Signals, States, Events, and Modes

Peter Braun[1] Jan Philipps[1] Bernhard Schätz[2]

[1] Validas AG
Lichtenbergstr. 8
85748 Garching, Germany
{braun,philipps}@validas.de

[2] Technische Universität München
Boltzmannstr. 3
85748 Garching, Germany
schaetz@in.tum.de

For embedded control software – especially in form of networks of interacting components – correctness of inter-communication is a key issue. While architectural compatibility notions do not capture the necessary dynamic aspects, analysis based on full behavioral models does not yet scale for practical applications. Here, communication obligations offer a trade-off between completeness and usability. For describing embedded control software, these obligations essentially depend on signal-related issues like state/event communication as well as application-related issues like modes of operation. To motivate the use of communication obligations, we introduce the concepts of state and event signals as well as modes of communication; furthermore, their formalization and analysis using temporal model checking is sketched.

1 Introduction

The increasingly complex functionalities of embedded systems have led to the use of software components, cooperating by exchange of information and often implemented on networks of ECUs. To ensure the construction of reliable systems, as a first step typically component-oriented approaches are used, such as those found in state-of-the-art modeling languages and their corresponding CASE tools such as Simulink. As illustrated in Figure 1, these approaches use structural concepts, describing the components (e.g., Vehicle Status Manager, Door Control Unit), their communication interfaces (e.g., BATT, DIAG ports) as well as their connections. Additionally, behavior is described, e.g., using data flow interpretation or some kind of state transition diagrams.

Independent of whether modeling formalisms or standard programming languages are used to describe the behavior of a component, generally some middleware layer supports the deployment of these components to the target ECU. Here, the component-based approach eases the deployment and ensures their (static) communication compatibility within frameworks like AUTOSAR [5] and EAST/EEA [8].

However, *static compatibility of interaction*, e.g., type correctness of messages exchanged between components, is only capable of detecting gross — but important — cases of communication incompatibility. To detect additional sources of defects in the development process during functional design, in academic approaches the verification of (an abstracted version of) the behavior of networked components has been investigated. By adding the full description of the interactions of components, arbitrary properties of these networks can be analyzed. However, these approaches in general require sophisticated analysis techniques, and do not readily scale to real-world applications.

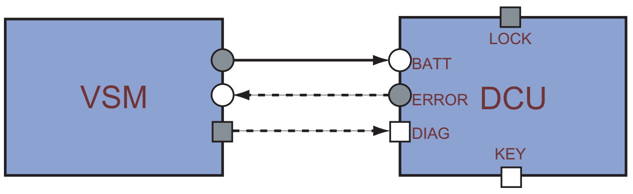

Figure 1: Description of Static Aspects of Components

Thus, in the approach presented here, we focus on the description and analysis of a notion of compatibility in form of communication obligations as an intermediate aspect between the description of completely static aspects and fully-fledged dynamic descriptions. As in embedded systems (volatile) signals are used for communication, possible loss/lack of signals as a source of unintended behavior (e.g., ignored lock/unlock signals in door control, sporadic status events in sleep mode) is an essential aspect in the development of embedded control software. As timing of interaction plays a decisive role in the construction of reliable embedded systems, the detection of possible lack and loss of signal due to out-of-time reading or writing is considered.

To support such a (practically feasible) notion of compatibility, domain-specific abstractions of interaction behavior commonly found in embedded systems are used. To that end, *state signals*, *event signals* and *modes of operation* are introduced (Section 2), allowing to describe basic interaction patterns extending static descriptions. By formalizing these patterns (Section 3), incompatibilities still manageable by analysis tools like temporal model checkers can be achieved (Section 4).

2 Signals and Obligations

When constructing networks of communicating control components, generally *signal-based communication* is used. Basically, a signal can be understood as a function mapping time values to a data value, thus including a value domain (the *type* of the signal) as well as a temporal domain (the value of a signal depending on the time point).

Current state-of-practice approaches include these value domains of signals in their interface descriptions. As illustrated in Figure 1 and Table 1 (columns ‘Direction’, ‘Name’, and ‘Domain’), these interface descriptions consists of a number of input and output *ports* (e.g., BATT, ERROR, DIAG), used to receive and send signals; for each port, furthermore the type of the signal exchanged via this port is defined. When composing components to networks by linking output ports to input ports, compatibility of the types of the linked ports is ensured. Thus, types can be regarded as a simple (static) contract between a sender and a receiver of a signal, stating that the former does not send values that the latter cannot process.

Especially on the implementational level, the temporal domain of an interface of a component (e.g., in terms of its CAN bus interface) is additionally described. In general, the temporal description defines when a new signal value is needed or must be provided at its interface ports. Often, these describe a maximum communication profile and do not consider the fact that produced and consumed signals may vary throughout the operation of a component. These coarse type and static communication aspects of

Direction	Type	Phase	Period	Name	Domain
in	state	0 ms	100 ms	BATT	Voltage 5V - 18V, 0.1 V Integer Value 50 to 180
in	event	0 ms	100 ms	KEY	Door key status LCK, UNL, HLD
⋮	⋮	⋮	⋮	⋮	⋮
out	state	0 ms	500 ms	ERROR	Error Code Locking B_LOW_SEAT B_LOW_KEY ERROR_KEY
out	event	100 ms	100 ms	DOOR	Door lock command OPN, CLS, RST

Table 1: Functional Description of Basic Interfaces

a component interface only allow restricted forms of compatibility analysis concerning the possible lack and loss of signals. Therefore, by introducing classes of signals and modes of operations into these descriptions, finer distinctions are possible.

2.1 States and Events

Besides the type and the value domain, signals can also be regarded as having a functional domain, which determines how information must be exchanged and used. In the field of embedded control software, signals generally can be classified as *state signals* typically used to describe data flows communicated between quasi-continuous functional components, modeled, e.g., using Simulink or ASCET; or *event signals* typically used to describe commands communicated between discrete state-based components, modeled, e.g., using Stateflow or Rhapsody. This distinction influences the communication patterns applied when combining components to form reliable systems. As state signals are especially used to convey computational information, it is generally considered safe to write a signal more often than it is read, overwriting unused state information by its more up-to-date version. Symmetrically, as event signals are especially used to transport commands, it is generally considered safe to read a signal more often than it is written, ensuring the observation of raised events.

This functional dimension is important since the resulting communication obligations influence the contract between a system and its environment established by a schedule:

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

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

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

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

From	BATT	KEY	...	...	ERROR	LOCK	To
STD_BY	-	LCK,UNL	...	...	-	CLS,OPN	NORMAL
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮

Table 2: Description of a Mode Transition

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.

By considering these functional aspects, a more fine-grained analysis of compatibility is possible. To that end, we must explicitly specify for each port whether we expect communication to occur, by assigining *communication obligations* to ports ('Communication forbidden', 'Communication may occur', 'Communication must occur'). Combined with temporal restrictions, these obligations are added to the interface description of a component, e.g., using standard variations. As shown in Table 1 (columns 'Direction', 'Type'), combinations like 'state input', 'event input', 'state output' or 'event output' can be used to declare that communication must at least/at most occur during certain intervals. For compatibility checks, it is sufficient to identify combinations of communication obligations that are erroneous, e.g., communication must take place at an output port, but is forbidden at the input port; or communication must take place at an input port, but is forbidden at the output port.

2.2 Modes and Obligations

As indicated above, communication obligations of an embedded control component generally vary during its operation, depending on the current (internal and external) state of the component. Therefore, a description of the global and thus static interaction aspects of a component, as introduced in Subsection 2.1 does not suffice for a practicable notion of compatibility. However, just as a static description of the communication obligations of a component only allows a very weak characterization of its dynamic behavior, a complete dynamic description mostly results in characterizations too complex to be mechanically checked for realistic components. Therefore, a coarse partitioning is needed that includes the description the interactions of a component, but is less complex than a full behavioral model.

This partitioning is achieved by means of *modes of operation*. Typically, modes are coarse partitions, characterizing different phases (e.g., start-up, operation, shut-down) or control schemes (e.g., cranking, warm-up, running). Each mode is an abstraction from the exact state of a component; it combines all states of a component that result in identical obligations. Thus, while residing in a certain mode, the interactions of a component can be described using, e.g., the notation applied in Table 1.

For the description of the mode-dependent communication obligations of a component, the declaration of the modes of a component (including its initial mode), the communication obligations of every mode, and the transition between these modes must be described. While a mode characterizes the behavior of a component over a certain range of time (basically defined by the timing conditions assigned to its signals), a transition is assumed to take no time. As shown in Table 2, a transition is described by the source mode ('From') the transition is originating in, the collection of values for

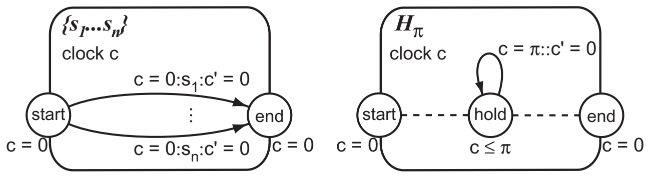

Figure 2: Simple Patterns of Standard Behaviors

the input signals triggering in the transition (e.g., 'BATT', 'KEY'), the collection of values for the output signals effected by the transition (e.g., 'DIAG', 'DOOR'), the target mode ('To') the transition is terminating in. A transition enabled in a source mode terminates the behavior characterized by this mode and continues with the behavior characterized by the target mode. If no transition is triggered, the component remains in its current mode for another cycle.

3 Formalizing Obligations

To mechanically check the compatibility of communication obligations, the introduced concepts like state and event signals and their timing dimensions must be formalized. Here, timed automata are used. Furthermore, as incompatibility of communication obligations can be defined as lack or loss of signals, these formalizations are extended to detect those forms of communication faults.

The interface descriptions introduced define communication obligations; their formalizations describe *contracts* between the component and its environment. To construct these contracts, basic patterns of interaction are combined. *Signal schedules* describe these contracts for a single signal; their parallel composition is used to describe a contract for a *compound interface*. Finally, combining these compound schedules, *mode schedules* are defined for components with mode-dependent contracts.

3.1 Interaction Patterns

Schedules are focused on the description of the interaction obligations of components. Thus, in general they represent abstractions of the actual behavior of those components. For practical usability, it is necessary to offer standard forms to describe these abstractions. Here, we use a modular approach similar to [6] that allows to construct complex descriptions by combining simpler patterns.

To illustrate the principles of this form of modular description, Figure 2 shows some simple behavioral modules. Each module describes a part of the overall behavior of a component. To combine these modules, each module includes (a set of) entry and exit locations. The left-hand module $\{\mathsf{s}_1, \ldots, \mathsf{s}_n\}$ of Figure 2, e.g., describes a partial behavior that, once entered through entry location start is ready to accept a single signal from the set $\{\mathsf{s}_1, \ldots, \mathsf{s}_n\}$ and can then be exited through exit location end.

To formalize the behavior of a module, the concepts of timed automata are used: locations (e.g., start, end, hold), variables (including clocks, e.g., c), and transitions. Here, transitions – connecting locations – are annotated with a pre-condition (characterizing a possible state of the variables prior to the execution of the transition), a

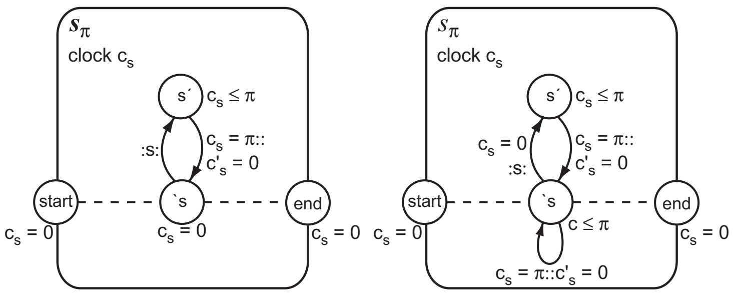

Figure 3: Formalization of Schedules for Obligatory and Optional Signals

synchronization label (synchronizing the interactions of a component and its environment), and a post-condition (characterizing a possible state of the variables after the execution of the transition). Thus, the label “$c = 0 : s_i : c' = 0$” states that by exchanging signal s_i when clock $c = 0$, the transition can be executed, leaving $c = 0$ unchanged. As usual, unprimed variables reference values in the state before the execution of the transition, primed variables reference values after its execution.

Note that entry and exit locations need not be disjoint; the right-hand module $\mathbf{H}_\pi$ describes a partial behavior with overlapping entry and exit location (indicated by the dashed lines connecting them to the internal location hold). To define the behavior – requiring a component to repeatedly hold all interaction for a duration of π until exited – invariants are used, restricting the possible state of variables while in that location. Invariant “$c \leq \pi$”, e.g., enforces a transition at time π.

3.2 Signal Schedules

Embedded software is generally built upon periodic behavior (e.g., speed measurement activated every 500 ms); therefore, in the domain of embedded control software, modular forms of periodic behavior are essential patterns to base more complex descriptions on. For a component with a very basic communication scheme, its communication schedule can be defined independently for all its ports. A standard communication behavior consists of repeatedly performing an interaction; the delay between those equidistant interactions is called *period*.

However, besides this timing aspect, additional functional aspects must be considered when describing those patterns, especially the distinction between event-based and state-based communication paradigms discussed in Subsection 2.1. This functional dimension is important since it does influence the obligations of either systems and environment established by a schedule, as discussed in Subsection 2.1. While event input and state output signals offer guarantees about the signals consumed or produced without imposing requirements about the environment, state input and event output signals require the environment to produce or consume signals in time. Thus, the former can be understood as optional obligations to interact, while the later are obligatory obligations to interact.

Figure 3 shows the formalization of these kinds of signal schedules. The module s_π in the left-hand side describes the obligatory case. The corresponding automaton uses a clock variable c_s to formalize the timing conditions defined by the schedule. Location `s characterizes the state prior to the reception of a signal; location s´ characterizes

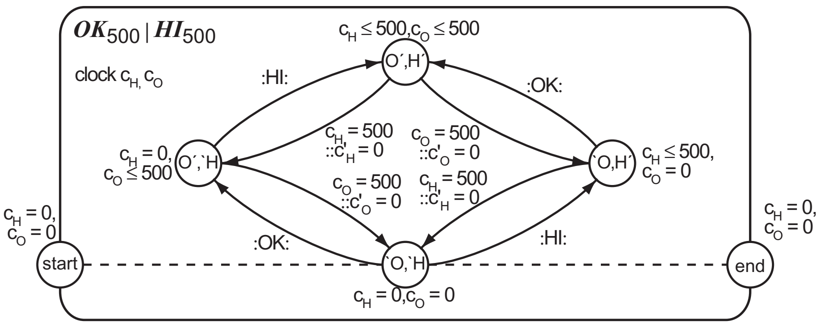

Figure 4: Combining Signal Schedules to Compound Schedules

the state when a signal has been exchanged. Location `s is both an entry and an exit location as well.

The transition from `s to s´ – labeled ": s :" – corresponds to the exchange of a signal at time $0$. The transition from s´ to `s – labeled "$c_s = \pi$: s : $c'_s = 0$" – marks the end of the current period and the beginning of the next. Note that this formalization states that the *exchange must take place at the defined time points*: as `s restricts c_s to 0, the corresponding transition *must* be taken, unless the signal schedule is aborted. As $c_s = 0$ is entry and exit condition, the schedule is started at time 0 and may be aborted at any time $n \times \pi$ for $n \in \mathbb{N}$.

The right-hand side of Figure 3 shows the formalization s_π of an optional obligation for signal exchange. In contrast to module s_π, an optional signal offers an exchange each period, *but does not require the exchange to be imposed on the environment*. Therefore, compared to obligatory schedule its formalization allows the environment to ignore the interaction by means of a weakened invariant $c_s \leq \pi$, while the synchronization transition is strengthened to $c_s = 0$: s :. Furthermore, a feedback transition in location `s with pre-condition $c_{out} = \pi$, resetting the clock variable ($c'_s = 0$) is added.

By using several transitions from `s to s´, each using a different synchronization label as in the basic module $\{s_1, \ldots, s_n\}$ of Figure 2, the exchange of signals with distinct values communicated over a single port is formalized. Thus, by means of obligatory and optional schedules, state input and event output as well as event input and state output signals can be adequately described: a state input signal corresponds to a obligatory schedule, as does a event output signal; symmetrically, a state output signal corresponds to an optional schedule, as does an event input signal.

3.3 Compound Schedules

Signal schedules describe interactions concerning a single (generally multivalued) signal, e.g., occurring at a single port of a component. By combining signal schedules to *compound schedules* the restrictions imposed on the interactions of a components are described, which are communicating via multiple ports.

To formalize the definition of a compound schedule, conjunctive composition of modular descriptions by means of their interface locations is used, similar to [6]. Basically, conjunctive composition is obtained by constructing the product space of the models, using the locations and local variables of either module. The transitions are obtained by applying the original transitions on either part of the product space. Lo-

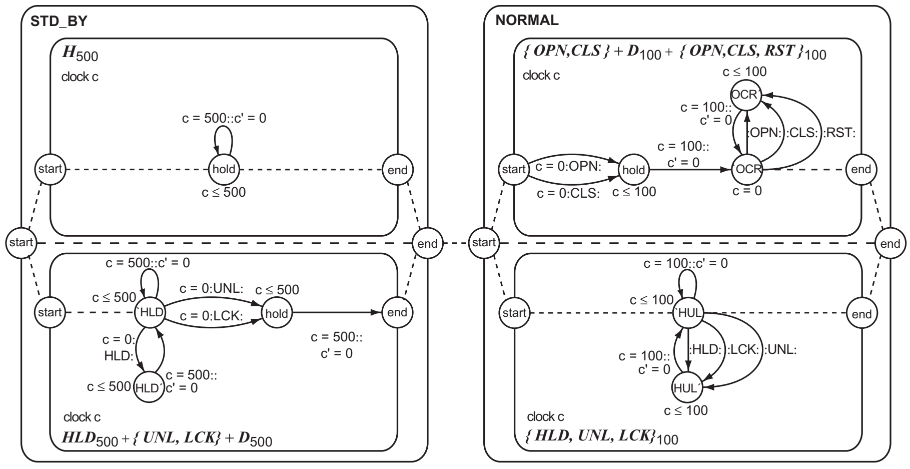

Figure 5: Combining Compound Schedules to Mode Schedules

cation invariants are obtained by the conjunctions of the corresponding invariants of either module; entry and exit locations obtained from locations corresponding to entry and exit locations in both modules.

Figure 4 illustrates this composition for the signal schedules $\mathbf{OK}_{500}$ and $\mathbf{HI}_{500}$, describing the interaction obligations for a component with an event output signal OK (e.g., ECU status information) and a state input signal HI (e.g. battery status information). Intuitively, the activation of the compound schedule through its entry locations corresponds to the joint activation of the signal schedules through their entry location connectors; their joint deactivation corresponds to the deactivation of the component.

Thus, by combining signal schedules into compound schedules the interaction obligations can be described for components with a stereotypic behavior unchanged during its execution.

3.4 Mode Schedules

When controlling a (technical) process, the corresponding control scheme of the control system does not remain unchanged during its execution; often it can rather be broken up into distinct subschemes depending on state of the controlled process. E.g., when controlling fuel injection, the applied control scheme essentially depends on whether the engine is turned off, cranking, warming up, or running normally. Furthermore, often user-controlled modes of operation are used when constructing control schemes. E.g., in engine control, a set-up mode, a maintenance mode, and a normal operation mode are commonly found.

Basically, mode schedules are constructed from compound schedules and basic description modules by sequential composition: As compound schedules are well-suited to describe stereotypic interaction obligations, they are used to characterize the interaction within a single mode; basic modules are used to describe the switch between modes. Figure 5 illustrates this form of combination for the transition of Table 2, switching from mode ‘STD_BY’ to mode ‘NORMAL’. For each mode, its compound schedule is described by the parallel composition of its signal schedules (here, the ‘DOOR’ and the ‘KEY’ schedule). In the originating mode, no ‘DOOR’ event signal is provided, while the event input signal ‘KEY’ is observed with a period of 500. In the terminating mode, both signals are provided and observed with a period of 100. The switch is triggered upon reception of either UNL or LCK; it results in the initializing the output to either OPN or CLS.

While compound schedules are described via conjunctive composition of description modules, mode-based schedules are described via disjunctive composition. To that end, the union space of the modules is constructed, with locations and variables as well as transitions from both modules. To connect modules, additionally, entry and exit locations can be identified to form shared locations. Intuitively, entering a shared exit/entry location corresponds to simultaneously deactivate and activate the corresponding modules.

While conjunctive and disjunctive composition of signal schedules and basic modules allow the construction of complex mode-based schedules, arbitrary behavioral descriptions can be constructed by using very general basic modules.

4 Analyzing Obligations

By explicitly describing the interaction obligations of a component, we can check whether the interactions of two components are compatible, or whether the obligations imposed on a system are ensured by the interactions of its components. To that end, the notion of *compatibility* of interface descriptions is introduced, to detect possible loss or lack of signals when composing components to form systems.

Intuitively, by means of compatibility we want to ensure that no signal is lacking or lost when exchanged between a component and its environment. More formally, if a signal interaction is imposed by a component, it must not be rejected by the environment and vice versa. Obviously, the schedules introduced in Subsection 3 are generally not enabled to accept any signal at any time: for some states and signals, no transitions with a corresponding synchronization labels are enabled; thus the exchange of those signals is blocked.

By using obligations to describe the interactions of components, two forms of compatibility can be ensured in the development process: *compatibility of composition* and *compatibility of abstraction*. The former is used to ensure that the obligations of two components interacting with each other are compatible, thus avoiding any lack or loss of signals in their communication; the latter is used to ensure that the obligations imposed on a system by means of its interface description are met by the obligations of its constituting component interface descriptions.

4.1 Compositional Compatibility

To check for compatibility of components, we compose their corresponding schedules in parallel and check whether the combined schedules may lead to a terminating (i.e., dead-lock) state. Figure 6 illustrates this for the case of a state signal s used both as an input signal with a period of 100 and as an output signal with a period of 200. Composing their schedules – shown in the left-hand side – in parallel with synchronization on s-transitions, leads to the behavior shown in the right-hand side, depicting only the reachable states. During execution, the combined timed automata reaches a deadlock at time point 100 while the receiver is in location`s_1 with $c_1 = 0$ restricting any further delay, the sender is in location s_2´with $c_2 = 100$. Thus, the only transition leaving this combined state – depicted in gray – is not enabled, leading to a deadlock.

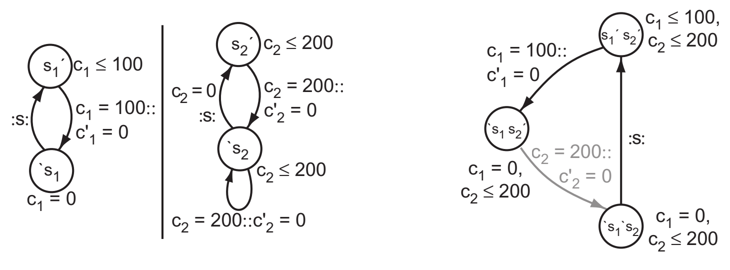
Figure 6: Incompatible Signal Schedules

Thus, a collection of interface descriptions is considered incompatible if their parallel composition may deadlock. While in an asynchronous implementation – as found, e.g., in embedded control networks implementation via CAN and OSEK – deadlock does not occur, it corresponds to the lack or loss of signal. Correspondingly, compositional compatibility can be rephrase as a question of reachability (i.e., reaching a deadlock state), making it accessible to standard model checking procedures.

Based on the kind of deadlock state, furthermore the class of the error can be identified: if the sender is blocked from performing its synchronized action, loss of event occurs in an asynchronous implementation; symmetrically, blocking the receiver corresponds to lack of data. In the example in Figure 6, lack of signal s is detected, assuming that s is a state signal. As, symmetrically the schedules also describe an event output signal with period 100 and an event input signal with period 200, under the assumption that s is an event signal, the loss signal s is detected.

4.2 Abstraction Compatibility

In case of *compatibility of abstraction*, to show that system meets the requirements imposed by its interface description, we ensure that

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

To check compatibility of interface schedules, a technique based on the canonical completion of temporal automata and their parallel composition, e.g., described in [2], can be used, checking the existence of a (pre-)simulation relation between an abstract and a concrete timed automaton. As in the case of the compositional consistency, incompatibility can be rephrased as a reachability issue. Again, the nature of offending state of the composed system can be used to characterize the nature of the incompatibility (lack or loss of signal).

5 Conclusion

This paper introduced a methodological approach to ensure the compatibility of embedded software components.

The idea of using contracts as a basis of interface descriptions has a long tradition (e.g., [7]). By transferring this notion to interaction-based interface descriptions, this concept was applied to the domain of distributed systems (e.g., [1]). To explore possibilities of automatic checks, behavioral interface descriptions, notions of compatibility and refinement have been investigated in approaches like [3], or [4].

However, while these approaches provide general frameworks, (e.g., focusing on blocking communication in general), here, in contrast, we focus on the domain-specific patterns of event- and signal-based communication and the avoidance of lack or loss of signals.

To automatically perform the compatibility analysis steps, UPPAAL [2] is used. To that end, interface descriptions are schematically translated in synchronized timed automata using the modular construction of schedules introduced here. Using parallel composition, compatibility of composition and abstraction is ensured via the validity of the safety properties ensuring the unreachability of locations resulting in loss or lack of signal. The basic techniques of this approach have been applied to case studies in chassis electronics in the automotive domain, demonstrating the basic feasibility of the presented approach.

References

[1] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.

[2] Gerd Behrmann, Alexandre David, and Kim G. Larsen. *A Tutorial on Uppaal*. Department of Computer Science, Aalborg University, Denmark, November 2004.

[3] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *European Software Engineering Conference/ACM SIGSOFT Foundations of Software Engineering*, pages 109–120, 2001.

[4] Luca de Alfaro, Thomas A. Henzinger, and Mariëlle Stoelinga. Timed interfaces. In *EMSOFT Embedded Software*, pages 108–122, 2002.

[5] Harald Heinecke, Klaus-Peter Schnelle, Helmut Fennel, Jürgen Bortolazzi, Lennart Lundh, Jean Leflour, Jean-Luc Maté, Kenji Nishikawa, and Thomas Scharnhorst. AUTomotive Open System ARchitecture - An Industry-Wide Initiative to Manage the Complexity of Emerging Automotive E/E-Architectures. Whitepaper, www.autosar.org, 2004.

[6] Thomas A. Henzinger. Masaccio: A Formal Model for Embedded Components. In *Proceeding of the First International IFIP Conference of Theoretical Computer Science*, pages 549–563. Springer, 2000. LNCS 1872.

[7] B. Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[8] Thurner et al. Das Projekt EAST-EEA – Eine middlewarebasierte Softwarearchitektur für vernetzte Kfz-Steuergeräte. In *VDI-Kongress Elektronik im Kraftfahrzeug*, number 1789 in VDI Berichte, Baden-Baden, 2003.