Compositional Design of RT Systems: A Conceptual Basis for Specification of Linking Interfaces

Hermann Kopetz
Real-Time Systems Group
TU-Vienna, Austria
hk@vmars.tuwien.ac.at

Neeraj Suri
Dependable Embedded Systems & SW Group
TU-Darmstadt, Germany
suri@informatik.tu-darmstadt.de

Abstract

Composition of a system is driven by the (a) identification and specification of basic components, and (b) specification of the interactions across the components, i.e., the communication linkages, that are needed to communicate value and temporal information across the components from which the aggregate system results. This paper addresses compositional design of distributed Real-Time (RT) systems focusing specifically on the role of specification of linking interfaces (LIFs) across components.

1. Introduction

In many engineering disciplines, large systems are built from prefabricated components with known and validated properties. The composition of discrete components into a meaningful aggregate system is achieved via the interactions defined across the components as realized by dissemination of messages or alternate underlying communication paradigms. Ideally, such component linking interfaces are stable and standardized entities. Desired properties of components are functional cohesion and autonomy, to provide structure and limit the complexity at the system level. The inner mechanisms of components should be encapsulated, such that a system designer can use the components based on their interface specifications. In distributed computer systems, the components interact by exchanging messages across service interfaces. This realizes the emergent services, i.e., those properties of the system of components that are not explicitly part of the components themselves, but come into existence by the interactions among the components. We call a service interface, that link components together, a linking interface (LIF) and the precise specification of these is also a prerequisite for the reuse of components.

Composability, i.e., the formulation of a whole system from parts (components), has developed into an actively researched area. Varied efforts exist, addressing composability from an abstract perspective dealing with theory of aspects and formal composition viewpoints. Other efforts have considered architectural description languages and SW processes [1] including OO design. A substantial effort in composability has considered, among other alternatives, the protocol viewpoint [2], the compositional verification and validation viewpoints [3,4], to system level (both dependability and RT oriented) compositions [5,6], including many other variations of the theme. In spite of a growing body of work in this area, there is still a lack of conformal definitions of (a) the elemental components – SW, system level etc, (b) the interfaces linking the components, and (c) the domain-specific guidelines establishing the emergence of relevant system properties from the compositional process.

Thus, the focus of this paper is on the conceptual basis of composability, establishing the component definitions, operational and meta-level specifications, and especially the specification of linking interfaces in distributed real-time (RT) systems. We emphasize that the thrust is development of the conceptual basis and guidelines to facilitate compositional LIF based RT system design.

In Section 2, we introduce basic concepts on the notion of time and components underlying our work. Sec. 3 explores the notion of composability relevant to RT services and introduces the concepts of operational and meta-level LIF specification. Sec 4 details operational LIF specifications focusing on temporal properties of LIF's. Sec 5 develops meta-level specifications and details LIF state and the related service models. We summarize the ramifications of the developed concepts in Section 6.

2. Basic Concepts

In order to develop our system context, we reiterate basic concepts essential for understanding the paper.

2.1. The Notion of Sparse Time

In the Newtonian model, commonly used for RT applications, time progresses along a *dense* timeline consisting of an infinite set of *instants*. A *duration (or interval)* is a section of the timeline, delimited by two instants. A happening that occurs at an instant (i.e., a cut of the timeline) is an *event*. An *observation* of the state of the world at an

instant is an event. The *time-stamp* of an event is established by assigning the observer's local clock time to the event immediately after the event occurrence. Given the impossibility of perfect synchronization of clocks and the denseness property of real time, there is always the possibility that a single event *e* is time-stamped by two clocks *j* and *k* with a difference of one tick. The finite precision of the global time-base (*dense* time) and the discretization of time makes it impossible to order events consistently based on their global time-stamps. This is solved by the introduction of a *sparse time base* [7], where time is partitioned into an infinite sequence of alternating durations of *activity* and *silence*. The activity intervals form a synchronized system-wide *action lattice*.

Regarding temporal ordering, all events that occur within a specified duration of activity of the action lattice *are* considered to happen *at the same time*. Events that happen in the distributed system in different components at the same global clock-tick are thus considered *simultaneous*. Events that happen during different durations of activity (at different points of the action lattice) and are separated by the required interval of silence (the duration of this silence interval depends among others, on the precision of the clock synchronization [8]) can be temporally ordered on the basis of their global timestamps. The architecture needs to ensure that significant events, such as the sending of a message, or the observation of the environment, occur only during an interval of activity of the action lattice. The time-stamps of events based on sparse time can be mapped on to the set of positive integers. It is then possible to establish the temporal order of events by integer arithmetic. The timestamps of events that are outside the control of the distributed computer system (and therefore happen on a dense timeline) must be assigned to an agreed lattice point of the action lattice by an *agreement protocol.* Agreement protocols are also needed to achieve a system-wide consistent view of analog values that are digitized by more than one A/D converter.

The proposed compositional approach, will utilize the premise that a sparse time base exists across the system.

2.2. Defining Components and Interfaces

Conventionally, a *component* is a self-contained subsystem that can be used as a building block in the design of a larger system. A component provides a desired service to its environment across a well-specified interface. An example of a component is a processor in a computer. A component can have a complex internal structure that is neither visible nor accessible to the user. The services provided by the component, as specified by interfaces, define its usefulness in an aggregate system.

An ideal component should maintain its encapsulation (value and temporal) when used in a larger context. The larger system is constructed from components that can be integrated without violating the principle of composability, i.e., properties that have been established at the component level will also hold at the system level (see Section 3.1). In the context of a distributed real-time system, we consider each discrete computing node as a component [9], with the component-behavior specified in the domains of time and value, (this is in contrast to a discrete software component, which, on its own, does not have a temporal dimension). Thus, we consider a component as a self-contained time-aware computational element with its own hardware (processor, memory, communication interface, and interface to the controlled object) and software (application programs, operating system), which interacts with its environment by exchanging messages across linking interfaces (LIFs).

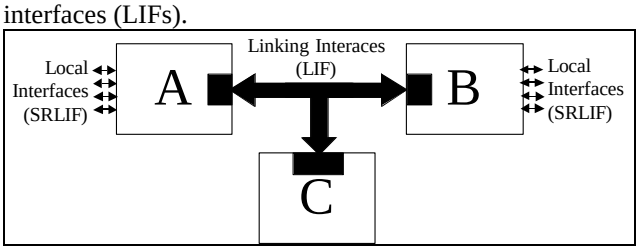

Figure 1: A composition of three components, A, B, C and LIF's

Figure 1 depicts a system, of three components, that realizes emergent services via timely exchange of messages across the LIFs. In order to achieve a complexity reduction, a user should be able to reason about these emergent services by examining only the LIF specifications. The internals of the components, including the related local interfaces, should remain encapsulated; the composition is driven strictly by LIF level interactions.

It should be noted that the term "interface" represents a common boundary between two or more components (or subsystems). Thus, a LIF is characterized by:

(a) Its data properties, i.e., the structure and semantics of the data items crossing the interface.

(b) Its temporal properties, i.e., the temporal conditions that have to be satisfied by the interface for both control and data delivery validity.

The following table further outlines this aspect with an example set of rules and also illustrates the role of the interface as a conformal check of the messages across it. Please note that this represents a local level information check that is valid across a given set of components.

Attribute	Interface Feature Check
Valid	A message is valid if it meets CRC, range and logical checks
Checked	A message is checked if it passes the output assertion
Permitted	A message is permitted with respect to a receiver if it passes the input assertion of that receiver
Timely	A message is timely if it is in agreement with its temporal specifications
Correct	A message is correct if its value and temporal specifications are met
Insidious	A message is insidious if it is permitted but incorrect (requires a global judgment)

An implication of this interface characterizing is the consequent simplification of component interactions. In general, the interface across components can often be a complex one where the stimuli behind component interactions could be time or event triggered. Similarly, the transfer of data can be absolute state or relative. However, by our delimiting the validity of information exchange to be governed with both value and temporal specifications, this simplifies the component interactions to be direct and simple interactions across components. It is worth keeping in perspective, that our driver is real-time system design where such value and temporal information validity specifications are especially relevant.

When analyzing the interactions between a component and its environment, we found it useful to distinguish between four different types of component interfaces. As our focus is on communication level interfaces, we demarcate the interfaces and component interactions into the following discrete categories, namely:

(i) *The Service Providing Linking Interface **(SPLIF)***: Fundamentally, a component must be a unit of service provision. The service is offered to the component environment across a SPLIF. It is the primary interface of a component and is discussed in detail in Section 3.

(ii) *The Service Requesting Linking Interface **(SRLIF)***: In order to meet its specification, a component may request services from other components. The corresponding interface is the SRLIF. A user of the SPLIF may not be aware that a component requests the services of other components via SRLIF's to achieve its objectives.

(iii) *The Configuration Planning **(CP)** Interface:* This is analogous to a global resource manager used to configure - initial and subsequent – components to provide the stipulated services in a specified environment. The access can be sporadic in nature and may not be time-sensitive.

(iv) *The Diagnostic and Management **(DM)** Interface:* The DM interface provides selective access to the (operational) internals of the component for monitoring and diagnostic purposes. Since the DM interface is not exposed during normal component operation, the DM view is not relevant for the user of a component. The DM interface can be used to parameterize a component in order to optimize it for the given task.

The concepts of an SRLIF and an SPLIF are closely related to the concepts of a client interface and a server interface in a client-server architecture or the concepts of a required interface and a provided interface in [10]. It will be shown later that the interactions between an SRLIF and an SPLIF are more differentiated than the interactions in the client-server model. This is the reason why we introduced these new terms. Subsequently, we will use the term LIF to refer to both interfaces.

It is also important to clearly distinguish between the *state of a LIF* and *the state of the component that supports the LIF*. Consider a simple example of an e-commerce server. From the point of view of a particular shopper's LIF the articles that are contained in his/her shopping cart at the chosen instant determine the state of this particular LIF. Since the server may support many concurrent shopping sessions at any particular instant, the state of the server component is formed by the articles in *all* active shopping carts.

Based on the relations between a component and its environment we distinguish between two primary component types, namely:

Closed Component: Consider a component that interacts with its environment only via a single SPLIF, i.e., the output messages that are produced at this SPLIF are only a function of the SPLIF-input messages and the SPLIF state. We call such a component a *closed component*. If, the output messages are a deterministic function (in value and time) of the input messages and the SPLIF state, we call such a closed component a *deterministic closed component*. A typical example of a closed component is a component that implements a *stack* providing the well-known functions of *push* and *pop*.

A special case of a closed component is a *semi-closed component*. A *semi-closed component* has, in addition to the LIF input messages, only one other type of (hidden) input message, a *clock message* that is generated by a synchronized clock denoting the instant of the beginning of a sparse time-granule [12]. While a closed component is not time-aware, a semi-closed component is *time-aware*. A semi-closed component can generate an output message when the time reaches a certain value without having received an input message at its LIF. An example of the simplest semi-closed component is a *clock* that generates periodic output messages after every *n* time-units, where a time-unit represents the granularity of the clock.

Open Component: an open component has one or more SRLIF's that accept inputs from the *natural environment*. The natural environment refers to reality as it exists in nature, as opposed to a digital model of reality [11]. Since the natural environment possesses a boundless number of properties it is difficult to provide for its complete or rigorous specification.

A special case of an open component is a *semi-open component*. This is a component that can exchange data with the natural environment without delegating control to the natural environment. A *sampling system* that periodically looks at the natural environment under the control of the component's internal timer is an example of a semi-open component. An interrupt-driven component that accepts interrupts from the natural environment is an example of an open component. The class of semi-open components is of particular interest for the rest of this analysis.

Connecting closed components to all SRLIF's of an open component can transitively close it. An example of

such a transitive closure is the replacement of a controlled object (the natural environment) by a real-time simulation model of the controlled object. Conversely, the replacement of a real-time simulation model by a real controlled object (e.g., hardware-in-the-loop) can transform a closed component to an open component.

2.3. The RT Context: RT Entities and Images

A distributed RT system is depicted as a set of components connected via a RT communication channel. As mentioned in Sec 2.1, the components are assumed to have access to a global time base of known precision.

It is important to clearly delineate the representation of RT state information. Here we have found the notions of RT entity and RT image introduced in [13, 7] to be highly useful:

RT Entity: A real-time application is modeled by a set of relevant state variables, the *real-time entities* that change their state as time progresses. Example RT entities are the set-point of a control loop or the intended position of a control valve. An RT entity has static attributes that do not change during the lifetime of the RT entity, and has dynamic attributes that change with time. Examples of static attributes include the name, the type, the value domain, and the maximum rate of change. The value set at a particular instant is the most important dynamic attribute. Another example of a dynamic attribute is the rate of change at a chosen instant.

The information about the state of an RT entity at a particular instant is captured by the notion of an *observation*. An observation is an *atomic data structure*

$$Observation = <Name, Value, t_{obs}>$$

consisting of the name of the RT entity, the observed value of the RT entity, and the instant when the observation was made (t_{obs}). A continuous RT entity can be observed at any instant while a discrete RT entity can only be observed when the state of this RT is not changing.

RT Image: A *real-time image* is a *temporally accurate* picture of an RT entity at instant *t*, if the duration between the time-of-observation and the instant *t* is less than *the accuracy interval* d_{acc}, which is an application specific parameter associated with the dynamics of the given RT entity. An RT image is thus *valid* at a given instant if it is an accurate representation of the corresponding RT entity, both in the value and the time domains [13]. While an *observation* records a fact that remains valid forever (a statement about an RT entity that has been observed at an instant), the validity of an RT image is *time-dependent* and is invalidated by the progression of real-time.

2.4. State-Information versus Event-Information

The information exchanged across an interface is either *state information* or *event informatio*n, as explained in the following paragraphs. Any property of *a RT entity* that is observed by a component of the distributed real-time system at a particular instant, e.g., the temperature of a vessel, is called a *state attribute* and the corresponding information *state information.* A *state observation* records the state of a state variable at a particular instant, the *point of observation*. A *state observation* can be expressed by the same atomic triple as seen in Sec. 2.3.

For example, the following is a state observation: *"The position of control valve A was at 75 degrees at 10:42 a.m."* State information is *idempotent* and requires an *at-least once semantics* when transmitted to a client. At the sender, state information *is not consumed on sending* and at the receiver, state information requires an *update-in-place* and a *non-consumable* read. State information is transmitted in *state messages*.

A change of state of a RT entity that occurs at an instant is an *event*. Information that describes an event is called *event information.* Event information contains the *difference* between the state *before* the event and the state *after* the event. An *event observation* can be expressed by the atomic triple *<Name, Value difference, Time of event>*. For example, the following is an event observation: *"The position of control valve A changed by 5 degrees at 10:42 a.m."* Event observations require *exactly-once semantics* when transmitted to a consumer. Events must be *queued on sending* and *consumed on reading*. Event information is transmitted in *event messages*.

Periodic state observations or sporadic event observations are two *alternative* approaches for the observation of a dynamic environment in order to reconstruct *the states and events* of the environment at the observer [14]. Periodic state observations produce a sequence of equidistant "snapshots" of the environment that can be used by the observer to reconstruct those events that occur within a minimum temporal distance that is longer than the duration of the sampling period. Starting from an initial state, a complete sequence of (sporadic) event observations can be used by the observer to reconstruct the complete sequence of states of the RT entity that occurred in the environment. However, if there is no minimum duration between events assumed, the observer and the communication system must be infinitely fast.

3. Specifying LIFs for RT Services

3.1. The Composability Viewpoint

In a distributed system, a given set of components compositionally realize *emergent services* by exchanging messages across LIFs [15]. For example, an n-node system defines new (emergent) services such as group membership, synchronization or agreement.

As a definition, a composition is *the act of combining parts or elements to form a whole* [16]. Thus, *composability* is defined as *the ease of forming a whole by combining parts*. In our context, the *whole* is the system and the *parts* are the components. The key to composition is identifying

and specifying the interactions across the identified components from which the system is desired to be composed. These interactions represent value and time domain information, and are required to be semantically and syntactically compatible. Composability is driven by the stipulated property that the composition is supposed to achieve, e.g., safety or timeliness.

For an architecture to support composability with respect to LIF and RT, it must adhere to the following principles:

(1) Independence: A composable architecture should clearly distinguish between architecture design and component design. Principle one of a composable architecture is concerned with design at the architecture level. Components, based on principles of autonomy, need to be designed independently of each other. However, their utility is obtained if the architecture supports the interface specification of the LIF's during LIF design. The interface data structures must be precisely specified in the value domain and in the temporal domain and a proper *LIF service model* of the component service, as *viewed* by a user of the component, must be available (See also Section 3.2). Only then is the component designer in a position to know exactly *what* to expect *when* from the environment, and *what* must be delivered *when* to the environment by the component across its LIF. This knowledge is a prerequisite for the independent development of the component software, for reusing a component in a new application context, and for reasoning about the composition of components into a system or system-of-systems.

(2) Invariance: Principle two of a composable architecture concerns the design at the component level. The *stability-of-prior-service* principle ensures that the validated service of a component, both in the value and time domains, is not refuted by the integration of the component into an encompassing system-of-systems, i.e., ensuring the invariance of existing properties of components.

(3) Growth: Principle three of a composable architecture is concerned with the performance of the communication system. The integration of the components into a system usually follows a step-by-step composition process. The *performability of the communication system* principle ensures that if n components are already integrated, the integration of the $n+1^{st}$ component will not disturb the correct operation of the n already integrated components. This has stringent implications for the management of the network resources. If the network resources are managed dynamically, then it must be ascertained that even at the *critical instant*, i.e., when all components request the network resources at the same instant, the specified timeliness of all communication requests can be satisfied. Otherwise service degradation, such as response time, could occur sporadically with a rate corresponding to an increase in the number of integrated components.

Example: If a real-time service requires that the network delay must always remain below a critical upper limit (else a local time-out within the component may signal a communication failure) then the dynamic extension of the network delay by adding new components may be a cause of concern. In a dynamic network the message delay at the critical instant (when all components request service at the same instant) increases with the number of components.

(3.1) Resilience: This is a sub-principle if fault-tolerance is implemented by the replication of components, then the architecture and the components should support replica determinism. A set of replicated components is *replica determinate* [17] if all the members of this set have the same encapsulated state, and produce the same output messages at points in time that are at most an interval of d time units apart (as seen by an omniscient external observer). In a fault-tolerant system, the time interval d determines the time it takes to replace a missing message or an erroneous message from a component by a correct message from redundant replicas.

3.2. Operational versus Meta-level Service Specification of LIF's

The LIF service specification is the mediator between a service supplier and the service user. The first principle of composability states that the LIF service specification must be precise in the value domain and in the temporal domain. On the one hand, the LIF service specification *should* be *complete* in the sense that it contains all information required to understand and use the services of the component that are offered at the particular LIF. On the other hand, the LIF service specification should be *minimal* in the sense that it contains only information that is essentially required by the user of the services. The challenge for a LIF designer is to meet both of these somehow conflicting requirements.

In a distributed real-time system based on message exchanges among components, the LIF specification comprises the following parts:

(i) The **syntactic specification** of the messages, i.e., the specification of the data elements that cross the interface. The syntactic specification forms out of the sequence of bits in a message, larger *(information) chunks* (such as a number, a string, a method call, a structure consisting of a combination thereof, or a complex data object such as a picture [10]) and assigns a *name* to each chunk. Although from the view of mechanical processing any name would suffice, a *descriptive name* that establishes a link between the chunk and its meaning helps human understanding. The syntactic specification bridges the gap between the *logical level* and the *informational level* [18].

(ii) The **temporal specification** of the message send and receive instants, e.g., at what instants the messages are sent and arrive, how the messages are ordered, and the rate of message arrival. This information can be formal-

ized if an appropriate model of real-time is available. In non safety-critical applications the temporal specification can be expressed in probabilistic terms.

(iii) The **operational input assertion** specifies an executable predicate on the incoming message (and the interface state) of a component to determine whether the message is permitted at a given time instant.

(iv) The **LIF service model specification**. In many cases the *concept* associated with the *name of a chunk* (the result of the syntactic specification) is not sufficiently precise to cover all aspects of the semantics of the interface data. In these cases it is necessary to specify a *conceptual interface model* that relates the names of the *chunks* to the user's conceptual world and thus assigns a deeper meaning to the *chunks*. It follows that the LIF service model must be expressed in concepts that are familiar to the user of the interface services. The conceptual model bridges the gap between the *informational level* and the *user's level* [18].

There are obvious interdependencies across these different parts of the specification, e.g., the concept of an *observation*, contains an element of each one of the parts. The proposed partitioning aids in structuring the LIF specification into syntactic, temporal and service components to ensure compatibility across these dimensions over the aggregate process of system composition.

We subsume under the term *operational specification* of an interface the *syntactic specification*, the *temporal specification* and the *operational input assertion*. The operational specification is not concerned with an interpretation of the data that crosses the LIF. Consistency of the operational specification of interacting components is a necessary (but not sufficient) prerequisite for the proper operation of the system of components. It can be achieved by forcing all components to adhere to a given architectural style. Consistency of the specifications of the LIF service models of the communicating partners, called the *meta-level specifications*, assures that the meaning of the information chunks in all involved components is in agreement with the user's intent.

Whereas the operational specification must be rigorous, it is often difficult to develop a formal model for the meta-level specification if the information chunks exchanged across an interface relate to concepts of the natural environment (as they normally do). For example, how can we formalize the concept of *temperature* or the concept of *drive at a safe speed*? These are highly subjective terms and explicitly need a defined context for them to be interpreted meaningfully. This qualitative difference in the nature of the operational specification and meta-level specification justifies the clear distinction between these two kinds of specifications.

The demand for minimal cognitive complexity of a LIF has the following consequences:

(i) An interface should only serve a single purpose. If there is a need to interact with a component for different purposes, different interfaces should be provided. For example, a component may support, in addition to a LIF, a *diagnostic interface* for diagnosis and a *configuration interface* for configuring the component into a new environment (see Sec. 2.2). As these three interfaces target different user groups and different views of a component, interfacing would become unnecessarily complicated if all three interfaces were integrated into one [19].

(ii) If multiple users of a sequential LIF can access a component concurrently, then it is desirable to hide this concurrency from the LIF user, since concurrency increases the cognitive complexity of an interface significantly.

(iii) The LIF conceptual model should be structured along a *means-end hierarchy* [20] in order to limit the amount of information that must be dealt with at a selected level of abstraction. This requirement is derived from the characteristic of human cognitive information processing [21]. The proper structure and representation of the LIF service model is crucial for controlling the effort needed to understand the component and for reusing the component in a new context.

As mentioned before, communication across an interface is only successful if the operational specification and the meta-level specification of the interfaces of all communication partners are consistent. If there are syntactic mismatches among the interface specifications of the communicating partners, it is possible to resolve these by connection systems inserted between the component interfaces [15]. These connection systems are not needed, if all components are designed according to the same architectural style. Mismatches that have their cause in conflicts of the LIF service models cannot be resolved by connection systems but require human intervention.

4. Operational LIF Specification

The operational LIF specification covers the syntactic specification of the data items contained in the input and output messages, and the temporal specification of the message send and receive instants. Since the operational LIF specifications govern the exchange of information between computers, they must be precise and formal. Any ambiguity or incompleteness of the operational specification can be the cause of later system failures.

The syntactic specification of data items is a well-understood topic. Standardized interface definition languages, such as the IDL of the OMG, are in wide use for the syntactic specification of data items exchanged across LIFs. In this section we will focus on the temporal specification of LIFs, which in many systems is missing or incomplete. We discuss two alternatives, state-message interfaces and event-message interfaces.

4.1. State Message Interfaces

A state message has the following characteristics:

A) Content: A state message contains state information (see Section 2.4).
B) Flow control: A state message is sent and received periodically at *a priori* known instants that are common knowledge to the sender and the receiver. Flow control is implicit and unidirectional.
C) Delivery method: Every state message must be delivered *at-least* once.
C.1) Error detection: Error detection is performed by the receiver based on the *a priori knowledge* of the instant of message arrival.

State messages are well suited for control applications. Sensors observe periodically the state of the controlled object, control algorithms are calculated, and the setpoints are delivered to the actuators. In order to support these applications effectively, a special interface, the *temporal firewall*, has been designed. A temporal firewall is an operationally fully specified digital interface for the unidirectional exchange of periodic *state messages* between a sender/receiver over a time-triggered communication system. The basic data and control transfer of a temporal firewall interface is depicted in Figure 2.

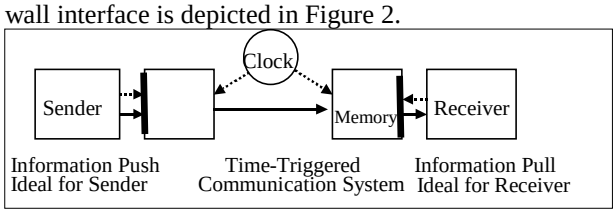

Figure 2: Information/Data flow (solid line) and control flow (dashed line) across a temporal firewall interface.

The Communication Network Interface (CNI) memory at the sender forms the output firewall of the sender and the CNI memory of the receiver forms the input firewall of the receiver. The sender deposits its output information into its temporal firewall (update in place) according to the information *push* paradigm, while the receiver must *pull* the input information out of its CNI (non-consumable read). The transport of the information is realized by a time-triggered communication system that derives its control signals autonomously from the progression of time. It is *common knowledge* to the sender and the receiver at what instants (on a sparse time base) the typed data structure is fetched from the sending CNI and at what instants this data structure is delivered at the receiving CNI by the communication system. On input, these precise operational interface specifications (in the temporal and value domain) are the *pre-conditions* for the correct operation of the application software. On output, the precise interface specifications are the *post-conditions* that must be satisfied by the application software, provided the preconditions have been satisfied by the environment.

Since no control signals cross such a temporal firewall interface, control-error propagation across this interface is eliminated by design. Components that interact via temporal firewalls with the natural environment are thus *semi-open* (see Section 2.2). A semi-open component still maintains a high degree of autonomy. A temporal firewall also eliminates (low-level) concurrency from the interface. The sparseness of the global time establishes a system wide *action lattice*, the lattice points of which are precisely synchronized with the global time. The behavior of a system can be explained by the sequential stepwise progression through this action lattice. This elimination of concurrency from the interface simplifies the understanding and consequent specification of the interface. [21].

4.2. Event Message Interfaces: Perspectives and Limitations of the Client-Server Approach

An event message has the following characteristics:
a) Content: An event message contains event information (see Section 2.4).
b) Flow control: An event message is sent as a consequence of the occurrence of a significant event. The receiver has no *a priori* knowledge at what instant an event message will arrive. Flow control is thus explicit and requires a bi-directional protocol.
c) Delivery method: Every event message must be delivered *exactly* once
c.1) Error detection: Error detection is performed by the sender based on a timeout for an acknowledgment message from the receiver.

Event messages are needed for the communication of sporadic processes that have an *a priori* unknown temporal control pattern. An example of such a process is a sporadic client-server interaction, e.g., a diagnostic investigation of a component after a malfunction has been observed. From the point of view of temporal predictability, communication by event messages in a hard real-time system poses a number of problems:
(a) Restriction of component autonomy: A component that interacts with the natural environment by event messages is an *open* component (see Section 2.2). There are two mechanisms that restrict the component autonomy of open components. First the temporal control of an open component is delegated to the component environment, i.e., is outside the sphere of control of the component proper. Control error propagation from the environment into the interfacing component and further throughout the system is thus possible (in contrast to a *semi-open* component). The second mechanism that leads to a restriction of component autonomy is the need for a bi-directional protocol between a sender and a receiver for the purpose of error detection. Even in a scenario, where there is a unidirectional data transfer, a sender becomes dependent on the receiver because of the bi-directional control transfer [22].
a) Server overload: Since there is no coordination among different clients, it may happen that all clients request a particular service at the same critical instant. Whatever scheduling techniques are selected, some clients may have

to wait until all other clients have been serviced, causing a significant *jitter* in the service provision.

b) Communication system overload: In addition to the uncoordinated server requests (see b), the bi-directional control flow in a multicast scenario can lead to a communication system overload caused by the correlated response messages.

Event message communication requires the provision and careful management of message queues in order to faithfully implement the *exactly-once* semantics of event messages. If no assumptions about the request rate with respect to the service rate are made, there is always the possibility that the queues will grow beyond limit and that messages are delayed past their deadline or will be discarded. Unrestricted event-message communication can thus provide only a best-effort level of service which does not meet the requirements of hard real-time systems.

In this section we have only considered simple client-server interactions among components. In some application, more complicated interactions formed by a sequence of event messages are governed by complex protocols (e.g., transactions). From a temporal predictability viewpoint, currently there are limitations in analyzing these complicated interactions, as even the analysis of simple client-server interactions poses a number of problems.

4.3. An Example

The recently approved OMG Smart Transducer Interface [23] is an example of an interface standard that specifies the operational properties of the SPLIF in the value domain and in the temporal domain and thus establishes the prerequisites for composability (see Section 3.1). A smart transducer (ST) is a hardware/software device that comprises in a compact small unit a sensor or actuator element (possibly both), a micro-controller, a communication controller and the associated software for signal conditioning, calibration, diagnostics, and communication. The ST provides the intended services across interfaces to its clients: the temporally predictable service providing linking interface (SPLIF), the diagnostic and management interface (DM), and the configuration and planning interface (CP). The internal structure and operation of STs remain encapsulated within the ST and are not exposed at the interfaces accessible from the client. Interfaces conforming to this OMG standard have the same form and behavior for the wide array of sensor and actuator nodes in the various engineering disciplines. A user of smart transducers have to cope only with one single generic smart transducer interface specification for the multitude of existing and new sensor and actuator types.

It is assumed that every ST contains a synchronized *physical clock* for measuring time. The synchronization of the clocks is performed as a background service by the communication system. It is possible to relate the clocks in the ST nodes to an external time standard, such as GPS time. The standard also contains a section that deals with the uniform representation of time.

The information transfer between an ST and its client is achieved by sharing information that is contained in an encapsulated ST internal *interface-file system* (IFS), as depicted in Figure 3. This IFS is at the core of the conceptual model, and has been optimized to fit into 8-bit micro-controllers commonly used in smart transducers.

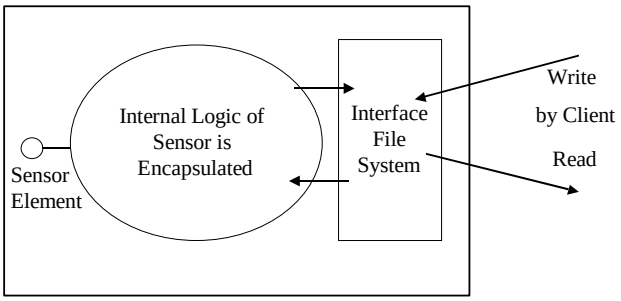

Figure. 3: Interface File System in a Smart Transducer.bylle

The IFS provides the structured un-interpreted name space required for the operational exchange of information between a ST and its clients. A ST node contains in dedicated IFS files [23] the information for all three interfaces: the temporally predictable service providing linking interface (SPLIF), diagnostic and management interface (DM), and configuration and planning interface (CP).

The structure and the meaning of the data items in the IFS files are only intelligible if some meta-level specification about the particular IFS is known. Since an ST has only a very limited storage capacity, this metadata describing the semantics of the ST files resides outside the ST at a web site associated with each transducer type. The meta-data information is essential for the development of applications by a “human design process” or by an “automated design process”. At the moment, this meta-data is described by an ad-hoc combination of “structured English” and XML meta-data tags.

In a smart transducer system we distinguish between three kinds of communication services, called a *multi-partner round*, a *master-slave (MS) round*, and a *broadcast round*. The periodic multi-partner rounds are used to implement the *temporal firewall* of th*e SPLIF*. The sporadic MS rounds are used to implement with best-effort temporal quality the two *client-server* interfaces, *the diagnostic interface* and the *configuration and planning interface*. They operate concurrently with the periodic multi-partner rounds. It is thus possible to look at the diagnostic information in an IFS file or to plan for the on-line reconfiguration of a sensor without interfering with the predictable real-time service provided at the SPLIF. *Broadcast rounds* are used to implement operations that must be executed by all nodes of a cluster.

5. Meta-level LIF Specification

The meta-level LIF specification assigns meaning to the information chunks exchanged between two communicating LIFs at the operational level. It thus bridges the gap between the information chunks formed at the syntactic level and the user's mental model of the service provided at the interface. Central to this meta-level specification is the LIF service model.

5.1. LIF Service Model

The LIF service model interprets the information chunks that are formed by the syntactic specification. This interpretation will be qualitatively different for closed components and open components (see Section 2.2).

The LIF service model for a *closed* (or semi-closed) component can be formalized. The relationship between the LIF inputs and LIF outputs depends on the discrete algorithms implemented within the component. There is no input from the natural environment that can bring unpredictability into the component behavior. The sparse time-base is discrete and supports a consistent temporal order of all events.

The LIF service model for an *open* (or semi-open) component is fundamentally different since it must deal with the inputs from the natural environment. Since the natural environment is not rigorously defined, the interpretation of these 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 world; otherwise the description will not be understood. The following discussion will focus on LIFs of open components, since the systems we are interested in must interact with the natural 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 utilized 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 designer) needs consider system-wide effects of an interaction with a component. The LIF service specification must address all direct and indirect effects interactions across the LIF and beyond the boundary of the interfacing component.

The LIF service model of a component is different from the model describing the algorithms implemented within a component. The LIF service model is goal oriented, while the algorithmic model is process oriented. A goal-oriented model specifies the intended state, while a algorithmic model specifies the actions that must be taken in order to reach this intended state [24] p.223.

5.2. Perspectives on LIF State behavior

The LIF service specification is required to be self-contained and must provide a complete description of the component behavior as seen from the viewpoint of a LIF user. From the user's point of view, it is helpful to distinguish between the following two types of LIFs:

(i). **Stateless LIF**: This is a LIF where a response to a service request depends only on the parameters of the request and is independent of the time of the request.

(ii). **Stateful LIF**: This is a LIF where a response to the service request depends not only on the parameters of the request but also on the instant when the request is delivered, i.e., the same request may lead to different results, depending on the history of the component or the state of the environment at the instant of the request. Examples are the sampling of a sensor or a query to a database.

Since a LIF with state is the more general concept (a stateless LIF can be viewed as a special case of a stateful LIF), the subsequent discussion focuses on stateful LIFs.

Taking this view it follows that the notions of state and time are inseparable. If an event that updates the state cannot be assigned to a well-defined tick of a global clock, then the notion of a system-wide state becomes diffuse. It is not known whether the state assigned to a clock tick includes this event or not. The sparse time-base introduced in Sec 2.1 makes it possible to define a system-wide notion of time, which is a prerequisite for an indisputable border between the past and the future, and thus the definition of a system-wide state. If there is no global sparse time-base available, one often recourses to a model of an abstract time that is based on the order of messages sent and received across LIFs. If the relationship between the physical time and the abstract time remains unspecified, then this model is imprecise whenever this relationship is relevant. It may be difficult to determine in such a model the precise state of a system at an instant of physical time.

For the designer and user of a LIF it is consequential to know what actions of the past have relevance for the future. Since there may be some past actions which are more relevant than other past actions, it can be advantageous to partition and classify the state according to the criticality of the state for the future behavior of the component. In case of a component restart in an emergency, the critical state can then be recovered swiftly in order to provide the services required for the safe operation of a system with-

out a long delay. In some applications, it may be possible to restart from a safe initial state. For example, in a traffic control system such a safe initial state may be "traffic lights *red* in all directions".

The concept of *initial state* merits further consideration, particularly for open components. After the startup of a component, what should be its initial LIF state? The initial state of a LIF must be consistent with the state of the environment at the instant of startup [7], p.102. Since the state of the environment is observed by sensors, the instrumentation must be designed to observe directly or indirectly all state variables of the environment that are relevant for the future operation of the system, as seen from the viewpoint of the LIF. In discrete processes where it is not possible to observe the state of all relevant state variables at all instants, it might be necessary to introduce designated start-up intervals, where the environment is in a state of intermediate stability and can be observed by the available sensors. The precise definition and capture of the state of a LIF after startup requires careful planning during system design.

A final remark relates to the concept of state in the widely used notion of *a software component*. As outlined above, the definition of state requires a model of time, since state and time are intertwined. Since a software component per se does not support any notion of time it is only consequent that Szyperski [25, p.30] requires that a software component has no persistent state. The state must be viewed as external to the software component. This notion of a software component is thus fundamentally different from the component notion introduced in Section 2.2 of this paper.

6. Conclusions

In this paper we have proposed separating the operational and meta-level specifications of a linking interface. Such a separation leads to the development of a clearer understanding of each category. In a real-time distributed system, the operational specification must be precise in both the value and temporal domains. It has been shown that the temporal firewall concept provides the capability to provide a precise temporal specification of a LIF. It is difficult to provide such a precise temporal specification if the client-server model is used. The meta-level specification assigns meaning to the information chunks formed at the operational level by providing a LIF service model. This LIF service model should be presented in a form that considers the cognitive constraints of the system engineer. The LIF service model will be informal if the component interacts with the natural environment.

7. Acknowledgements

We acknowledge support by the IST projects DSOS and Next TTA. We are grateful for the interesting interactions with Cliff Jones and Nick Moffat from DSOS.

8. References

[1] A. Jhumka, M. Hiller, and N. Suri, "Component Based Synthesis of Dependable Embedded SW". FTRTFT 2002.

[2] M. Hiltuen, and R. Schlicting, "An Approach to Constructing Modular FT Protocols". Proc. of SRDS, 105-114, 1993.

[3] E. Juan, and T. Tsai, "Compositional Verification of High Assurance Systems", Kluwer Press, 2000.

[4] P. Sinha, and N. Suri, "On Simplifying Modular Specification and Verification of Distributed Protocols", Proc. of HASE 2001.

[5] Ptolemy Project: ptoloemy.eecs.berkeley.edu, 2002.

[6] J. Sifakis, "Scheduler Modeling Based Controller Synthesis Paradigm". FTRTFT Invited Talk. 2002.

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

[8] H. Kopetz, "Sparse Time versus Dense Time in Distributed Real-Time Systems". Proc. ICDCS, 1992.

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

[10] J. G. Wijnstra, "Components, Interfaces and Information Models within a Platform Architecture". Lecture Notes on Computer Science 2186, Springer Verlag: 25-35, 2001.

[11] G. Rosen, Discrete Systems, 1985.

[12] K. H. Kim and H. Kopetz, "A Real-Time Object Model RTO.k and an Experimental Investigation of its Potential", Proc. COMPSAC, 1994.

[13] H. Kopetz and K. H. Kim, "Temporal Uncertainties in Interactions among Real-Time Objects", Proc. SRDS, 1990.

[14] F. Tisato and F. DePaoli, "On the Duality between Event-Driven and Time Driven Models". Proc. of 13th. IFAC DCCS, 1995.

[15] C. H. Jones, H. Kopetz, et al., "Revised Conceptual Model of DSOS", University of Newcastle upon Tyne, Computer Science Department, 2001.

[16] Webster "Encyclopedic Dictionary", 1989.

[17] S. Poledna, "Replica Determinism in Fault-Tolerant Real-Time Systems", Technical University of Vienna, 1994.

[18] A. Avizienis, "The Four-Universe Information System Model for the Study of Fault Tolerance", Proc. FTCS-12, 1982.

[19] A. Ran and J. Xu, "Architecting Software with Interface Objects", Proc. of the Eight Israeli Conference on Computer Systems and Software Engineering, 1997.

[20] K. J. Vicente and J. Rasmussen, "Ecological Interface Design: Theoretical Foundations." *IEEE Transactions on Systems, Man, and Cybernetics*, 22(4): 589-606, 1992.

[21] J. Reason, "Human Error", Cambridge Univ. Press, 1990.

[22] H. Kopetz, "Elementary versus Composite Interfaces in Distributed Real-Time Systems", Proc. of ISADS, 1999.

[23] OMG 2002: Object Management Group Standards, 2002

[24] H. A. Simon, "Science of the Artificial", MIT Press, 1981.

[25] C. Szyperski, "Component SW", Addison Wesley, 1998.