

Interface Descriptions for Embedded Components

Bernhard Schätz
Fakultät für Informatik, TU München,
schaetz@in.tum.de

Abstract

Embedded software systems are increasingly constructed using interacting functionalities, leading to the development of networks of communicating software components distributed over connected processors. At the same time, the growing complexity of the functionality as well as the increase in variations caused by product lines requires a modular development process. As repeatedly witnessed, e.g., in embedded automotive applications, without a suitable notion of interface specifications, integration faults are often only detected after deployment. Here, timed interface specifications can be used to simplify a modular development process, including mechanic checks for consistent modular composition.

1 Introduction

When dealing with real-time software, the value of a variable or the occurrence of an event depends essentially on the timing of that value or occurrence; e.g., a measurement of the position of a crank shaft is only meaningful shortly after the measurement; similarly, detecting the pressing of a button is only meaningful shortly after this event. Therefore, in the following, *signals* in form of a timed functions – i.e., a function over time values – are used to combine the time and value dimension of a variable or an event; e.g., a signal describing a variable is a function mapping a time value to the value of the variable at that time.

When analyzing timing properties of a real-time system, a central aspect is the timeliness of the exchange of signals: Using an outdated value of a variable, e.g., the position of the crank shaft, may lead to the computation of a wrong value, e.g., the time of ignition; analogously, missing an event, e.g., the firing of a crash sensor, may lead to a wrong reaction, e.g., leaving an airbag unfired. Thus, an integral part of the functional correctness of a real-time system is the *timeliness of production and consumption of signals*.

As in general the validity of a signal is characterized by time points or intervals, it can be conveniently described by timed automata [1]. In the following we use *schedules of signals* as a special form of timed automata to describe such signals. Since embedded software is generally built upon variants of periodic behavior, we use *periodic behavior as the elementary blocks* of schedules; by using parallel and sequential composition, we support more complex forms like *parallel schedules* and *mode-based schedules*.

Due to the increasing size of embedded systems, modularity of design has become a major issue in the development of deeply embedded systems. In hard real-time systems, the timing of signals is an integral aspect of the interface of a component. Therefore, we introduce ‘interface types’ for embedded components including the *compatibility of the timing of signals*. Similar to type consistency rules for syntactic compatibility, we supply a notion of *mechanically checkable interface compatibility* accessible to standard model checkers for timed automata.

1.1 Motivation

Our approach aims at the problems occurring during the *integration phase of embedded real-time components*. Currently, integration is one of the problematic phases in domains like avionics or automotive. According to the state of the art, (timing) compatibility of embedded components is often established by testing, with varying degrees of success. Present approaches, e.g., AUTOSAR [8] address this issue by introducing syntactic notions of component interfaces as well as technical infrastructure to ease modular implementation and integration of components.

For a more substantial improvement, it is essential to *include functional timing aspects* with interface descriptions of components and to *move compatibility issues from the integration to the design phase*. To ensure timing compatibility of hard real-time embedded systems during the constructive phase, similar to traditional type consistency, we *add the timing aspects* of the communication as part of the interface speci-

Direction	Id	Class	δt	Byte	Bit	Name	Function	Length
In	0x003	cyclic	100	0	6	ZV_SCHLA	Central Locking 00 (Hold), 01 (Lock) 10 (Unlock), 11 (Invalid)	2
In	0x020	cyclic	100	0	0	BATT	Voltage 5V – 18V, 0.1 V	8
Out	0x6fe	sporadic		0	0	B_LOW_SEAT	Low volt. for seat mov.	1
					1	B_LOW_KEY	Low volt. for lock.	1
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 1. Technical Description of Control Unit Interfaces

fication of the components, *ensure the compositional compatibility* between components based on the architecture of the system, and finally *establish the interface compatibility* between the interface of a component and its internal realization. As all these steps are performed during the constructive phases of the development process, costly design errors are detected earlier, leading to increased efficiency and quality.

Using timed interfaces allows to factor out the verification of the timing aspects of the communication by means of abstraction. Similar to traditional type checking, analysing timing consistency allows to establish a basic soundness property of a system (‘no timing conflicts in communication’). Additionally, the checking procedure can be mechanized to support automatic verification of the timing consistency, using, e.g., temporal model checkers like UPPAAL [3].

1.2 Signal-Based Communication

In a *state-based* approach, communication is performed via *shared variables*. It is commonly used to model the computational aspects of embedded behavior (e.g., compute the amount of fuel injected), related to the control-oriented aspects of the system (e.g., engine control). Generally, update-frequencies for variables are introduced, leading to a discrete-time model of computation. However, shared variables provide little means of synchronization: unless the value of a variable is changed by the writing process, the reading process cannot detect a write action (e.g., if the same value is written again). This model is most used in tools like ASCET-SD [6].

In contrast, in a *event-based* approach, communication is performed via *message-exchange*. Here, communication events support direct synchronization: the receiving process waits for an input event to occur until the sending process performs an output event. To ensure timeliness of interaction, generally, timers are used. The event-based approach is commonly used to model the interactive behavior (e.g., control window

movement), related to the reactive aspects of the system (e.g., chassis electronics). It is used in tools like Rhapsody for MicroC [9].

In the following, we use a *signal-based* approach, which allows to treat both approaches as special cases. In the signal-based approach, communication is achieved by exchange of signals. Therefore, in contrast to a variable-based approach, explicit synchronization can be used when analyzing the timeliness of communication. Furthermore, signals remain available until invalidated. Thus, in contrast to a message-based approach, a signal remains accessible until it is consumed by the receiver or invalidated by the sender. This model supports an integrated description for state-based and event-based communication as the main paradigms used in embedded real-time systems.

Signal-based behavior can be formalized by a continuous-time version of the model used in [11], using partiality to explicitly describe the consumption of signals.

2 Timed Interface Descriptions

In this section, we introduce *schedules* as the basic abstraction formalism to support the mechanical analysis of timing properties of an embedded system. Communication schedules have been traditionally used to describe and coordinate communication via shared media (e.g., CAN, TTP). Table 1 shows a tabular description of a CAN schedule for an automotive electronic control unit [10]. While there schedules describe a technical implementation, here they are a part of the interface description of components on the level of the logical architecture. By adding schedules to an interface description, it does not only describe static aspects (i.e., name and type of ports) of a component, but also includes the timing of its interaction (i.e., a when signal must be sent/received).

The notion of a *port schedule* covers basic timing properties found on the technical level. However, the timing aspects of a signal do not only depend on the

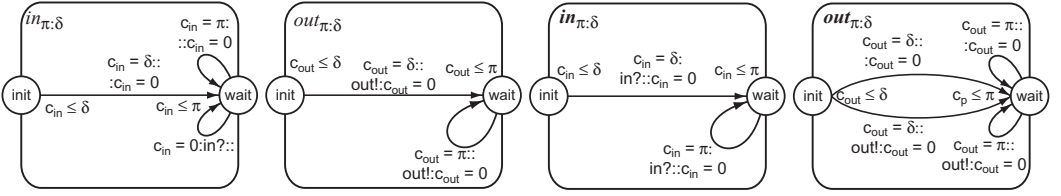


Figure 1. Periodic Timed (π, δ) Event Input, State Output, State Input, and Event Output Port

period of a signal, but also on the *direction* (i.e., input or output) of the port the signal is associated with as well as the *functionality* of the signal (i.e., state signal vs. event signal). Therefore, *signal schedules* are introduced as an adaption of port schedules especially suitable for embedded systems. Based on signal schedules, we introduce *range schedules* for time ranges, *interface schedules* for a combination of port-based schedules, and *mode schedules* for complex behavior based on operation modes.

2.1 Port Schedules

Embedded software is generally built upon periodically activated tasks, invoked unconditionally (e.g., speed measurement activated every 500 ms) or whenever an event occurred within a certain period (e.g., ignition time checked every 2 ms to fire on a crank-synchronous signal). Thus, we use *periodic signals* as the basic elements to describe the (abstracted) behavior of a system. In order to ensure the timeliness of production and consumption of signals, in most cases it is not necessary to look at concrete values of the communicated signals; timing properties often depend on classes of values. For simplicity, in the following, we only check for absence/presence of a signal.

In the most simple form, a communication schedule for a component can be defined independently for all ports of the components. A standard behavior consists of repeatedly performing an interaction; the delay between those equidistant interactions is called *period*. Additionally, the start of the periodic interaction offset by a *phase*. To describe this basic interaction, a *port schedule* is associated to a port, with period π and phase δ . Intuitively, a period-phase port schedule requires that an interaction may only take place at points t in time with $t = n \times \pi + \delta$ for every $n \in \mathbb{N}$.

The left-most automaton in Figure 1 shows the behavior associated to an input port in with schedule (π, δ) , carrying an event-based signal. The automaton in uses a time variable c_{in} to formalize the timing conditions defined by the schedule. Location $init$ characterizes the state when the schedule is still delay-

ing interactions ($c_{in} \leq \delta$) as defined by phase δ ; furthermore, location $wait$ characterizes the state when the schedule is the phase of repeatedly performing an interaction ($c_{in} \leq \pi$) as defined by period π . Since those port schedules are used in a type-like fashion to construct more complex schedules, the corresponding automata define interface locations (e.g., $init$ and $wait$) for activation and termination. The transitions from $init$ to $wait$ correspond to the first (possible) interaction occurring at time δ . Each transition – consisting of precondition, input, output, and postcondition – corresponds to either an input of an event signal ($c_{in} = \delta : in? :: c_{in} = 0$) at time point δ , or the absence of such an input event ($c_{in} = \delta :: c_{in} = 0$) at that time. Similarly, the feed-back loops of the state $wait$ correspond to the repeated (possible) occurrence of an input event at each period.

The second-left automaton of Figure 1 characterizes the behavior of a state output port out with schedule (π, δ) . It is defined similarly to the event input port, substituting input actions by output actions (e.g., $c_{out} = \pi :: out! : c_{out} = 0$). As the interactions are required to occur, transitions to skip interactions (e.g., $c_{out} = \pi :: c_{out} = 0$, resp.) are missing.

2.2 Signal Schedules

The interaction occurring at a port does not only depend on the timing parameters of a schedule (i.e., π and δ), but also on the direction of the port (input or output) as well as the kind of signal communicated (state-signal or event-signal), resulting from the functional aspect to the distinction between state-signals and event-signals. While state-signals generally correspond to *values* used in the *computational* part of the behavior, event-signals often correspond to *commands* used in the *reactive* part of the behavior.

To that end, the concept of *signal schedules* as a special form of port schedules is introduced. A signal schedule covers the *functional aspect* of a signal; i.e., a signal schedule also addressed the question how a signal is used when defining the behavior of a system. We distinguish between

a **state input port** used as input for computations; it must always hold a valid signal when the component can read that port.

an **event input port** used to trigger reactive behavior; it may hold a valid signal when the component can read that port.

a **state output port** used as output of computations; it must always hold a valid signal when the component can write that port.

an **event output port** used to communicate events; it may hold a valid signal when the component can write that port.

The behavior of state input ports and event output ports is shown in the right side of Figure 1.

Obviously, ports with different port schedules may be compatible: an event input port with higher period can be linked to an event output port with a lower frequency without loss of information. In contrast, even ports with the same port schedule are not compatible if using different signals: an event output linked to a state input port may lead to inaccurate computations if the state signal is updated less frequently by the sender than assumed by the reader. This different nature of signals is reflected in approaches like CARTRONIC [7]. There, signals are distinguished concerning whether they are issued or required as well as whether they represent a command or an information.

2.3 Range, Interface, and Mode Schedules

Signal schedules use a precise timing of interaction, leaving little implementation freedom. For more flexibility, less rigid schedules are needed, defining *ranges of time*. As in practical applications often a signal is valid for a certain duration, *range schedules* – for both state and event signals – require that signals are exchanged between the beginning and the end of a period rather than exactly at its beginning.

So far, schedules describe interactions occurring at a single port of a component. As, in general, we are interested in describing the complete interface when analyzing its interactions, we use *interface schedules* to describe a component communicating via multiple ports according to their signal schedules. To formalize the definition of an interface schedule, parallel composition of timed automata (with interface locations) is used. The activation of the interface automaton through its entry locations corresponds to the joint activation of the signal automata through their entry location connectors; their joint deactivation corresponds to the deactivation of the component.

Based on the notion of simple periodic schedules, more complex schedules are constructed using modes. Especially embedded systems often make use

of modes of operation (e.g., configuration mode, degraded mode). While the behavior of such a system is mostly rather homogeneous within a mode of operation, it differs substantially between those modes, including their schedules (e.g., sparse interaction in the stand-by mode, dense interaction in standard mode). Using hierarchic forms of timed automata allowing arbitrary mixtures of parallel and sequential compositions, *mode schedules* are formalized; those mode schedules cover, e.g., the interaction schedules of Mode Transition and Data Flow Diagrams [2].

3 Consistency

By explicitly adding timing information to interface descriptions, we can check whether the timing of the interactions of two components is compatible, and whether a component meets the timing conditions imposed by its interface. To that end, the notion of *consistency* of timed interfaces is introduced, to avoid loss or lack of signals when composing components, corresponding to the notion of consistency used in [11]. Since we are interested in the availability of valid signals, we separate different causes of inconsistency, distinguishing loss of signal (i.e., an event signal not read in time and thus lost by overwriting it) and lack of signal (i.e., a state signal not issued in time causing use of outdated information).

3.1 Compositional Consistency of Interfaces

Signal schedules can be interpreted as contracts between a component and its environment:

State input port: The environment provides a signal whenever the component requests a signal according to the schedule.

State output port: The component provides a signal whenever the environment requests a signal according to the schedule.

Event input port: The component consumes a signal whenever the environment supplies a signal according to the schedule.

Event output port: The environment consumes a signal whenever the component supplies a signal according to the schedule.

While event input ports and state output ports offer guarantees about the signals consumed or produced without imposing requirements about the environment, state input ports and event output ports require the environment to produce or consume signals in time. If the environment does not respect this, *read loss* (i.e., receiver expects sender to provide a valid signal but sender does not perform a signal update) or *write loss* (i.e., sender expects receiver to consume an event but receiver ignores it) may occur.

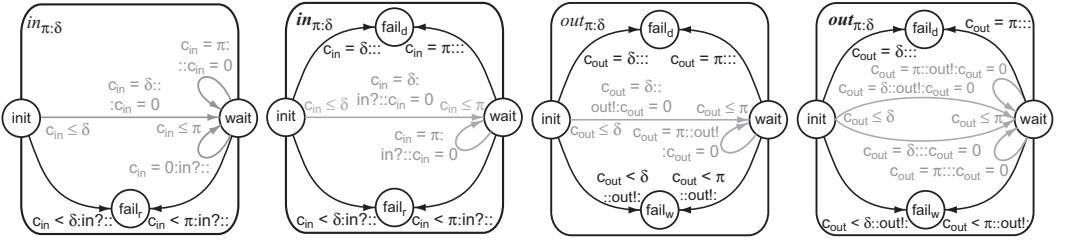


Figure 2. Completion of State and Event Signals

To ensure that no information is lost between communicating components, we use the notion of *compatibility* of components, stating that the contracts defined by their interface schedules agree on those ports linked by channels. To check the compatibility of interface schedules, we extend their descriptions to *detect possible incompatibilities*. As shown in Figure 2, we add failure locations $fail_r$ and $fail_d$ to the timed automata describing the interface schedules of an input state and an input event port:

Delay Failure Location $fail_d$ indicates that a valid signal was required by the component but not supplied by the environment. A delay transition is activated if the component unsuccessfully tries to *enforce a read* action. It connects every location to $fail_d$; its triggering condition is the negation of the conjunction of all delay transitions enabled in this location (including all implicit transitions represented by the invariant condition of the location). As shown, e.g., in the second automaton of Figure 2, location $init$ is extended with a delay transition $c_{in} = \delta \dots$. Its (simplified) triggering condition $c_{in} = \delta$ corresponds to the negation of $c_{in} < \delta$ of the implicit transition.

Read Failure Location $fail_r$ indicates that a valid signal was supplied by the environment but not consumed by the component. A read transition is activated when the component unsuccessfully tries to *enforce a delay* action. Analogously to the delay case, for each location, a read transition (i.e., a transition with synchronization label $in?$) is added from this location to $fail_r$; its triggering condition is the negation of the conjunction of all read transitions enabled in this location.

Similarly, we add failure locations and corresponding transitions to extend output schedules:

Write Failure Location $fail_w$ corresponds to those situations when a valid signal is required by the environment but not supplied by the component.

Delay Failure Location $fail_o$ corresponds to those situations when a valid signal is supplied by the

component but not consumed by the environment. This canonical extension is not restricted to state/event ports but applies to arbitrary timed automata with directed synchronization labels (i.e., labels that are either read or write labels) including range, interface, and mode schedules. Signal loss may occur if the extended schedule can reach a failure location. To check for compatibility of components, we construct the extended schedules of all components, compose these schedules in parallel using *symmetric synchronization* on read/write labels, and check whether an interface schedule reaches a failure location if executed in combination with the other interface schedules.

Loss or lack of information occurs if either the sending or the receiving component can reach a failure location. Based on the kind of failure location, furthermore the class of the error (possible loss of event in case of $fail_r$ in receiver/ $fail_d$ in sender, possible lack of data in case of $fail_d$ in receiver/ $fail_w$ in sender) is identified. Note that not all failure locations represent a relevant loss of information when linking ports. E.g., a state signal updated more often by the sender than required by the receiver may not impose a problem; then it suffices to check whether the reached read failure location is contained in a state input port schedule.

3.2 Abstraction Consistency of Interfaces

To show that a component meets the requirements imposed by its interface description, we ensure that

- the component only requires to read some input from the environment when the interface description requires to read that input
- the component at least guarantees to read some input from the environment when the interface description guarantees to read that input
- the component only guarantees to produce some output to the environment when the interface description guarantees to produce that output
- the component only requires to produce output to the environment when the interface description requires to

When considered from the functional point of view, a component behavior observes events at least as often as defined by the interface description, but it does not assume more updates of state signal as defined by the interface description; similarly, it updates state signals at least as often as defined by the interface description, but does not produce events more often than defined by the interface description.

To check for consistency between a component and its interface description, we extended the interface description as defined above and compose it in parallel with the automaton describing the behavior of the component, using *direct synchronization* on the read and write actions. The behavior of the component is inconsistent with the interface description if a failure location is reached. Analogously to composition consistency, based on the kind of failure location the class of the error is identified (possible loss of observed event in case of $fail_d$ in input signal, possible lack of observed state in case of $fail_r$ in input signal, possible lack of provided data in case of $fail_d$ in output signal, possible loss of provided event in case of $fail_w$ in output signal). Note that abstraction consistency establishes a compositional refinement relation between an abstract and a concrete behavior. Thus it can be used to modularize the consistency check of hierarchical systems by establishing a refinement relation between the interface description of a component and the interface description of its sub-components.

4 Conclusion and Related Work

State-of-the-art descriptions of interfaces of (embedded) software components focus on static information (e.g., type, range) of exchanged information. To effectively cope with system faults arising during integration, however, timing information must be added to interface descriptions. By furthermore adding the functional dimension of event and state signals, typical faults (loss of events, lack of state updates) can be mechanically identified at the level of the logical architecture, ensuring interface descriptions that provide the necessary information for a sound integration on the technical level with process- and bus-schedules.

Behavioral interface descriptions, notions of compatibility and refinement have been investigated in approaches like [4], or [5], however focusing on blocking communication in general. In contrast, here we focus on event- and signal-based communication and avoidance of lack or loss of signals. Canonical completion of temporal automata has, e.g., been used in [3], however, focusing on simulation relations between abstract and a concrete timed automata. Here, based on the functional dimension of signal schedules,

we use a finer form of completion, suitable for composition and abstraction consistency.

Similar to these approaches, the approach presented here focuses on the mechanical support by standard model checkers, specifically addressing the requirements of embedded systems. To automatically perform its consistency analysis steps, [3] is used, translating schedules to synchronized timed automata and describing the consistency conditions as safety properties ensuring the unreachability of failure locations; inconsistencies are described graphically by executions leading to failure locations.

Acknowledgment

We thank Peter Braun, Ulrich Freund, and Jan Philipps, for fruitful discussions on the state-of-the-art in automotive software engineering.

References

- [1] Rajeev Alur and David L. Dill. Automata for modeling real-time systems. In *Proc. of ICALP*, page 322–335, 1990. volume 443 of LNCS.
- [2] Andreas Bauer, Manfred Broy, et al. AutoMoDe—Notations, Methods, and Tools for Model-Based Development of Automotive Software. In *SAE 2005 World Congress*, 2005.
- [3] Gerd Behrmann, Alexandre David, and Kim G. Larsen. *A Tutorial on Uppaal*. Department of Computer Science, Aalborg University, Denmark, November 2004.
- [4] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *European Software Engineering Conference/ACM SIGSOFT Foundations of Software Engineering*, pages 109–120, 2001.
- [5] Luca de Alfaro, Thomas A. Henzinger, and Mariëlle Stoelinga. Timed interfaces. In *EMSOFT Embedded Software*, pages 108–122, 2002.
- [6] ETAS GmbH, Stuttgart, Germany. *ASCET-SD User's Guide Version. 4.2*, 2001.
- [7] P. Torre Flores, A. Lapp, et al. Integration of a Structuring Concept for Vehicle Control Systems into the Software Development Process using UML Modelling Methods. In *SAE International Congress 2001*, 2001.
- [8] Harald Heinecke, Klaus-Peter Schnelle, et al. AU-Tomotive Open System ARchitecture. Whitepaper, www.autosar.org, 2004.
- [9] I-Logix Inc. *Rhapsody in MicroC User's Guide*, 2001.
- [10] Bernhard Schätz, Tobias Hain, et al. CASE-Tools for Embedded Systems. TUM-I 0309, TU München, 2003.
- [11] Bernhard Schätz and Christian Salzmann. Service-Based Systems Engineering: Consistent Combination of Services. In *Proceedings of ICFEM'03*. Springer, 2003. LNCS 2885.