaces of the components. The linking interface specification of the PIM is *agnostic* about the component implementation and can be expressed in a high-level executable system language, e.g., in *System C*. A PIM component that is transformed to a form that can be executed on the final execution platform is called a *platform-specific model* (PSM) of the component. The PSM has the same interface characteristics as the PIM. In many cases, an appropriate compiler can transform the PIM to the PSM automatically.

An interface should serve a *single well-defined purpose* (Principle of *separation of concerns*, see Sect. 2.5). Based on *purpose*, we distinguish between the following four message interfaces of a component (Fig. 4.5):

- The *Linking Interface* (LIF) provides the specified service of the component at the considered level of abstraction. This interface is agnostic about the component implementation. It is the same for the PIM and the PSM.
- The *Technology Independent Control Interface* (TII) is used to configure and control the execution of the component. This interface is agnostic about the component implementation. It is the same for the PIM and the PSM.
- The *Technology Dependent Debug Interface* (TDI) is used to provide access to the internals of a component for the purpose of maintenance and debugging. This interface is implementation specific.
- The *Local Interface* links a component to the external world that is the external environment of a cluster. This interface is syntactically specified at the PSM level only, although the *semantic content* of this interface is contained in the LIF.

The LIF and the *local interface* are *operational interfaces*, while the TII and TDI are *control interfaces*. The control interfaces are used to control, monitor, or debug

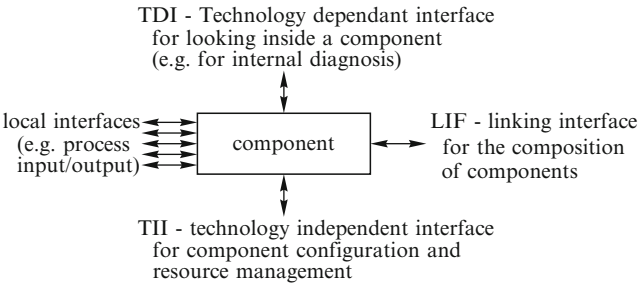

Fig. 4.5 The four interfaces of a component

a component, while the operational interfaces are in use during the normal operation of a component. Before discussing these four interfaces in detail, we elaborate on some general properties of message interfaces.

4.4.1 Interface Characterization

Push versus Pull Interface. There are two options to handle the arrival of a new message at the interface of a receiving component:

- *Information push*. The communication system raises an interrupt and forces the component to immediately act on the message. Control over the temporal behavior of the component is delegated to the environment outside of the component.
- *Information pull*. The communication system puts the message in an intermediate storage location. The component looks periodically if a new message has arrived. Temporal control remains inside the component.

In real-time systems, the information pull strategy should be followed whenever possible. Only in situations when an immediate action is required and the delay of one cycle that is introduced by the information pull strategy is not acceptable, one should resort to the information push strategy. In the latter case, mechanisms must be put into place to protect the component from erroneous interrupts caused by failures external to the component (see also Sect. 9.5.3). The information push strategy violates the principles of independence (see Sect. 2.5).

> **Example:** An engine control component for an automotive engine worked fine as long as it was not integrated with the theft avoidance system. The message interface between the engine controller and the theft avoidance system was designed as a push interface, causing the sporadic interruption of a time-critical engine control task when a message arrived from the theft avoidance system at an ill-timed instant. As a consequence the engine controller sporadically missed a deadline and failed. Changing the interface to a *pull interface* solved the problem.

Elementary versus Composite Interface. In a distributed real-time system, there are many situations where a simple unidirectional data flow between a sending and a receiving component must be implemented. We call such an interface *elementary* if both the data flow and the control flow are unidirectional. If, in a unidirectional data flow scenario, the control flow is bidirectional we call the interface *composite* (Fig. 4.6) [Kop99].

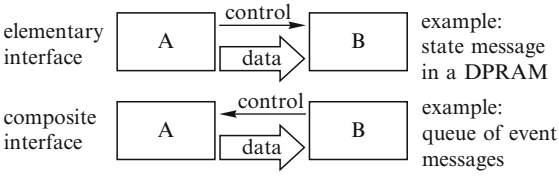

Fig. 4.6 Elementary vs. composite interface

Elementary interfaces are *inherently simpler* than composite interfaces, because there is *no dependency* of the behavior of the sender on the behavior of the receiver. We can reason about the correctness of the sender without having to consider the behavior of the receiver. This is of particular importance in safety-critical systems.

4.4.2 Linking Interface

The *services* of a component are accessible at its *cluster LIF*. The cluster LIF of a component is an *operational message-based* interface that interconnects a component with the other components of the cluster and is thus the interface for the integration of components into the cluster. The LIF of a component abstracts from the internal structure and the local interfaces of the component. The specification of the LIF must be self-contained and cover not only the functionality and timing of the component itself, but also the semantics of its local interfaces. The LIF is *technology agnostic* in the sense that the LIF does not expose implementation details of the internals of the component or of its local interfaces. A technology agnostic LIF ensures that different implementations of computational components (e.g., general purpose CPU, FPGA, ASIC) and different local Input/Output subsystems can be connected to a component without any modification to the other components that interact with this component across its message based LIF.

Example: In an input/output component, the external input and output signals are connected by a local point-to-point wiring interface. The introduction of a bus system, e.g., a CAN bus, will not change the cluster LIF of the input/output component, as long as the temporal properties of the data appearing at the LIF are the same.

4.4.3 Technology Independent Control Interface

The technology independent interface is a *control interface* that is used to configure a component, e.g., assign the proper names to a component and its input/output ports, to *reset*, *start,* and *restart* a component and to monitor and control the resource requirements (e.g., power) of a component during run time, if required. Furthermore, the TII is used to configure and reconfigure a component, i.e., to assign a specific *job* (i.e., core image) to the programmable component hardware.

The messages that arrive at the TII communicate either directly with the component hardware (e.g., *reset),* with the component's operating system (e.g., *start a task*), or with the *middleware* of the component, but not with the application software. The TII is thus orthogonal to the LIF. This strict separation of the application specific message interfaces (LIF) from the system control interface of a component (TII) simplifies the application software and reduces the overall complexity of a component (see also the *principle of separation of concerns* in Sect. 2.5).

4.4.4 Technology Dependent Debug Interface

The TDI is a *special control interface* that provides a means to look inside a component and to observe the internal variables of a component. It is related to the *boundary scan interface* that is widely used for testing and debugging large VLSI chips and has been standardized in the IEEE standard 1149.1 (also known as the JTAG Standard). The TDI is intended for the person who has a deep understanding of the component internals. The TDI is of no relevance for the user of the LIF services of the component or the system engineer who configures a component. The precise specification of the TDI depends on the technology of the component implementation, and will be different if the same functionality of a component is realized by software running on a CPU, by an FPGA or by an ASIC.

4.4.5 Local Interfaces

The local interfaces establish a connection between a component and its outside environment, e.g., the sensors and actuators in the physical plant, the human operator, or another computer system. A component that contains a local interface is called a *gateway component* or an *open component,* in contrast to a component that does not contain a local interface, which is called a *closed component*. The distinction between open and closed components is important from the point of view of the specification of the semantics of the LIF of the component. Only closed components can be fully specified without knowing the *context of use* of the component.

From the point of view of the *cluster LIF*, only the *timing* and the *semantic content*, i.e., the meaning of the information exchanged across a local interface is of relevance, while the detailed structure, naming, and access mechanisms of the local interface is *intentionally left unspecified* at the cluster level. A modification of the local access mechanisms, e.g., the exchange of a CAN Bus by Ethernet, will not have any effect on the LIF specification, and consequently on the users of the LIF specification, as long as the *semantic content* and the *timing* of the relevant data items are the same (see also the *principle of abstraction* in Sect. 2.5).

Example: A component that calculates a trigonometric function is a *closed component*. Its functionally can be formally specified. A component that reads a temperature sensor is an *open component*. The meaning of *temperature* is application specific, since it depends on the position where the sensor is placed in the physical plant

4.5 Gateway Component

Viewed from the perspective of a cluster, a gateway component is an *open component* that links two worlds, the *internal world* of the cluster and the *external world* of the cluster environment. A gateway component acts as a *mediator* between these two

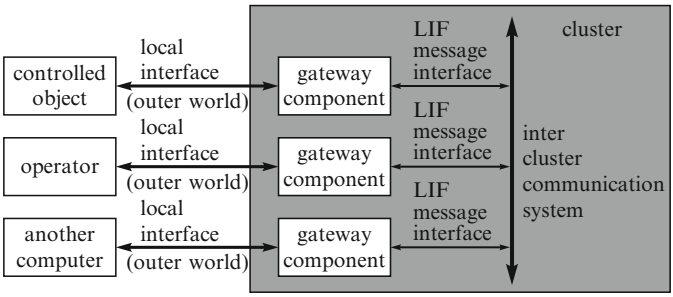

Fig. 4.7 Gateway components between the LIF and the local external world interface

worlds. It has two operational interfaces, the LIF message interface to the cluster and the local interface to the external world, which can be the physical plant, a man–machine interface, or another computer system (Fig. 4.7). Viewed from the outside of a cluster, the *role of the interfaces is reversed*. The previous local interface becomes the new LIF and the previous LIF becomes the new local interface.

4.5.1 Property Mismatches

Every system is developed according to an *architectural style*, i.e., a set of adopted rules and conventions for the concept formation, representation of data, naming, programming, interaction of components, semantics of the data, and many more. The architectural style characterizes the *properties* of the entities that represent the design. The details of the architectural style are sometimes explicitly documented, but more often only implicitly followed by the development team, reflecting the unstated conventions that govern the development community (see also the last paragraph of Sect. 2.2.4).

Whenever a communication channel links two systems, developed by two different organizations, it is highly probable that some of the properties of the messages that are exchanged across this channel are in disagreement. We call any disagreement in a property of the data or the protocol between the sender and the receiver of a message a *property mismatch*. It is up to a gateway component to resolve property mismatches.

> **Example:** Assume that there is a difference in *endianness*, i.e., the byte ordering of data, between the sender and the receiver of a message. If the sender assumes *big endian*, i.e., the most significant byte is first, and the receiver assumes *little endian*, i.e., the least significant byte is first, then this *property mismatch* has to be resolved either by the sender, or by the receiver, or by an intermediate connector system, i.e., the gateway component.

Property mismatches occur at the borders where systems interact, not inside a well-designed system. Inside a system, where all partners respect the rules and constraints of the architectural style, property mismatches are normally no issue.

Property mismatches should be resolved in the gateway components that link two systems in order to maintain the integrity of the architectural styles within each of the interacting subsystems.

4.5.2 LIF Versus Local Interface of a Gateway Component

As noted before, the set of related components that form a cluster share a common *architectural style* at their cluster LIFs. This means that *property mismatches* among LIF-cluster messages are rare.

This is in contrast to the messages that cross a gateway component. These messages come from two different worlds that are characterized by two different architectural styles. Property mismatches, syntactic incompatibility, incoherent naming, and differences in representation between these two sets of messages are the rule rather than the exception. It is the main task of a gateway component to translate the architectural style of one world to the architectural style of the other world without changing the *semantic content* of the message variables.

There are situations when the architectural styles of the two interfacing worlds are based on a different *conceptualization* of reality. i.e., there are not only differences in the *names* and the *representations* of the same concepts (as shown in the example in Sect. 2.2.4), but the concepts themselves, i.e., the semantic contents are dissimilar.

Viewed from a given cluster, the gateway component hides the details of the external world (the local interface) from the standardized message formats within the computational cluster and filters the incoming information: only the information that is relevant for the operation of the cluster under consideration is exposed in the form of cluster-standard messages at the cluster LIF of the gateway component.

An important special case of an external-world interface is a process I/O interface, which establishes a link between the cyber world and the physical world. Table 4.2 depicts some of the differences between the LIF messages and a process control I/O interface to the physical plant.

Table 4.2 Characteristics of a LIF versus a local process I/O interface

Characteristic	Local process I/O interface	LIF message interface
Information Representation	Unique, determined by the given physical interface device	Uniform within the whole cluster
Coding	Analog or digital, unique	Digital, uniform codes
Time-base	Dense	Sparse
Interconnection Pattern	One-to-one	One-to-many
Coupling	Tight, determined by the specific hardware requirements and the I/O protocol of the connected device	Weaker, determined by the LIF message communication protocol

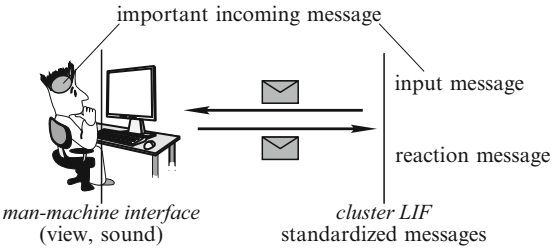

Fig. 4.8 Standardized LIF versus concrete MMI

Example: In the external world of a process plant the value of the process variable *temperature* may be coded in an analog 4–20 mA sensor signal, where 4 mA means 0% and 20 mA means 100% of the selected measurement range, i.e., between 0° and 100°C. This analog representation must be converted to the standard representation of temperature that has been defined in the architectural style of the given cluster, which might be degrees *Kelvin*.

Example: Let us look at an important interface, the man–machine interface (MMI), in order to distinguish between the *local interface* and the *LIF message interface* of a gateway MMI component (Fig. 4.8). At the level of architectural modeling, we are not interested in the representational details of the local external world interface, but only in the *semantic content* and temporal properties of the message variables at the *LIF message interface*. An important message is sent to the MMI component. It is somehow relayed to the operator's mind. A response message from the operator is expected within a given time interval at the *LIF message interface*. All intricate issues concerning the representation of the information at the graphic user interface (GUI) of the operator terminal are irrelevant from the point of view of architecture modeling of the interaction patterns between the operator and the cluster. If the purpose of our model were the study of human factors governing the specific man–machine interaction, then the form and attributes of the information representation at the GUI (e.g., shape and placement of symbols, color, and sound) would be relevant and could not be disregarded.

A gateway component can link two different clusters that have been designed by two different organizations using two different architectural styles. Which one of the two interfaces of such a gateway component is considered the LIF and which one is the local interface depends on the view taken. As already mentioned, if we change the view, the LIF becomes the local interface and the local interface becomes the LIF.

4.5.3 Standardized Message Interface

To improve the compatibility between components designed by different manufacturers, to enhance the interoperability of devices, and to avoid property mismatches, some international standard organizations have attempted to standardize message interfaces. An example of such a standardization effort is the SAE J 1587 Message Specification. The Society of Automotive Engineers (SAE) has standardized

the message formats for heavy-duty vehicle applications in the J 1587 Standard. This standard defines message names and parameter names for many data elements that occur in the application domain of heavy vehicles. Besides data formats, the range of the variables and the update frequencies are also covered by the standard.

4.6 Linking Interface Specification

As noted in Sect. 4.1.1, the *timed sequence of messages* that a component exchanges across an interface with its environment defines the *behavior* of the component at that interface [Kop03]. The *interface behavior* is thus determined by the properties of all messages that cross an interface. We distinguish three parts of an interface specification: (1) the *transport specification* of the messages, (2) the *operational specification* of the messages, and the (3) the *meta-level specification* of the messages.

The *transport specification* describes all properties of a message that are needed to transport the message from the sender to the receiver(s). The transport specification covers the addressing and temporal properties of a message. If two components are linked by a communication system, the transport specification suffices to describe the requested services from the communication system. The communication system is *agnostic* about the contents of the data field of a message. For the communication system is does not matter whether the data field contains multimedia data, such as voice or video, numerical data or any other data type.

Example: The Internet provides a defined message transport service between two end systems, not knowing what types of digital data are transported.

In order to be able to interpret the data field of a message at the end points of the communication, we need the *operational* and the *meta-level specification*. The *operational specification* informs about the syntactic structure of the message that is exchanged across the LIF and establishes the *message variables*. Both, the transport and the operational specification must be *precise* and *formal* to ensure the *syntactic interoperability* of components. The *meta-level specification* of a LIF assigns meaning to the *message variable names* introduced by the operational specification. It is based on an interface model of the user environment. Since it is impossible to formalize all aspects of a real-world user environment, the meta-level specification will often contain natural language elements, which lack the precision of a formal system. Central concepts of the application domains and applications can be specified using *domain specific ontologies*.

4.6.1 Transport Specification

The *transport specification* contains the information needed by the communication system to transport a message from a sender to the receiver(s). An interface contains

a number of ports where the messages destined for this interface arrive or where messages are placed for sending. The following attributes must be contained in the transport specification:

- Port address and direction
- Length of the data field of the message
- Type of message (e.g., *time-triggered* or *event-triggered* or *data stream*)
- Cycle of a time-triggered message
- Queue depth for an event-triggered message or a data stream

It is a design decision whether these attributes are linked with the message or are associated with the port where the message is handled.

As noted above, the transport specification must contain the information about the temporal properties of a message. For time-triggered messages, the temporal domain is precisely specified by the cycle that is associated with every time-triggered message. For event-triggered messages, the temporal specification is more difficult, particularly if event-triggered messages can arrive in bursts. In the long run, the message arrival rate must not be larger than the message consumption rate by the receiver, since an event-triggered message must conform to the *exactly once semantics* (see Sect. 4.3.3). In the short run, an arriving message burst can be buffered in the receiver queue until the queue is full. The specification of the proper queue length for bursty event-triggered messages is very important.

4.6.2 Operational Specification

From the point of view of communication, the data field of an arriving message can be considered as an *unstructured bit vector*. At the endpoints of the communication, the operational specification determines how this bit vector must be structured into *message variables*. A *message variable* is a syntactic unit that consists of a fixed part and a variable part (see Sect. 2.2.4). The information about how the data field of a message is structured in syntactic units is contained in a *message-structure declaration (MSD)*. The MSD contains the *message variable names* (i.e., the fixed part of the message variable) that point to the relevant concepts on one side and, on the other side, specifies which part of the unstructured bit vector denotes *the value (the variable part) of a message variable*. In addition to the structure information, the MSD may contain *input assertions* for checking the validity of incoming data (e.g., to test if the data is within a permitted data domain that the receiving component is capable to handle) and *output assertions* for checking outgoing data. An incoming data element that passes the input assertion is called a *permitted data element*. An outgoing data element that passes the output assertion is called a *checked data element*. The formalism used for specifying the data structures and the assertions in the MSD depends on the available programming environment.

In many real-time systems the MSD is static, i.e., it does not change over the lifetime of the system. In these systems, for performance reasons, the MSD is not transmitted in the message but stored in the memories of the communicating partners where the data fields must be handled.

The link between the *unstructured bit vector* arriving at a port and the associated MSD can be established by different means:

- The *MSD name* is assigned to the input port name. In this case only a single message type can be accepted at a port.
- The *MSD name* is contained in the data field of the message. In this case different message types can be received at the same port. This approach is followed in CAN, (see Sect. 7.3.2).
- The *MSD name* is assigned to the cyclic arrival instant of a time-triggered message. In this case different message types can be received at the same port, without the need to store the MSD name in the data field of the message. This approach is followed in TTP see Sect. 7.5.1.
- The *MSD name* is stored in a server that can be accessed by the receiver of a message. This approach is followed in CORBA [Sie00].
- The MSD itself is part of the message. This is the most flexible arrangement, at the cost of having to send the full MSD in every message. This approach is followed in service-oriented architectures (SOA) [Ray10].

4.6.3 Meta-Level Specification

The meta-level LIF specification assigns a meaning to the message variables exchanged between two communicating LIFs at the operational level and thus establishes *semantic interoperability*. It thus bridges the gap between the syntactic units and the user's mental model of the service provided at the interface. Central to this meta-level specification is the LIF service model. The LIF service model defines the concepts that are associated with the *message variable names* contained in the operational specification. These concepts will be qualitatively different for *closed components* and *open components* (see Sect. 4.4.5).

The LIF service model for a *closed* component can be formalized, since a closed component does not interact with the external environment. The relationship between the LIF inputs and LIF outputs depends on the discrete algorithms implemented within the closed component. There is no input from the external environment that can bring unpredictability into the component behavior. The sparse time-base within a cluster is discrete and supports a consistent temporal order of all events.

The LIF service model for an *open* component is fundamentally different since it must encompass the inputs from the external environment, the local interfaces of the component in its interface specification. *Without knowing the context of use of an open component, only the operational specification of an open component can be*

provided. Since the external physical environment is not rigorously definable, the interpretation of the external inputs depends on human understanding of the natural environment. The concepts used in the description of the LIF service model must thus fit well with the accustomed concepts within a user's *internal conceptual landscape* (see Sect. 2.2); otherwise the description will not be understood.

The discussion that follows focuses on LIFs of open components, since the systems we are interested in must interact with the external environment. The LIF service model of an open component must meet the following requirements:

- *User orientation*. Concepts that are familiar to a prototypical user must be the basic elements of the LIF service model. For example, if a user is expected to have an engineering background, terms and notations that are *common knowledge* in the chosen engineering discipline should be used in presenting the model.
- *Goal orientation*. A user of a component employs the component with the intent to achieve a goal, i.e., to contribute to the solution of her/his problem. The relationship between user intent and the services provided at the LIF must be exposed in the LIF service model.
- *System view*. A LIF service user (the system architect) needs to consider the system-wide effects of an interaction of the component with the external physical environment, i.e., effects that go beyond the component. The LIF service model is different from the model describing the algorithms implemented within a component, since these algorithms are within the component's boundaries.

Example: Let us analyze the simple case of a variable that contains a temperature. As any variable, it consists of the two parts, the *static variable name* and the *dynamic variable value*. The MSD contains the *static variable name (*let us assume the variable is named *Temperature-11*) and the position where the *dynamic variable value* is placed in an arriving bit stream of the message. The meta-level specification explains the meaning of *Temperature-11* (see also the examples in Sect. 2.2.4).

4.7 Component Integration

A component is a self-contained validated unit that can be used as a building block in the construction of larger systems. In order to enable a straightforward composition of a component into a cluster of components, the following four *principles of composability* should be observed.

4.7.1 Principles of Composability

1. *Independent Development of Components*: The architecture must support the precise specification of the linking interface (LIF) of a component in the domains

of value and time. This is a necessary prerequisite for the independent development of components on one side and the reuse of existing components that is based solely on their LIF specification on the other side. While the operational specification of the value domain of interacting messages is *state-of-the-art* in embedded system design, the temporal properties of these messages are often ill defined. Many of the existing architectures and specification technologies do not deal with the temporal domain with the appropriate care. Note that the transport specification and the operational LIF specification are independent of the context of use of an open component, while the meta-level LIF specification of an open component depends on the context of use. *Interoperability* of open components is thus not the same as *interworking* of open component, since the latter assumes the compatibility of the meta-level specifications.

2. *Stability of Prior Services*: The *stability of prior services principle* states that the services of a component that have been validated in isolation (i.e., prior to the integration of the component into the larger system) remain intact after the integration (see the example in Sect. 4.4.1).
3. *Non-Interfering Interactions*: If there exist two disjoint subgroups of cooperating components that share a common communication infrastructure, then the communication activities within one subgroup must not interfere with the communication activities within the other subgroup. If this principle is not satisfied, then the integration within one component-subgroup will depend on the proper behavior of the other (functionally unrelated) component-subgroups. These global interferences compromise the composability of the architecture.

   Example: In a communication system where a single communication channel is shared by all components on a *first-come first-serve basis*, a *critical instant* is defined as an instant, when all senders start sending messages simultaneously. Let us assume that in such a system ten components are to be integrated into a cluster. A given communication system is capable to handle the critical instant if eight components are active. As soon as the ninth and tenth component are integrated, sporadic timing failures are observed.

4. *Preservation of the Component Abstraction in the Case of Failures*: In a composable architecture, the introduced abstraction of a component must remain intact, even if a component becomes faulty. It must be possible to diagnose and replace a faulty component without any knowledge about the component internals. This requires a certain amount of redundancy for error detection within the architecture. This principle constrains the implementation of a component, because it restricts the implicit sharing of resources among components. If a shared resource fails, more than one component can be affected by the failure.

   Example: In order to detect a faulty component that acts like a *babbling idiot*, the communication system must contain information about the permitted temporal behavior of every component. If a component fails in the temporal domain, the communication system cuts off the component that violates its temporal specification, thus maintaining the timely communication service among the correct components.

4.7.2 Integration Viewpoints

In order to bring an understandable structure into a large system, it makes sense to view – as seen from an integration viewpoint – a (original) cluster of components as a single gateway component. The integration viewpoint establishes a new cluster that consists of the respective gateway components of the original clusters. Viewed from an original cluster, the external (local) interface of the gateway become the LIFs of the new cluster, while, viewed from the new cluster, the LIF of the gateway to the original cluster is the local interface of the new cluster (see Sect. 4.5.2). The gateways to the new cluster make only those information items available to the new cluster that are of relevance for the operation of the new cluster.

Example: Figure 4.1 depicts a cluster of components that form the control system within an automobile. The vehicle-to-vehicle gateway component (the right lower component in Fig. 4.1) establishes a wireless link to other vehicles. In this example, we distinguish the following two levels of integration: (1) the integration of components into the cluster depicted in Fig. 4.1 and (2) the integration of a car into a *dynamic system of cars* that is achieved via the car-to-car (C2C) gateway component. If we look at the integration of components within the cluster of Fig. 4.1, then the communication network interface (CNI) of the C2C gateway component is the *cluster LIF*. From the C2C communication viewpoint, the cluster LIF is the (unspecified) local interface of the C2C gateway component (see also the last paragraph of Sect. 4.5.2).

The hierarchical composition of components and clusters that leads to *distinct integration levels* is an important special case of the integration of components. Multi-levelness is an important organizing principle in large systems. At the lowest integration level primitive components (i.e., components that are considered to be atomic units and are not composed any further) are integrated to form a cluster. One distinct component of this cluster is a gateway component that forms, together with distinct gateway components of other clusters a *higher level cluster*. This process of integration can be continued recursively to construct a hierarchical system with distinct levels of integration (see also Sect. 14.2.2 on the recursive integration of components in the time-triggered architecture).

Example: In the GENESYS [Obm09, p. 44] architecture, three integration levels are introduced. At the lowest level, the chip level, the components are IP cores of an MPSoC that interact by a network on chip. At the next higher level, chips are integrated to form a device. At the third integration level devices are integrated to form *closed* or *open* systems. A *closed* system is a system in which the subsystems that form the systems are known *a priori*. In an open system subsystems (or devices) join and leave the system dynamically, leading to a system-of-systems.

4.7.3 System of Systems

There are two reasons for the rising interest in *systems-of-systems:* (1) the realization of new functionality and (2) the control of the complexity growth of large

systems caused by their continuous evolution. The available technology (e.g., the Internet) makes it possible to interconnect *independently developed systems* (*legacy systems*) to form new *system-of-systems (SoS)*. The integration of different *legacy systems* into an SoS promises more efficient economic processes and improved services.

The continuous adaptations and modifications that are necessary to keep the services of a large system relevant in a dynamic business environment brings about a growing complexity that is hard to manage in a monolithic context [Leh85]. One promising technique to attack this complexity problem is to break a single large monolithic system up into a set of nearly autonomous constituent systems that are connected by well-defined message interfaces. As long as the *relied-upon properties* at these message interfaces meet the user intentions, the internal structure of the constituent systems can be modified without any adverse effect on the global system level services. Any modification of the *relied-upon properties* at the message interfaces is carefully coordinated by a coordination-entity that monitors and coordinates the system evolution. The SoS technology is thus introduced to get a handle on the complexity growth, caused by the necessary evolution of large systems, by introducing structure and applying the simplification principles of *abstraction*, *separation of concerns,* and *observability of subsystem interactions* (see Sect. 2.5).

The distinction between a *system of sub-systems* and a *system of systems* is thus based on the degree of autonomy of the constituent systems [Mai98]. We call a monolithic system made out of subsystems that are designed according to a master plan and are in the *sphere of control* of a single development organization a *system of sub-system*. If the *systems* that cooperate are in the *sphere of control* of different development organization we speak of a *system of (constituent) systems*. Table 4.3 compares the different characteristics of a *monolithic system* versus a *system of systems*. The interactions of the autonomous constituent systems can evoke planned or unanticipated emergent behavior, e.g., a *cascade effect* [Fis06], that must be detected and controlled (see also Sect. 2.4).

In many distributed real-time applications it is not possible to bring *temporally accurate real-time information* to a central point of control within the available time interval between the *observation of the local environment* and the need to *control the local environment*. In these applications central control by a monolithic control system is not possible. Instead, the autonomous distributed controllers must cooperate to achieve the desired effects.

Example: It is not possible to control the movement of the cars in an open road system, where cyclists and pedestrians interfere with the traffic flow of the cars, by a monolithic central control system because the amount and timeliness of the real-time information that must be transported to and processed by the central control system is not manageable within the required response times. Instead, each car performs autonomous control functions and cooperates with other cars in order to maintain an efficient flow of traffic. In such a system a *cascade effect* of a traffic jam can occur due to emergent behavior if the traffic density increases beyond a tipping point.

Any ensemble of constituent systems that form an SoS must agreed on a shared purpose, a chain of trust, and a shared ontology on the semantic level. These global

Table 4.3 Comparison of a *monolithic system* and a *system of systems*

Monolithic system	System of systems (SoS)
Sphere of control and system responsibility within a single development organization. Subsystems are obedient to a central authority.	Constituent systems are in the sphere of control of different development organizations. Subsystems are autonomous and can only be influenced, but not controlled, by other subsystems.
The architectural styles of the subsystems are aligned. Property mismatches are the exception.	The architectural styles of the constituent systems are different. Property mismatches are the rule, rather than the exception.
The LIFs that effectuate the integration are controlled by the responsible system organization.	The LIFs that effectuate the integration are established by international standard organizations and outside the control of a single system supplier.
Normally hierarchical composition that leads to levels of integration.	Normally the interactions among the constituent systems follow a mesh network structure without clear integration levels.
Subsystems are designed to interact in order to achieve the system goal: *Integration*.	Constituent systems have their own goals that are not necessarily compatible with the SoS goal. Voluntary cooperation of systems to achieve a common purpose: *Interoperation*.
Evolution of the components that form the subsystems is coordinated.	Evolution of the constituent systems that form the SoS is uncoordinated.
Emergent behavior controlled.	Emergent behavior often planned, but sometimes unanticipated.

properties must be established at the meta-level and are subject of a carefully managed continuous evolution. A new entity must be established at the meta-level that monitors and coordinates the activities of the constituent systems in order that the shared purpose can be achieved.

An important characteristic of an SoS is the independent development and uncoordinated evolution of the constituent systems (Table 4.3). The focus in SoS design is on the linking interface behavior of the monolithic systems. The monolithic system themselves can be heterogeneous. They are developed according to different architectural styles by different organizations. If the monolithic systems are interconnected via open communication channels, then the topic of security is of utmost concern, since an outside attacker can interfere with the system operation, e.g., by executing a *denial-of-service* attack (see Sect. 6.2.2).

[Sel08, p. 3] discusses two important properties of an *evolutionary architecture*: (1) The complexity of the overall framework does not grow as constituent systems are added or removed and (2) a given constituent system does not have to be reengineered if other constituent systems are added, changed, or removed. This implies a precise specification and continuous revalidation of the relied upon interface properties (in the domains of value and time) of the constituent systems. The evolution of a constituent system will have no adverse effect on the overall behavior if the *relied-upon interface properties* of this constituent system are

not modified. Since the precise specification of the temporal dimension of the relied-upon interface properties requires a time-reference, the availability of a synchronized global time in all constituent systems of a large SoS is helpful, leading to a *time-aware architecture* (TAA, see Sect. 14.2.5). Such a global time can be established by reference to the global GPS signals (see Sect. 3.5). We call an SoS where all constituent systems have access to a synchronized global time a *time-aware SoS*.

The preferred interconnection medium for the construction of *systems of systems* is the Internet, leading to the Internet of Things (IoT). Chapter 13 is devoted to the topic of the *Internet of Things*.

Points to Remember

- A real-time component consists of a *design* (e.g., the software), an *embodiment* (e.g., the hardware, including a processing unit, memory, and an I/O interface), and a real-time clock that makes the component aware of the progression of real-time.
- The *timed sequence of output messages* that a component produces at an interface with its environment is the *behavior* of the component at that interface. The *intended behavior* is called the *service*. An unintended behavior is called a *failure*.
- *Temporal control* is concerned with determining the instants in the domain of real time when tasks must be activated while *logical control* is concerned with the control flow *within* a task.
- *Synchronous programming languages* distinguish cleanly between temporal control, which is related to the progression of real time, and logical control, which is related to execution time.
- A cycle, characterized by its *period* and *phase*, is associated with every time-triggered activity.
- At a given instant, the *state of a component* is defined as a data structure that contains the information of the past that is relevant for the future operation of the component.
- In order to enable the dynamic reintegration of a component into a running system, it is necessary to design periodic *reintegration instants* into the behavior, where the state at the reintegration instant is called the *ground state* of the component.
- A *message* is an atomic data structure that is formed for the purpose of communication, i.e., data transmission and synchronization, among components.
- *Event information* conveys the *difference* of the previous state observation and the current state observation. Messages that contain event information must adhere to the *exactly-once* semantic.
- *State messages* support the *principle of independence* because sender and receiver can operate at different (independent) rates and there is no danger of buffer overflow.

- In real-time systems, the *information pull strategy* should be followed whenever possible.
- *Elementary interfaces* are *inherently simpler* than composite interfaces, because there is *no dependency* of the behavior of the sender on the behavior of the receiver.
- The *services* of a component are offered at its *cluster LIF* to the other components of the cluster. The cluster LIF is an *operational message-based* interface that is relevant for the integration of components into the cluster. The detailed structure, naming, and access mechanisms of the local interface of a component is *intentionally left unspecified* at its cluster LIF.
- Every system is developed according to an *architectural style,* i.e., a set of adopted rules and conventions for the conceptualization, representation of data, naming, programming, interaction of components, semantics of the data, and many more.
- Whenever a communication channel links two systems developed by two different organizations, it is highly probable that some of the properties of the messages that are exchanged across this channel are in disagreement because of the differences in architectural styles.
- A *gateway component* resolves property mismatches and exposes the external-world information in the form of cluster-standard messages at the cluster LIF of the gateway components.
- We distinguish between three parts of a LIF specifications: (1) the *transport specification* of the messages (2) the *operational specification* of the messages and the (3) the *meta-level specification* of the messages.
- Only the *operational specification* of an open component can be provided without knowing the context of use of the open component.
- The information on how the data field of a message is structured into syntactic units is contained in a *message-structure declaration (MSD)*. The MSD establishes the *message variable names* (i.e., the fixed part of the message variable) that point to the respective concepts and specify which part of the unstructured bit vector denotes the *variable part of a message variable*.
- The four principles of composability are (1) independent development of components, (2) stability of prior services, (3) non-interfering interactions, and (4) preservation of the component abstraction in case of failures.
- Multi-levelness is an important organizing principle in large systems.
- The distinction between a *system of sub-systems* and a *system of systems* is based more on organizational than on technical grounds.

Bibliographic Notes

The presented real-time model of computation has been developed over the past 25 years and is documented in a number of publications, starting with *The Architecture of Mars* [Kop85] and further in the following publications: *Real-time Object Model* [Kim94], the *Time-Triggered Model of Computation* in [Kop98], *Elementary versus Composite Interfaces in Distributed Real-Time Systems* [Kop99], and Peri*odic Finite State Machines* [Kop07].

Review Questions and Problems

4.1 How is a *real-time system component* defined? What are elements of a component? How is the behavior of a component specified?
4.2 What are the advantages of separating computational components from the communication infrastructure? List some of the consequences of this separation.
4.3 What is the difference between *temporal control* and *logical control?*
4.4 What is the definition of the *state* of a real-time system? What is the relationship between *time* and *state*? What is the *ground state*? What is a database component?
4.5 What is the difference between *event information* and *state information*? What is the difference in the handling of an *event message* from the handling a *state message*?
4.6 List and describe the properties of the four interfaces of a component? Why are the local interfaces of a component intentionally left *unspecified* at the architectural level?
4.7 What are the differences between an *information push interface* and an *information pull interface*? What are the differences between an *elementary interface* and a *composite interface*?
4.8 What do we mean by the term *architectural style*? What is a *property mismatch*?
4.9 What are the characteristics of a local process I/O interface and the LIF message interface?
4.10 What is the role of a gateway component?
4.11 What are the three parts of a linking interface specification?
4.12 What is the *message-structure declaration (MSD)?* How do we associate the MSD with the bit-vector contained in a message?
4.13 List the f*our principles of composability*?
4.14 What is an integration level? How many integration levels are introduced in the GENESYS architecture?
4.15 Assume that the pressures p_1 and p_2 between the first two pairs of rolls in Fig. 1.9 are measured by the two controller nodes and sent to the man–machine interface (MMI) node for verifying the following alarm condition:

when $(p_1 < p_2)$
then *everything ok*
else *raise pressure alarm*;

The rolling mill is characterized by the following parameters: maximum pressure between the rolls of a stand = 1,000 kp cm^{-2} [kp is kilopond], absolute pressure measurement error in the value domain = 5 kp cm^{-2}, maximum rate of change of the pressure = 200 kp cm^{-2} s^{-1}. It is required that the error due to the imprecision of the points in time when the pressures are measured at the different rolls should be of the same order of magnitude as the

measurement error in the value domain, i.e., 0.5% of the full range. The pressures must be *continuously* monitored, and the first alarm must be raised by the alarm monitor within 200 ms (at the latest) after a process has possibly left the normal operating range. A second alarm must be raised within 200 ms after the process has definitely entered the alarm zone.

1. Assume an event-triggered architecture. Each node contains a local real-time clock, but no global time is available. The minimum time d_{min} for the transport of a single message by the communication system is 1 ms. Derive the temporal control signals for the three tasks.
2. Assume a time-triggered architecture. The clocks are synchronized with a precision of 10 μs. The time-triggered communication system is characterized by a TDMA round of 10 ms. The time for the transport of a single message by the communication system is 1 ms. Derive the temporal control signals for the three time-triggered tasks.
3. Compare the solutions of 4.16.(a) and 4.16.(b) with respect to the generated computational load and the load on the communication system. How sensitive are the solutions if the parameters, e.g., the jitter of the communication system or the duration of the TDMA round, are changed?