Determining Compatibility of Embedded Software Components by Communication Obligations

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

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

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

The implementation of automotive systems by steadily growing ECU networks leaves testing increasingly inappropriate as the only means of assuring compatibility of controller interactions. Other techniques, including prescriptive and analytic methods at design level, are needed instead, to ensure an effective and efficient development process. While compatibility checks restricted on architectural descriptions do not capture the necessary dynamic aspects, checks based on full behavioral models do not yet scale for practical applications. Here, the description of communication obligations offers a trade-off between completeness and applicability. To support a methodical approach to compatibility assurance, domain-specific concepts like state/event-signals and modes are introduced, a formalization suitable for mechanic checking is defined, and the relation between interface descriptions and testing is discussed.

1 Introduction

Software engineering for embedded systems is undergoing a paradigm shift from (a more or less monolithic view on) ECUs to networks of interacting software components with well-defined interfaces.

This component view is supported by a number of modeling languages for embedded systems. Typically, these languages are supported by CASE tools such as ASCET, MATLAB & Simulink, Statemate or AutoFocus. Most of these languages describe both the system structure—by some kind of structure diagrams, which show the interconnection of system components—and system behavior, usually with some kind of state transition diagrams.

In addition to these behavioral approaches, there are also modeling approaches that mainly describe system structure, while giving the system developer almost complete freedom of how to implement system components—for instance, in standard programming languages like C, C++ or Java, or by wrapping legacy code or COTS components. Here the system structuring serves to restrict the interface complexity between components. Examples for this approach are AUTOSAR [4] and EAST/EEA [7].

Independent of the implementation question, modeling languages typically have a notion of compatibility of components in order to ensure that component composition is well-defined. In the simplest structural models, compatibility means only that component outputs may not be connected to other outputs or that a component input may only be connected to a single component output. Usually, type information is also considered, meaning for instance that integer-valued outputs may not directly be connected to floating-point-valued inputs.

For behavioral models, more complex compatibility checks can be defined, taking into account the proper pairing and timing of output and input commands in the system components; see [2] for such an approach.

In this paper, we introduce the concept of compatibility for an interface notion based on *communication obligations* (Section 2). Essentially, communication obligations state whether communication between two components is forbidden, whether it is required, or whether it is optional. This approach is then extended to deal with more complex interaction patterns based on com-

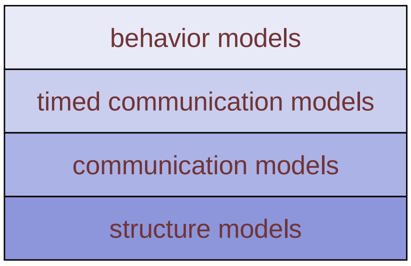

Figure 1: Abstraction levels for interface specifications

munication schedules [6] (Section 3).

As shown in Figure 1, the specification of communication obligations and communication patterns can be regarded as modeling at an abstraction level just between a pure structure model (where compatibility checks are based on type checking) and the detailed communication models derived from complete behavioral models.

In addition to determining whether two components can safely be *connected*, the compatibility notion presented here also allows us to determine if a component can be safely *replaced* by another component. For this application scenario it is easy to see that full behavioral compatibility is not only computationally costly but also methodically inadequate: Only rarely is a replacement component intended to have precisely the same behavior as the original component. More often, it incorporates a number of bug fixes and additional functionalities. At the level of communication models, however, compatibility can be ensured for the harmonious interoperation of the system after the replacement. In Section 4 we show how communication obligations can be analysed.

Of course, compatibility of component interfaces does not guarantee that a system implementation indeed obeys these conditions; Section 5 sketches how existing test processes can be extended to observe whether component implementation indeed satisfy their obligation-based interfaces.

2 Communication Obligations

We consider systems consisting of networks of components. Components communicate via *signals*, i.e. functions mapping time values to a data value. Signals can be used to describe both state-based and event-based communication. In the first case, signal values represent data flow (e.g., between quasi-continuous components modeled by Simulink block diagrams), in the second case, signal values represent control flow in the sense that they carry synchronisation information (e.g., between discrete state-based components models by Stateflow state transition diagrams).

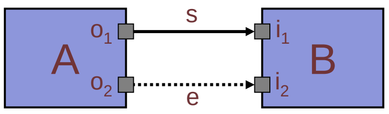

Figure 2: Component connection by state signal s, event signal e

System components have *interfaces*, which consist of a number of input and output *ports*. *Connectors* link output ports of one component with input ports of another. For each connector, there is a signal describing the data or control flow. It is possible for one output port to be connected to multiple inputs ports, but we assume that in the dual case explicit merge components are included to resolve inconsistencies between different signals.

Figure 2 shows a trivial model of two components A and B which are connected by a state signal s and an event signal e. Not all components can be connected—they must be *compatible*.

2.1 Compatibility

What does it mean for components A and B to be compatible? In general, we expect that there is a common condition on the communication between the components that ensures that the sender (component A) only generates signals that can be processed by the receiver (component B).

Usually, this condition is just *type correctness*. For type correctness, the component interface of A assigns to each of the output ports o_1 and o_2 a type (i.e., a set of data values that A may send over the port); similarly, the component interface of B assigns to each of the input ports i_1 and i_2 a set of values that B is able to read from the ports. Type correctness in its simplest form means that the types assigned to o_1 and i_1 and to o_2 and i_2 are identical. Types can be regarded as a simple contract between A and B that A does not send values that B cannot process.

In our model we enhance the notion of compatibility beyond this simple type correctness condition. First, we use the distinction between event- and state-based signals. Second, we introduce communication obligations for output and input ports of components, which may change depending on the current mode (respectively the internal state) of a component. In the following the notion of communication obligations and the definition of compatibility based upon obligations is introduced.

2.2 Static Obligations

Types refer to possible signal values, but not to communication acts themselves. Intuitively, we might expect state communication to take place throughout system execution, and event communication to take place only at some discrete time points chosen by the sender component. This is not reflected in the component interfaces, however.

To formalize the communication aspect, we must explicitly specify for each port whether we expect communication to occur, by assigning *communication obligations* to ports (‘Communication forbidden’, ‘Communication may occur’, ‘Communication must occur’).

For state-based communication, which is used to convey computational information, it is generally considered safe to write a signal more often, than it is read. Symmetrically, event-based communication used to transport commands is considered safe, when each raised event is observed.

This leads to the following communication obligations, which can be read as contracts between a component and its environment:

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

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

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

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

That means for example for an event input port of a component, that if communication may occur the component guarantees that the input is consumed.

For compatibility checks, it is sufficient to identify combinations of communication obligations that are erroneous: communication must take place at an output port, but is forbidden at the input port; communication must take place at an input port, but is forbidden at the output port; communication must take place at an input port, but communication only may take place at the output port.

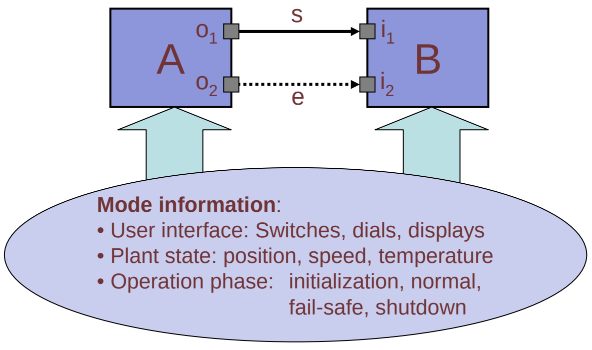
A
B
o_1
o_2
i_1
i_2
s
e
Mode information:
• User interface: Switches, dials, displays
• Plant state: position, speed, temperature
• Operation phase: initialization, normal, fail-safe, shutdown

Figure 3: System modes

In the static form presented above, communication obligations offer little use: If communication is forbidden, one might as well just remove the connection between the relevant ports; the difference between “may” and “must”-communication could simply be modeled by two different connection kinds.

The situation is different, when the communication obligations are dependent on modes or states, as discussed below. Furthermore, obligations can combined with temporal restrictions (like signal periods); this extension is presented in Section 3.

2.3 Mode-dependent Obligations

Embedded systems typically operate in different modes. Modes are rather coarse partitions, distinguishing between different phases (e.g., start-up, operation, shutdown) or control schemes (e.g., cranking, warm-up, running). As theses phases and scheme often affect the overall system, modes of operation are introduced to reflect them. Usually, the communication between two components is mode-dependent.

In contrast with the (internal) component states, there is a system-wide agreement on the current mode. Of course, this agreement must be ensured somehow, either by common observation of externally visible inputs, or by a synchronization protocol. The details of how agreement is ensured are irrelevant for the specification of application-dependent interface behavior. Instead, as indicated in Figure 3, on the application design level modes can in general be understood as externally visible states.

To describe mode-dependent communication obliga-

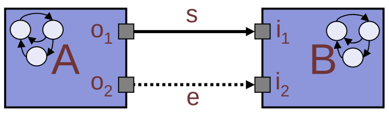

Figure 4: Internal State

tions, the notion of interface and compatibility check is adapted as follows:

- A component interface assigns to each port a mapping from modes to communication obligations.
- For compatibility checking, the compatibility of the communication obligations of two connected ports must be checked for every mode.

The complexity of a compatibility check is affected by the complexity of the modes. As, however, in general modes represent a coarse abstraction of environment and system state, they do not drastically increase the complexity; in practice, by structuring the behavior of the system, they often decrease the overall complexity of a check.

2.4 State-dependent Obligations

In addition to modes, component behavior—and thus, its communication behavior—may also depend on an internal state which is not directly observable. As illustrated in Figure 4, behavior depending on internal state is often described by state machines, but also arises from the control flow in common programming languages.

To describe (internal) state-dependent communication obligations, the notion of interface and compatibility check is adapted as follows:

- A component interface assigns to each port a mapping from the internal state to a communication obligation.
- For compatibility checking, the compatibility of the communication obligations of two connected ports must be checked for each reachable pair of internal states of component A and B.

Internal states, in contrast to modes, are not directly controllable by the component environment. Instead, the component state is dependent on a sequence of environment inputs. Components may also enter different states depending on purely internal choices; to an external observer, such components are nondeterministic. Because of nondeterminism, interface specifications may be weaker than intended or even contradictory (for two internal states which map onto the same mode).

Note that compatibility checks are substantially more expensive for states than for modes, as every reachable state combination must be examined—in practice, this requires state-space exploration tools like model checkers.

3 Timed Obligations

The notion of communication obligations can be extended with a timed dimension, describing for instance signal periods. The formalization of concepts like state and event signals and the interface descriptions that define communication obligations is based on timed automata and more complex than in the untimed case. As incompatibility of communication obligations can be regarded as lack or loss of signals, these formalizations are extended to detect those forms of communication faults.

Signal schedules describe the interface for a single signal; the interface descriptions follow a standard pattern. These aspects are described below. Signal schedules can also be composed; for instance, their parallel composition is used to describe the obligations for a *compound interface*. For details on the composition of schedules, see [6].

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 [5] that allows to construct complex descriptions by combining simpler patterns.

To illustrate the principles of this form of modular description, Figure 5 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 5, e.g., describes a partial behavior that, once entered through entry location start is ready to accept a single signal from

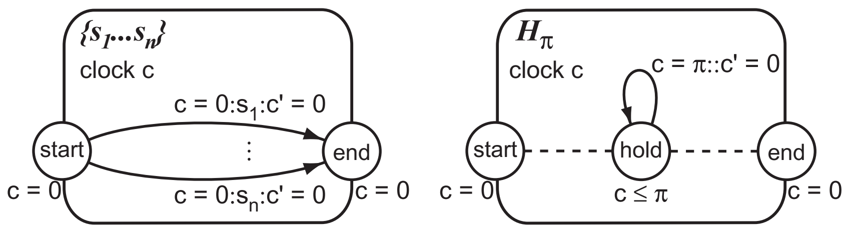

Figure 5: Simple Patterns of Standard Behaviors

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 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 "$\mathsf{c} = 0 : \mathsf{s}_i : \mathsf{c}' = 0$" states that by exchanging signal s_i when clock $\mathsf{c} = 0$, the transition can be executed, leaving $\mathsf{c} = 0$ 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 "$\mathsf{c} \leq \pi$", 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 2.2. This functional dimension is important since it does influence the obligations of either systems and environment established by a schedule, as discussed in Subsection 2.2.

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 6 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 $\grave{\mathsf{s}}$ characterizes the state prior to the reception of a signal; location $\acute{\mathsf{s}}$ characterizes the state when a signal has been exchanged. Location $\grave{\mathsf{s}}$ is both an entry and an exit location as well.

The transition from $\grave{\mathsf{s}}$ to $\acute{\mathsf{s}}$—labeled "$:\ \mathsf{s}\ :$"—corresponds to the exchange of a signal at time 0. The transition from $\acute{\mathsf{s}}$ to $\grave{\mathsf{s}}$—labeled "$\mathsf{c}_\mathsf{s} = \pi :: \mathsf{c}_\mathsf{s}' = 0$"—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 $\grave{\mathsf{s}}$ restricts c_s to 0, the corresponding transition *must* be taken, unless the signal schedule is aborted. As $\mathsf{c}_\mathsf{s} = 0$ 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 6 shows the formaliza-

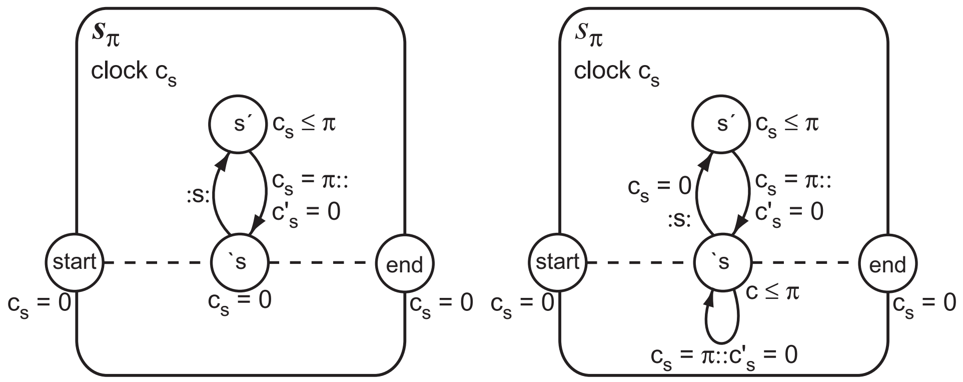

Figure 6: Formalization of Schedules for Obligatory and Optional Signals

tion 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 $c_s = \pi$, 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 $\{s_1, \dots, s_n\}$ of Figure 5, 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.

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.

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 7 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.

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

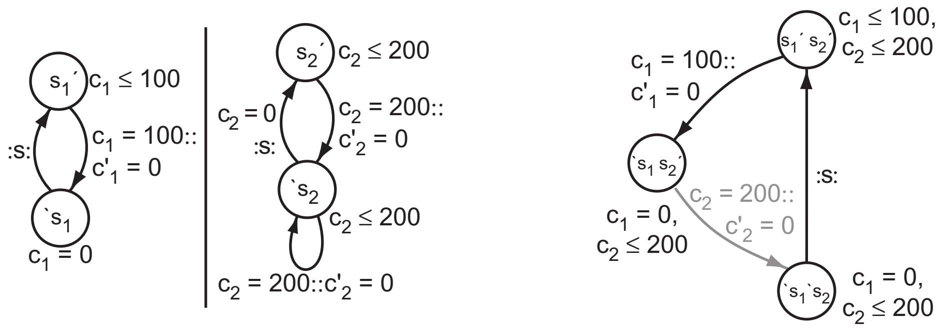

Figure 7: Incompatible Signal Schedules

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 7, 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.

5 Interface Testing

In the preceding sections different abstraction levels for modeling interfaces of components are described. Furthermore a notion of compatibility was introduced which allows to check the consistency of a composition of components for untimed and timed communication models. But beyond checking the compatibility of interface specifications of different components, it must be shown that the implemented system in form of the deployed components indeed satisfies its specification. In this section we illustrate how the conformance of component realizations to their specifications can be established by tests.

Usually, distributed embedded systems use some kind of bus for communication between different ECUs. In automotive systems most commonly the CAN bus is used, which allows state as well as event communication. In such a configuration, it is possible to add an observer to the realized system without changing the communication behavior of a component. So for example the components A and B of Figure 2 may be realized by two separate ECUs as depicted in Figure 8, extended by an additional observer.

Using this extension, the additional observer component monitors the communication between A and B. As the specification is usually more abstract than the technical realization, an abstractor has to be used to interpret the observed technical signals on the abstraction level of the specification. In case of CAN, the signals, specifically coded and packed into messages, have to be unpacked and decoded appropriately.

Using this form of abstraction, the observed communication can be used to evaluate the current mode of the observed components A and B. As explained in Section 2 the mode of an component can be deduced from the observed environment state signals. The current mode of a component determines the actual communication obligations. As the communication between A and B is also observed, the evaluator can establish if the communication is conform to the specification; e.g., if a state signal is not observed while A is in a mode where it must provide that signal, a violation of the specifica-

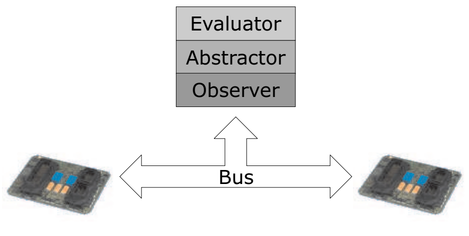

Figure 8: Observer for realized systems

tion by the implementation is detected.

The sketched technique for testing the conformance of a specification and its realization can be easily added into existing testing environments. As the observer doesn't influence the system itself, it is reasonable to add this observer component while the standard tests of a system take place.

One problem with such "piggy backed" tests is that it is not certain that sufficient test coverage has been reached. To determine coverage, the evaluator can log the modes which were reached while executing a given test case. Then, mode coverage can be computed and thus the tester is informed, which modes are not reached within a test case. Based on the modes, an abstract test scenario for reaching a certain mode can also be computed, using, e.g., model checking techniques exploiting the model of Section 3.

As mentioned before, in contrast to modes, states may lead to nondeterminism on the level of communication obligations. The evaluator has no knowledge about the current internal state of a component, but it can approximate the set of states, which may hold at a given time. Based upon this set the evaluator can check if at least the communication obligation of one state is fulfilled or violated.

While we used these testing techniques successfully in practice, they have several limitations. In particular, the communication between two components has to be observable. This is a problem if the relevant communication takes place within a single ECU. Another limitation is that the evaluator may need some time to synchronize its local mode or state model with the system by inferring state information from bus communication at the start of a test; during this time, of course, the evaluator may pass no verdicts. However, the described techniques can be adapted to most situations and lead to highly automated and cost-efficient tests.

6 Conclusion

This paper introduced a methodological approach to ensure the compatibility of embedded software components. Behavioral interface descriptions, notions of compatibility and refinement have been investigated in approaches like [1], or [3], however focusing on blocking communication in general. In contrast, here we focus on the methodical implications of modeling communication obligations, distinguish between event- and signal-based communication, and avoidance of lack or loss of signals.

For practical use, the presented concepts have been formalized and tool support is under development; first case studies in the domain of body electronics have demonstrated the basic feasibility of the presented approach. Obviously, there are many areas for improvement: Depending on the application domain, the concept of modes can be detailed and systematized; the communication schedules can be extended with mechanisms for communication over unreliable media; furthermore concepts for test case generation in addition to coverage measurement should be included.

References

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

[2] Luca de Alfaro and Thomas A. Henzinger. Interface theories for component-based design. In *Proceedings of EMSOFT 2001, LNCS 2211*, 2001.

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

[4] 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.

[5] 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.

[6] Bernhard Schätz. Interface descriptions for embedded systems. In *Proc. 3rd Workshop on Object-oriented Modeling of Embeddded Real-Time Systems (OMER'05)*, Paderborn, 2005.

[7] 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.