Innovations in Systems and Software Engineering
https://doi.org/10.1007/s11334-023-00531-4

S.I. : VECOS 2021

Contract-based specification of mode-dependent timing behavior

Janis Kröger[1] · Björn Koopmann[2] · Ingo Stierand[2] · Martin Fränzle[1]

Received: 18 March 2022 / Accepted: 14 June 2023

Abstract
The design of safety-critical systems calls for rigorous application of specification and verification methods. In this context, a comprehensive consideration of safety aspects, which inevitably include timing properties, requires explicit addressing of operating modes and their transitions in the system model as well as in the respective specifications. As a side effect, this helps to reduce verification complexity. This paper presents an extension of a framework for the specification of timing properties following the contract-based design paradigm. It provides enhancements of the underlying specification language, which enables specifying modes, mode transitions, and mode-dependent behavior. A formal semantics is given in order to enable reasoning about such specifications as well as about contract operations like refinement and composition, thus enabling to make statements about mode composition. The results are discussed using a real-world example.

Keywords Contract-based design · Operating modes · Timing specifications · Mode-dependent specifications · Mode composition

1 Introduction

The design of safety-critical systems heavily relies on the comprehensive elicitation and verification of relevant system properties. Fault tree analysis, for example, is a well-established—and in some application domains mandatory—method to investigate the impact of potential failures on the correctness of system functions and serves as an input for the development of counter-measures in the design. The same applies to failure mode and effects analysis, which is devoted to reveal potential propagation paths of failures in the system. The development and verification of systems that are hardened against failures calls for system models that incorporate operating modes. This enables, for example, detailing the behavior of redundancy mechanisms to check whether a takeover of safety-critical functions of components in failure modes by their backups is performed as expected. Modeling operating modes is also helpful for managing design complexity. For example, modern adaptive cruise control (ACC) systems in road vehicles are able to function in two modes. In *Cruise* mode, the vehicle's velocity is kept to a value set by the driver. In *Follow* mode, the ACC is controlling the velocity such that a safe distance to a previously detected vehicle in front is maintained. The active mode is selected according to the current traffic situation. The authors of Damm et al. [10] developed an approach in which operating modes are realized by individual components that interact via a dedicated protocol to transfer activity tokens. The approach allows to split the verification of system properties into the verification of the individual components and the correctness of the mode transition protocol, thus reducing the overall verification complexity. The reason for this is that modes provide an automaton-like view that significantly constrains the set of state pairs that can actually follow each other, thus focusing and thereby easing proofs of sequential consistency.

Correctness of mode transitions generally concerns two entangled aspects. First, they must occur consistently as specified and do not result in unexpected behavior. In case of the

J. Kröger, B. Koopmann, I. Stierand contributed equally to this work.

✉ Janis Kröger
janis.kroeger@uni-oldenburg.de

✉ Björn Koopmann
bjoern.koopmann@dlr.de

Ingo Stierand
ingo.stierand@dlr.de

Martin Fränzle
martin.fraenzle@uni-oldenburg.de

[1] Department of Computing Science, Carl von Ossietzky Universität Oldenburg, Oldenburg, Germany

[2] Institute of Systems Engineering for Future Mobility, German Aerospace Center, Oldenburg, Germany

Published online: 07 September 2023

ACC example, this means that at any point in time exactly one mode is active. The second aspect is timing, which is closely related to the first, since mode transitions typically do not occur simultaneously among the involved components but must be propagated in the system. Other timing aspects may also play an important role. Safety mechanisms, for example, are specified with respect to fault-tolerant time intervals, which define the maximum period of time allowed from the occurrence of a failure until the measures take effect.

Reasoning about these aspects calls for suitable specification means. Previous work [5, 6] established a framework for the specification and verification of timing requirements, which employs the contract-based design paradigm [4]. According to this framework, systems are modeled in terms of components. Timing requirements of the individual components are expressed in terms of Assume/Guarantee (A/G) contracts. In addition, a set of specification patterns has been defined that allows engineers to express many relevant timing aspects. However, it was already stated there that the incorporation of operating modes would be desirable. This paper aims at closing this gap. Thus, the contributions are as follows:

1. Integration of operating modes into pattern-based timing contracts, including specification of modes, transitions, and mode-dependent timing behavior.
2. Investigation of mode decomposition and resulting proof obligations for common contract operations such as composition and refinement.
3. Demonstration of the practical applicability based on a realistic example.

This article is an extended version of the conference paper [19] at the *15th International Conference on Verification and Evaluation of Computer and Communication Systems* (VECoS'21). It mainly includes refined pre- and post-phase definitions for mode transitions, mode combinatorics for contract composition, and generally improved explanations and application examples.

The paper is structured as follows. Section 2 summarizes previous work and places our contribution in its scientific context. Section 3 introduces the basic concepts that serve as building blocks to describe system models and to reason formally about operating modes. In Sect. 4, we investigate general properties and identify relevant timing phenomena using a simple example system that incorporates mode-dependent behavior. Based on this, we extend two commonly used timing specification patterns in Sect. 5. In Sect. 6, the developed concepts are applied to a realistic example system to examine their practical suitability. Furthermore, we discuss mode decomposition and relevant proof obligations when applying mode-dependent timing specifications in Sect. 7. Section 8 concludes the paper and gives an outlook on future activities.

2 Related work

Existing research that is particularly relevant in the context of our work can be split into two areas. The first area comprises *classical approaches* to model internal states and modes for system design. Related work from this area is presented in Sect. 2.1. The second area focuses on approaches in the field of *contract-based design*, which are discussed in Sect. 2.2. Previous work includes concepts that facilitate the *reuse of existing components* in new environments and application contexts (see Sect. 2.2.1) and address the use of operating modes as a tool to achieve *coordinated behavioral changes* (see Sect. 2.2.2).

2.1 Classical modeling approaches

In his seminal work [14], Harel introduces *statecharts* that extend classical state machines by concurrency and hierarchy to address the problem of state and arc explosion. It introduces a notion of event-based communication by adding constructs to generate and receive events in different subcharts. Over the years, numerous variants and dialects have been developed [31]. Statecharts rely on a powerful semantics to consider hierarchical states. In terms of our approach, this would be equivalent to establishing a *gray box* perspective on component specifications, which is contrary to the principles of contract-based design. In the following, we will therefore rely on an explicit mapping between higher-level and lower-level modes to reason about their relations.

Maraninchi and Remond propose *mode-automata* [22] as a formal and executable formalism to describe mode structures of reactive systems. It enables the modeling of operating modes and mode transitions by means of automata, whose states are assigned to dataflow programs. The authors also introduce operations for parallel and hierarchic composition, the latter adding support for hierarchical modes. In contrast with our approach, mode-automata do not provide descriptive means to express temporal properties.

Finally, modes are also included in languages specialized for the design of safety-critical systems like *Giotto* [15], which adopts a task semantics for modes. In this context, a mode is defined as a fixed set of tasks that are called according to a certain pattern. Giotto also supports mode switches that enable or disable the invocation of tasks. However, since it is dedicated to technical system design, its focus is clearly on implementation and it is not intended for formulating timing specifications.

All three formalisms have in common that there is no direct relation to contract-based design and thus no easy way

to integrate them into the existing methodology [5, 6]. However, they define a set of basic concepts that are essential for handling operating modes in the context of our work.

2.2 Contract-based design

Contract-based design facilitates component reuse by allocating responsibilities in specifications. A *contract* specifies, on the one hand, all environments in which a component is supposed to operate and, on the other hand, the expected behavior of the component if it is used in such an environment. The present work relies on the particular contract theory of A/G contracts developed in [4], in which the *assumption A* specifies the behavior of the environment, and the *guarantee G* specifies the behavior of the component. The explication of responsibilities between the environment and a component is a major advantage with respect to "flat" specifications. In addition, the use of natural language or semi-formal requirements, for example, the specification patterns proposed in [5], helps engineers to capture the meaning of specifications more intuitively than by using formal languages like *linear temporal logic* [26] or *metric temporal logic* [18] formulas.

2.2.1 Reuse of existing components

The idea of A/G contracts is further developed by the concept of *fine-grained contracts* [29]. Here, weak assumptions are employed for specifying alternative contexts in which components can be used. Corresponding guarantees allow to specify the behavior of the components individually for each of the assumed environments. The work is closely related to our approach, as both refine classical A/G contracts. Nevertheless, whereas fine-grained contracts allow to specify intended behavior for different environments, the concepts presented in this article can be used to describe dynamic behavior within an unaltered context.

Reussner et al. address specifications for components that require interface changes in order to be deployed in certain contexts by *parametric contracts* [27]. The concept can be used to explore different dimensions of compositionality when adapting components to new environments. Since parametric contracts are capable of modeling the dependencies between provided and required functionalities, they support predictions about the quality of service, analysis of architectural design, and automated protocol adaptation. In [28], specialized graph grammars are used to reason about these aspects. Firus et al. apply parametric contracts for measuring the performance of software components [13]. The approach determines discrete response time distributions while considering statistical distributions of response times of environmental services. With regard to our work, parametric contracts do not provide support for handling operating modes. Nevertheless, they allow the adaptation of component behavior and interfaces to changing contexts, thus ensuring interoperability in composite systems.

Kugele et al. introduce a component model enriched with interface assertions and *mode-based contracts* [20], which are formulated by means of predicate logic formulas. The approach provides support for determining composed interface assertions for given architecture configurations. It is complemented by algorithms for checking underspecification, overspecification, and specification compliance. Compared to our approach, mode-based contracts do not focus on timing aspects and lack support for hierarchical component models.

2.2.2 Coordination of behavioral changes

Contract-based design has also been employed to model dynamic behavioral changes and coordination mechanisms. In [10], Damm et al. exploit contract specifications to reason about the stability and safety of systems that consist of components, which provide the desired properties only in certain contexts. Control between the individual components is actively passed in terms of a token protocol. The method enables compositional reasoning to reduce the verification complexity.

Champion et al. have designed a mode-aware specification language called *CoCoSpec* [7], which extends the A/G paradigm and is designed to specify synchronous reactive systems. It provides benefits like rigorous feedback for fault localization, a scalable and adequate compositional analysis as well as defensive semantic analysis to identify oversights. The approach is developed as an extension of the synchronous programming language Lustre and provides support for mode-aware verification by using the Kind 2 model checker [8]. In contrast with our approach, CoCoSpec is an implementation-oriented language that is particularly suited for use in later design phases. It is therefore not specifically tailored to early requirements engineering. In addition, the focus is on functional aspects, without providing specification means for timing behavior. Similar to the present article, the relations between hierarchical modes are established by an explicit mapping.

The present work goes along the same line as those mentioned in Sect. 2.2. It builds on top of our previous work on contract specifications [5, 6] that are devoted to the timing aspect. The envisaged extensions will allow the specification of component modes and, in particular (the timing of), mode transitions and mode-dependent behavior, thus enabling a consistent reasoning about mode composition and the effects of individual modes on the behavior of the overall system.

3 Basic concepts

In the context of this work, we rely on an extended version of the system model defined in [5], which was later refined in [6]. According to this model, a system S consists of a set $\mathcal{C}$ of hierarchically nested *components*. Each component $c \in \mathcal{C}$ is equipped with three disjoint sets of *input ports* P_i, *output ports* P_o, and *variable ports* P_v, which together represent the observables of the encapsulated component behavior. In order to provide a meaningful functionality, each component must have at least one port belonging to one of these classes, i.e., $P_c := P_i \cup P_o \cup P_v \neq \emptyset$. The set of *system ports* $\mathcal{P} = \mathcal{P}_i \cup \mathcal{P}_o \cup \mathcal{P}_v$ is defined as the union of all ports of all components that are part of the system.

The interaction of two or more components $c_1, \ldots, c_n \in \mathcal{C}$ is represented by the definition of so-called *signals* $s : \mathcal{P} \to \mathcal{P}^n$. A signal usually connects an output port $p_o \in \mathcal{P}_o$ with a set of input ports $P_i \subseteq \mathcal{P}_i$. For connections between two hierarchy levels, however, signals can also be defined between two or more input ports $s_{in} : p_{i_1} \mapsto (p_{i_2}, \ldots)$ as well as between two or more output ports $s_{out} : p_{o_1} \mapsto (p_{o_2}, \ldots)$ with $p_{i_1}, p_{i_2} \in \mathcal{P}_i$ and $p_{o_1}, p_{o_2} \in \mathcal{P}_o$.

Figure 1 depicts a simple example system. The *System* is decomposed into two main components *Function* and *Observer*, each of which has multiple input and output ports as well as a variable port called *Mode*, whose characteristics are detailed in Sects. 3.1 and 3.2. An interaction component *CC* serves as a converter channel, which forwards only selected events from the variable port *Function.Mode* to the *Observer.Status* input port. The ports of the lower-level components are interconnected with three signals that pass occurring events and value changes along the connections. In addition, the lower-level input and output ports *Input* and *Verdict* are linked to the respective ports of the top-level component. A dotted arrow connects the *Observer.Mode* and *System.Mode* ports, indicating a dependency between the two ports. The exact semantics of this relation is explained later in Sect. 4.

In order to characterize desired system behavior, each component $c \in \mathcal{C}$ can be annotated with a set of *specifications* Φ_c. Naturally, each element $\varphi \in \Phi_c$ must refer only to the component ports P_c to make statements about behavior. Formally, the behavior observable at the components' interfaces is defined in terms of infinite timed traces. In order to specify Φ_c, we rely on declarative specifications in terms of A/G contracts within a compositional reasoning approach, which is detailed in Sect. 3.3.

3.1 Event ports

The sets of input and output ports are collectively referred to as *event ports* $\mathcal{P}_{i/o} = \mathcal{P}_i \cup \mathcal{P}_o$. The behavior observable at port $p \in \mathcal{P}_{i/o}$ is restricted to its value domain Σ_p specified by the port type. We assume a special value $\perp$ to be a member of every value domain, which represents the absence of a value. A notion of *dense time* is used to characterize the occurrence of events. Therefore, we define $\mathbb{T} = \mathbb{R}_{\geq 0}$ to be the time domain. Each event port has non-absent values for all $t \in T \subset \mathbb{T}$. A single event e occurring at port $p \in \mathcal{P}_{i/o}$ is defined as a tuple $e = (\sigma, t)$ that consists of an event value $\sigma \in \Sigma_p$ and a time of occurrence $t \in T$. This allows us to describe the semantics of event ports in terms of timed traces, for which we use definitions from Böde et al. [6].

A *timed trace* over p is defined as an infinite sequence $\omega_p = (\sigma_i, t_i)_{i \in \mathbb{N}}$, in which $\sigma_i \in \Sigma_p$ are elements from the value domain and $(t_i)_{i \in \mathbb{N}}$ forms a monotonic sequence of time instances. We require all timed traces to be non-zeno, i.e., for each $t \in \mathbb{T}$ exists (σ_i, t_i) such that $t_i \geq t$. Moreover, we denote the set of timed traces observable at port p by $\Omega_p = \{\omega = (\sigma_i, t_i)_{i \in \mathbb{N}}\}$. For each port set P, we define a set of timed traces $(\vec{\sigma}_i, t_i)_{i \in \mathbb{N}}$ over P, where $\vec{\sigma}_i = (\sigma_1, \ldots, \sigma_n) \in \Sigma_{p_1} \times \cdots \times \Sigma_{p_n}$ holds. Similar to the consideration of individual ports, we denote the set of traces over port set P by $\Omega_P = \{\omega_P = (\vec{\sigma}_i, t_i)_{i \in \mathbb{N}}\}$. Based on this, the set of *possible timed traces* observable at the component interfaces can be characterized by a timed language $L_P \subseteq \Omega_P$.

3.2 Variable ports

In order to enable specifying modes, we also define *variable ports* $\mathcal{P}_v$. Just like event ports, each variable port has a clearly defined value domain V_p. In contrast with event ports, variable ports have a specified value at each point in time $t \in \mathbb{T}$, which evolves at discrete time points, i.e., no absent values exist. The behavior of a variable port $p \in \mathcal{P}_v$ can thus be characterized as a *value history* $v_p : \mathbb{T} \to V_p$ that maps a well-defined value to each time point $t \in \mathbb{T}$.

To enable the interaction between event and variable ports, we introduce two new event types, namely *set()* and *change()*, that occur on an implicitly defined, *virtual* event port $p_{i/o}(p) \in \mathcal{P}_{i/o}$ that is assigned to each variable port $p \in \mathcal{P}_v$:

1. *set*(p, v) events represent the assignment of a value $v \in V_p$ to p and is called to initiate value changes. Assuming a *set()* event occurring at time $t_i \in \mathbb{T}$, the value of p is updated to $v_p(t_i) = v$.
2. *change*(p, v) events indicate a change in the value of $p \in \mathcal{P}_v$ that results from a *set()* event. They can be used to react to value changes. Whenever a time point $t_i \in \mathbb{T}$ exists such that $v_p(t_i) \neq v_p(t_{i-1}) \wedge v_p(t_i) = v$, a *change()* event occurs at time t_i.

For both events, the explicit specification of $v \in V_p$ is optional. In case of a *set()* event, omitting v means that

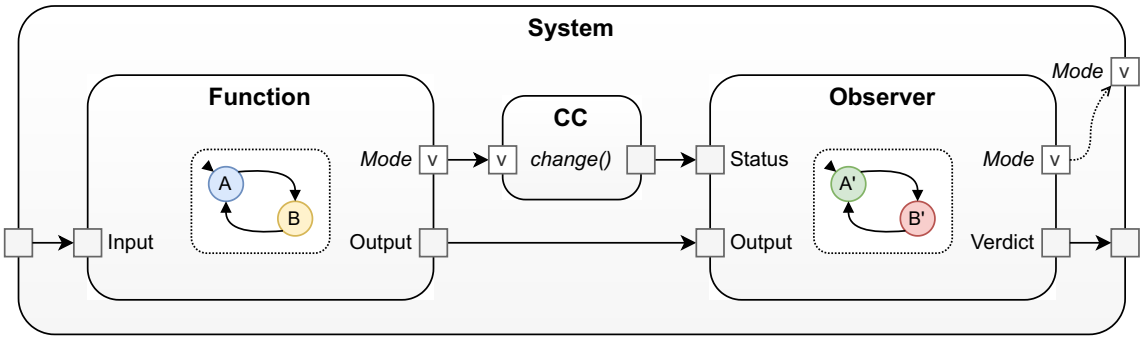

Fig. 1 Component model of the example system

an arbitrary value from V_p is set to p. Omitting v in a *change()* event allows any change of the value of p. Later on, we will need to reason about traces of *change()* events and hence define a *change event trace* $(change()_i, t_i)_{i\in\mathbb{N}}$ for each $p_{i/o}(p)$, which contains all *change*(p) events. Concerning the observable behavior of a component, note that each event port $p_{i/o}(p)$ is a (virtual) member of $P_i \cup P_o$, and thus, *set()* and *change()* events are "visible" in the timed language L_P.

3.3 Specifications

Components can be equipped with *specifications*, which in our case consist of A/G contracts. To keep things simple, we assume exactly one contract $\mathscr{C}_c$ to be assigned to each component $c \in \mathcal{C}$, where $\mathscr{C}_c = (\Phi_c^A, \Phi_c^G)$ such that $\Phi_c = \Phi_c^A \cup \Phi_c^G$. In other words, the specifications in Φ_c are partitioned into those that specify the assumption of the contract and those that specify the guarantee. Each specification defines a set of *valid timed traces* of c, which is denoted by L_φ. As said above, all specifications are defined over the components' interfaces, i.e., $L_\varphi \subseteq \Omega_{P_c}$. Furthermore, we define $L_\Phi = \bigcap_{\varphi\in\Phi} L_\varphi$ and designate the set of timed traces that comply with a given A/G contract as $L_{\mathscr{C}_c} = \Omega_{P_c} \setminus L_{\Phi_c^A} \cup L_{\Phi_c^G}$. In other words, the logical meaning of a contract is an implication, $\Phi_c^A \implies \Phi_c^G$. A component hence is supposed to adhere to its guarantee only if its environment adheres to the component's assumption.

This meaning gives reason to the following definition of *contract satisfaction*: If we denote by L_c the possible behavior of component c, we say, c *satisfies* $\mathscr{C}_c$, written $c \models \mathscr{C}_c$, if $L_c \cap L_{\Phi_c^A} \subseteq L_{\Phi_c^G}$. In other words, c satisfies its contract if it behaves as specified by the guarantee under all possible behaviors of its environment.

There are two main definitions in contract-based design that are relevant in the context of this paper. Any contract theory relies on a *composition* operation that defines how new components are obtained from the composition of other (sub-)components. Along this operation on components, *contract composition* defines how to obtain a contract from the contracts of other lower-level contracts, written $\mathscr{C} = \mathscr{C}_1 \otimes \mathscr{C}_2$. The *refinement* relation defines under which circumstances a component can be replaced by another without violating existing specifications. We say, $\mathscr{C}'$ refines $\mathscr{C}$, written $\mathscr{C}' \preceq \mathscr{C}$, if $L_{\Phi_c^{A'}} \supseteq L_{\Phi_c^A}$ and $L_{\Phi_c^{G'}} \subseteq L_{\Phi_c^G}$. It means that component c' allows for more behavior of its environment and has a more confined behavior than c. Note that $\mathscr{C}_1' \preceq \mathscr{C}_1$ and $\mathscr{C}_2' \preceq \mathscr{C}_2$ implies $\mathscr{C}_1' \otimes \mathscr{C}_2' \preceq \mathscr{C}_1 \otimes \mathscr{C}_2$. We refer the interested reader to Benveniste et al. [4] for more details.

Exploiting contract composition and refinement, we can define what it means for a system (component) S to satisfy a specification. Given that S is specified by a contract $\mathscr{C}_S$, and S is composed by a set of components $\mathcal{C}$, S satisfies its specification, i.e., $S \models \mathscr{C}_S$, if $\otimes_{c\in\mathcal{C}} \mathscr{C}_c \preceq \mathscr{C}_S$.

4 Operating modes

In general, an *operating mode* represents an internal state of a component. In the context of this work, we assume that the mode of a component does not change spontaneously, but is triggered by interactions via its event and variable ports. We define each $c \in \mathcal{C}$ to possess a dedicated variable port $p_m \in P_v$ called *Mode*, which has a *mode history*

$$m_c(t) = v_{p_m}(t) \tag{1}$$

with $t \in \mathbb{T}$. The *active mode* $m_c(t)$ of c results from the initial mode $m_c(0) \in V_{p_m}$ as well as the sequence of *set()* events over $p_{i/o}(p_m)$ that have occurred up to the current time t. The set of modes M_c of a component is given by $M_c = V_{p_m}$, where V_{p_m} has either been explicitly defined or is implicitly derived from the set of values specified in Φ_c. In the following, the change event trace $(change()_i, t_i)_{i\in\mathbb{N}}$ of port $p_{i/o}(p_m)$ is denoted by $(m_i, t_i)_{i\in\mathbb{N}}$. We define a *time-bounded mode-set projection*

$$(m_i, t_i)_{M,I} = \{(m_i, t_i) \mid m_i \in M \wedge t_i \in I\}, \tag{2}$$

where M is a set of modes and I is either a left-closed, right-open time interval $I = [a, b)$ or an open time interval $I = (a, b)$ with $a, b \in \mathbb{T}$.

To shed some light on the interaction of events occurring at the components' interfaces and the evolution of their modes, we again consider the example system presented in Fig. 1. The figure shows that the *Function* component has two modes A and B, which are visible at the variable port *Function.Mode*. The *Observer* component also exhibits two modes, namely A' and B'.

Figure 2 depicts a set of example traces that illustrate the intended system behavior. The *Function* and *Observer* components sequentially process a sequence of events. *Function* receives periodic *Input* events that occur every 6 ms with an offset of 5 ms. Depending on the value of the *Input* events, which is either 0 or 1, the variable port *Function.Mode* is set with a delay of 1 ms. The effects of the *set*(*Function.Mode*, A) and *set*(*Function.Mode*, B) events on the value of the *Mode* port are indicated by dotted arrows. Some of the *set()* events lead to changes of the mode value and thus to the occurrence of *change()* events at the virtual event port $p_{i/o}(p_{m_{fun}})$. They are forwarded to the *Status* port of the *Observer* via the converter channel *CC*, which filters out events other than *change()* events. In addition, the *Function* component adapts the frequency of the *Output* events according to its active mode $m_{fun}(t)$, i.e., by using a period length of 2 ms in mode A and a period length of 5 ms in mode B.

The *Observer* component, on the other hand, receives the *Output* events and evaluates whether the observed behavior is consistent with the (time-delayed) knowledge about the mode of the functional component. To this end, it stores the current mode of the *Function* component in $m_{obs}(t)$ by updating the value of the *Observer.Mode* port according to the received *Status* events. The component creates *Verdict* events to publish the results of the evaluation. A mode-dependent time span of 2 ms in mode A' and 5 ms in mode B' is required to create the verdict. Again, dotted arrows in Fig. 2 show how mode changes of the *Function* component are translated into *Status* events, which lead to mode changes of the *Observer* component. The mode-dependent delay of the *Observer* is visualized by dotted connections between the *Output* and *Verdict* events.

In the course of the paper, we will define mode-dependent timing specifications that are able to cover component behavior as shown in the example. To determine whether a specific behavior is valid with respect to a mode-dependent specification, we must first evaluate which mode the component is in. To this end, we have to define time points at which the current mode of a component is evaluated. Here, we distinguish two cases: simple event occurrences and reactions to trigger events, which correspond to two specification parts that are used to formulate any $\mathscr{C}_c$. For both cases, the *evaluation time points* are fully synchronized with the occurrence of events:

- *Instantaneous* parts specify the occurrence of events according to a specific model, such as the periodic *Output* events generated by the *Function* component. In this case, the mode of component $c \in \mathcal{C}$ is evaluated at each (anticipated) time of occurrence $t_{eval} = t_e^{\uparrow} \in \mathbb{T}$ of the event $e \in \Sigma_p$ at port $p \in P_{i/o}$ of the component. For a given set of modes M_{cond}, we define the corresponding condition as $\exists m \in M_{cond} : m_c(t_{eval}) = m$.
- *Reactive* parts specify the reaction of a component to trigger events $e \in \Sigma_p$, as, for example, the *Verdict* events that occur as a result of the *Output* events. In this case, the condition $\exists m \in M_{cond} : m_c(t_{eval}) = m$ is evaluated at each time $t_{eval} = t_e^{\uparrow} \in \mathbb{T}$ of the event e to be reacted to.

A missing puzzle piece is a mechanism for reasoning about the relations between modes at different levels of the component hierarchy. For the system shown in Fig. 1, an important verification task is checking whether $\mathscr{C}_{fun} \otimes \mathscr{C}_{cc} \otimes \mathscr{C}_{obs} \preceq \mathscr{C}_{sys}$ holds. This check is also called *virtual integration test* (VIT). In order to perform this test, we need to characterize the relation between the top-level modes M_{sys} and the lower-level modes of its subcomponents.

A simple solution is the definition of a *mode mapping function* $\mu_c : M_{sub_1} \times \cdots \times M_{sub_n} \to M_c$ between modes M_{sub_i} and modes M_c for each component c that is decomposed into lower-level components sub_i. In the example above, the mapping function simply maps the observer mode to the system. More precisely, $\mu_{sys} : M_{fun} \times M_{cc} \times M_{obs} \to M_{sys}$ is defined such that $\mu_{sys}(m_{fun}, m_{cc}, m_{obs}) = m_{obs}$. It is important to note that the mapping must be an input of any verification engine in order to perform the above checking task. Applying this approach establishes a tree-like hierarchy between modes, in which well-defined and traceable relations between modes along the component hierarchy exist.

A drawback of this solution is the fact that it introduces an additional specification mechanism. Moreover, it does not explain how such mapping is realized in the system architecture. An alternative solution is to design specific "mode management" components into the system, which realize the mode mapping. An advantage of the approach is the fact that such components could also be used to control and to orchestrate modes of individual subcomponents in addition to only aggregating them. Though the framework presented in this paper supports this approach without any adaptations, the specification patterns developed in the following section are sufficient for specifying only simple use cases. More elaborated scenarios would require specifications beyond purely event-based patterns. More precisely, it would need to allow variable ports as input ports of a component, which is out of scope of this paper. We will come back to this in Sect. 6.

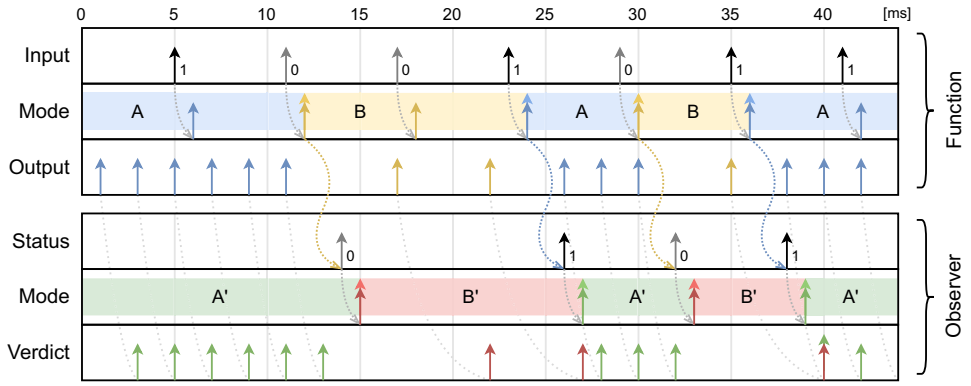

Fig. 2 Intended behavior of the example system

5 Mode-dependent timing specifications

To facilitate the specification of modes and mode-dependent timing behavior, we will extend two existing specification patterns that allow to specify repetitive event occurrences and latencies. The basic idea is to enhance each pattern with an optional set of modes M in which it has to be fulfilled. Since we assume a use in contract-based design, we will present consistent extensions for use in assumptions and guarantees.

On the assumption side, we will allow mode-dependent counterparts of general assumptions. This enables the formulation of more precise and situation-tailored constraints on the component's context. An even more significant reason for introducing mode-dependent assumptions is the advantage in terms of analyzability, since undefined or inconsistent mode combinations are made explicit and can thus be detected more easily. On the guarantee side, we need to define which mode is valid initially, which mode transitions exist, when and in what period these may be taken, and how the mode-dependent timing behavior of the component in focus looks like. In addition, immediate effects of mode changes in terms of adaptions in internal or output behavior will be described with mode-dependent guarantees.

In both cases, we may need to consider the behavior *before* and *after* a mode change takes effect in order to obtain a consistent and well-defined set of specifications. Inconsistencies could arise, for example, when leaving a specific mode and switching to another mode, in which a required input behavior is not (yet) provided by the interacting components. Two conceivable solutions to this problem would be to either ensure the provision of corresponding input events before the mode change is implemented, or to define a limited period of time in which the absence of the required input signal is tolerated after completing the transition.

The approach presented here aims for the second solution. We introduce *pre-* and *post-phases* that enable to permit temporary deviations from mode-dependent behavior within a predefined time interval. Pre-phases, on the one hand, specify a kind of "settling phase" from the time of a mode change $t_b \in \mathbb{T}$ to a fixed time bound $t_b + D_{pre}$ with $D_{pre} \in \mathbb{T}$, in which both the specified behavior of the active mode and (parts of) the behavior of the previous mode may apply. Post-phases, on the other hand, define a "tail phase" that starts with another mode change at time $t_d \in \mathbb{T}$ and ends at $t_d + D_{post}$ with $D_{post} \in \mathbb{T}$ at the latest. Within this period, the specifications of the active mode as well as the specifications of the previous mode pose valid behavior. In the following, we assume default durations $D_{pre} = 0$ and $D_{post} = 0$ for instantaneous as well as $D_{pre} = 0$ and $D_{post} = \infty$ for reactive specifications, respectively.

Based on the introduction of pre- and post-phases, we consider four *specification pattern states* $\mathcal{S} = \{\text{pre, on, post, off}\}$ that are illustrated in Fig. 3. As long as a specification pattern is in the *on* state, the component behavior must comply with this pattern. In the *off* state, the pattern does not need to apply. In *pre* and *post*, the behavior applies as described in the paragraph before. We require that each pattern passes through all states in the given order. Note, however, that each state may also be left after zero time.

For each change event trace $(m_i, t_i)_{i\in\mathbb{N}}$, we inductively define a *specification state trace* $st = (state_i, t_i)_{i\in\mathbb{N}}$ with $state_i \in \mathcal{S}$. The trace st contains all changes between the specification pattern states $\mathcal{S}$, including the exact time t of occurrence. The initial state of each trace is set to

$$(state_0, t_0) = \begin{cases} (\text{on}, 0) & \text{if } m_c(0) \in M \\ (\text{off}, 0) & \text{if } m_c(0) \notin M \end{cases} \tag{3}$$

based on the initial mode $m_c(0) \in M$ of the component in focus. For $(state_i, t_i)$ and $(m_j, t_j)(m_{j+1}, t_{j+1}) \cdots = (m_i, t_i)_{(t_i,\infty)}$, we extend the state trace based on the sequence of specification pattern states from Fig. 3 as follows:

1. If $state_i = \text{off}$:

$$(state_{i+1}, t_{i+1}) = \begin{cases} (\text{off}, t_j) & \text{if } m_j \notin M \\ (\text{pre}, t_j) & \text{if } m_j \in M \end{cases} \tag{4}$$

$$(state_{i+2}, t_{i+2}) = (\text{on}, t_{i+2}) \wedge t_{i+2} \in T_{pre} \text{ if } m_j \in M \tag{5}$$

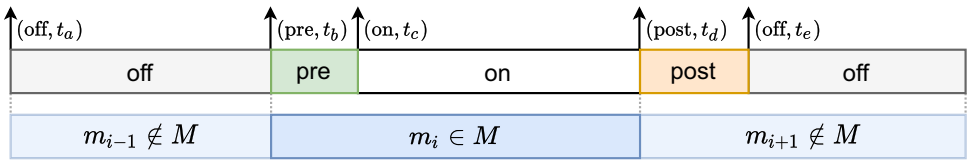

Fig. 3 Specification pattern states and component modes

2. If $state_i = \text{on}$:

$$(state_{i+1}, t_{i+1}) = \begin{cases} (\text{on}, t_j) & \text{if } m_j \in M \\ (\text{post}, t_j) & \text{if } m_j \notin M \end{cases} \tag{6}$$

$$(state_{i+2}, t_{i+2}) = (\text{off}, t_{i+2}) \wedge t_{i+2} \in T_{post} \text{ if } m_j \notin M \tag{7}$$

with

$$T_{pre} = [t_j, \min(t_j + D_{pre}, t_k \mid (m_k, t_k)_{M_c \setminus M, [t_{j+1}, \infty)}))) \tag{8}$$

and

$$T_{post} = [t_j, \min(t_j + D_{post}, t_k \mid (m_k, t_k)_{M, [t_{j+1}, \infty)}))\,. \tag{9}$$

In order to evaluate st at time t, we define a *specification state function* $S_{st} : \mathbb{T} \to \mathcal{S}$ with

$$S_{st}(t) = \begin{cases} s_0 & \text{if } \exists (s_1, t_1) \in \\ & (state_i, t_i)_{i \in \mathbb{N}} : t < t_1 \\ s_i & \text{if } \exists (s_i, t_i)(s_{i+1}, t_{i+1}) \in \\ & (state_i, t_i)_{i \in \mathbb{N}} : t_i \leq t < t_{i+1} \end{cases} \tag{10}$$

that returns the active specification state. Given a state trace st, a subset of states $\mathscr{S} \subseteq \mathcal{S}$ and a timed trace $(\sigma_i, t_i)_{i \in \mathbb{N}}$, the *specification state projection*

$$\text{pr}_{st, \mathscr{S}}((\sigma_i, t_i)_{i \in \mathbb{N}}) = \{(\sigma_i, t_i) \mid S_{st}(t_i) \in \mathscr{S}\} \tag{11}$$

extracts all events from the timed trace that occur while one of the states in $\mathscr{S}$ is active. Note that infinitely many traces st may exist for every change event trace $(m_i, t_i)_{i \in \mathbb{N}}$, because state changes may occur at any point in time within the pre- and post-phases. Finally, we define

$$St : (\Sigma \times \mathbb{T})^\omega \to 2^{(\mathcal{S} \times \mathbb{T})^\omega}\,, \tag{12}$$

which returns all state traces for a given change event trace. Since each specification pattern is assigned to exactly one component that has only a single mode port by definition, explicitly specifying $(m_i, t_i)_{i \in \mathbb{N}}$ is optional.

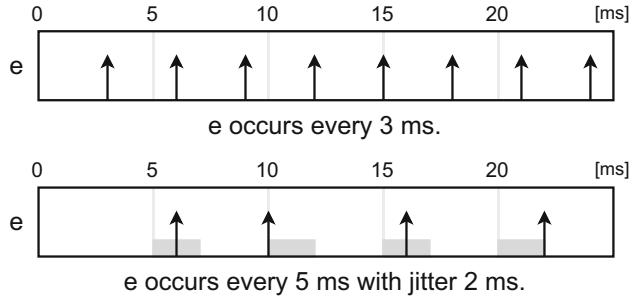

Fig. 4 Repetition pattern examples

5.1 Specification patterns

In the following, we will complete the extension considering two example specification patterns that allow to specify repetitive event occurrences and latencies. The basic idea is to extend each specification pattern of a contract $\mathscr{C}$ with an optional set of modes M in which it has to be fulfilled either by the environment or the behavior of the component itself. Additionally, pre- and post-phase durations D_{pre} and D_{post} are defined for each pattern.

5.1.1 Repetition pattern

The repetition pattern specifies an infinite sequence of recurring events. It expresses that a given event occurs every $P = [P^-, P^+]$ time units, possibly further delayed by up to J time units:

$$\textit{Event}\ \texttt{occurs every}\ P\ \texttt{with jitter}\ J\,.$$

In order to keep things simple, we require $J < P^-$. The pattern is a member of the class of *instantaneous* specifications and can thus be used either in assumptions or guarantees. Its mode-independent semantics is defined by a family of languages $L_{rep}(e, P^-, P^+, J)$, where e represents the *Event*:

$$L_{rep} = \{(e, t_i)_{i \in \mathbb{N}} \mid t_i = u_i + j_i \wedge u_0 \in [0, P^+] \\ \wedge u_{i+1} - u_i \in P \wedge j_i \in [0, J]\} \tag{13}$$

Figure 4 depicts two examples of the repetition pattern. In the upper part, a pattern instance with a period of 3 ms is shown. The resulting e events occur exactly with this periodicity. The lower part contains a repetition pattern with a

period of 5 ms and an additional jitter of 2 ms. In this case, L_{rep} includes all traces in which an e event occurs at a non-deterministic time in each jitter interval.

The syntax extensions for adding support of mode-dependent specifications are as follows:

$$\textit{Event}\ \texttt{occurs every}\ P\ \texttt{with jitter}\ J\ \texttt{in mode}\ M\ [\ \texttt{pre}\ D_{pre}\ ]?\ [\ \texttt{post}\ D_{post}\ ]?\,.$$

The additional components allow the definition of a set of modes M in which the pattern has to be fulfilled. Furthermore, the two optional parts `pre` D_{pre} and `post` D_{post} enable the explicit specification of pre- and post-phase durations.

Based on the results from the previous sections, the semantics of the mode-dependent repetition pattern is defined as

$$\begin{aligned} L^{M}_{rep} = \{&\mathrm{pr}_{st,\{\mathrm{on,post}\}}((e,t_i)_{i\in\mathbb{N}}) \mid \exists st \in St \\ &\wedge\, t_i = u_i + j_i \wedge u_0 \in [t_{init}, t_{init} + P^{+}] \\ &\wedge\, u_{i+1} - u_i \in U \wedge j_i \in [0, J]\} \end{aligned} \tag{14}$$

such that $\exists(\mathrm{on}, t_{init}) \in st : \nexists(\mathrm{on}, t') \in st \wedge t' < t_{init}$ and

$$U = \begin{cases} P & \text{if } \nexists(\mathrm{on}, t_{on}) \in st : u_i < t_{on} \leq u_{i+1} \\ t_{on} - u_i + [0, P^{+}] & \text{if } \exists(\mathrm{on}, t_{on}) \in st : u_i < t_{on} \leq u_{i+1} \end{cases} \tag{15}$$

5.1.2 Reaction pattern

The reaction pattern expresses a "classical" latency between two causally related events. It specifies that an $\textit{Event}_2$ event occurs $I = [I^{-}, I^{+}]$ after each occurrence of a trigger event $\textit{Event}_1$:

$$\texttt{Reaction(}\textit{Event}_1\texttt{,}\ \textit{Event}_2\texttt{) within}\ I\,.$$

The pattern belongs to the class of *reactive* specifications and can thus only be used in guarantees. Its original semantics is defined by $L_{rea}(e, f, I^{-}, I^{+})$:

$$L_{rea} = \{(\sigma_i, t_i)_{i\in\mathbb{N}} \mid \forall(e,t_i)\exists(f,t_j) : t_j - t_i \in I\} \tag{16}$$

Figure 5 shows two examples of the reaction pattern. The upper part depicts an example trace that results from a pattern instance with a latency of 2 ms. In this case, all f events occur exactly 2 ms after the corresponding trigger events e. In the lower part, a reaction pattern with a latency of 2 to 4 ms is shown. L_{rea} contains all traces in which an f event occurs 2 to 4 ms after the e event.

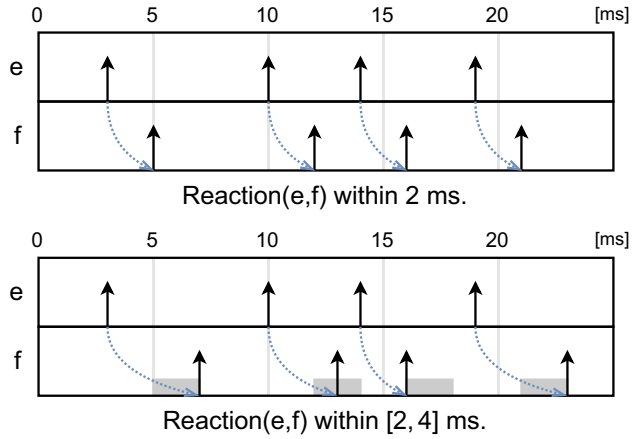

Fig. 5 Reaction pattern examples

In analogy to the extensions of the repetition pattern, the syntax of the reaction pattern is enriched by a set of modes M as well as optional definitions of pre- and post-phase durations:

$$\texttt{Reaction(}\textit{Event}_1\texttt{,}\ \textit{Event}_2\texttt{) within}\ I\ \texttt{in mode}\ M\ [\ \texttt{pre}\ D_{pre}\ ]?\ [\ \texttt{post}\ D_{post}\ ]?\,.$$

The extended pattern specifies a reaction to each trigger event e that occurs while a mode $m \in M$ is active, which only becomes visible within the validity period and the subsequent post-phase:

$$\begin{aligned} L^{M}_{rea} = \{&(\sigma_i, t_i)_{i\in\mathbb{N}} \mid \exists st \in St \wedge \forall(e,t_i) \in \mathrm{pr}_{st,\{\mathrm{on}\}}((e,t_i)_{i\in\mathbb{N}}) : \\ &\exists(f,t_j) \in (f,t_j)_{j\in\mathbb{N}} : t_j - t_i \in I \vee \exists(\mathrm{off}, t_j) \in st : \\ &t_i < t_j \wedge t_j - t_i \in I\} \end{aligned} \tag{17}$$

We are well aware that the extended patterns are only examples of instantaneous and reactive specifications. However, they clearly demonstrate the applicability of the approach and can already be used to cover many relevant application scenarios, some of which are presented in Sect. 6.

5.2 Fine-grained pre-phase and post-phase durations

So far, we were able to specify common pre- and post-phase durations for all modes in which the pattern must be fulfilled. The pre-phase of a pattern applies when the pattern becomes active independently from the previously active mode. Likewise, a specified post-phase applies when the pattern becomes deactivated independently from the following target mode.

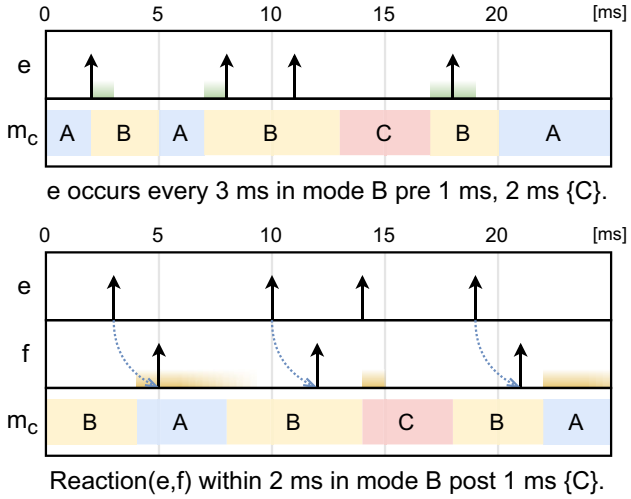

Fig. 6 Examples of fine-grained pre- and post-phases

In some application scenarios, it is desirable to define more fine-grained specifications. We hence further extend our patterns to be able to specify different pre- and post-phase durations depending on the previous (or following) mode. To this end, we extend the pattern syntax by replacing `pre` D_{pre} and `post` D_{post} as follows:

$$\begin{aligned} &\textit{Event}\ \texttt{occurs every}\ P\ \texttt{with jitter}\ J \\ &\quad \texttt{in mode}\ M\ [\ \texttt{pre}\ \textit{PreList}\ ]? \\ &\quad\quad [\ \texttt{post}\ \textit{PostList}\ ]?\,. \end{aligned}$$

$$\begin{aligned} &\texttt{Reaction(}\textit{Event}_1\,,\textit{Event}_2\texttt{)}\ \texttt{within}\ I \\ &\quad \texttt{in mode}\ M\ [\ \texttt{pre}\ \textit{PreList}\ ]? \\ &\quad\quad [\ \texttt{post}\ \textit{PostList}\ ]?\,. \end{aligned}$$

The *PreList* and *PostList* elements are defined as lists of complementary, fine-grained pre- and post-phase durations:

$$\begin{aligned} \textit{PreList} &::= D_{pre_i}\ [\ \{\ M_{pre_i}\ \}\ ]?\ [\ ,\ \textit{PreList}\ ]? \\ \textit{PostList} &::= D_{post_i}\ [\ \{\ M_{post_i}\ \}\ ]?\ [\ ,\ \textit{PostList}\ ]? \end{aligned}$$

With these extensions, we are able to specify optional mode sets M_{pre_i} from which the component changes to a mode in M such that the corresponding pre-phase duration D_{pre_i} applies. Likewise, the optional M_{post_i} specify mode sets to which the component switches when it leaves the mode set M, taking into account the post-phase duration D_{post_i}. Note that any number of pairs $D_{pre}\ \{M_{pre}\}$ and $D_{post}\ \{M_{post}\}$ can be specified.

Figure 6 illustrates two examples of specification patterns with fine-grained pre- and post-phase durations. The upper part depicts an example trace of a repetition pattern with a period of 3 ms and fine-grained pre-phases, which are highlighted in green. The specified duration of 1 ms applies to all pre-phases (at $t = 2$ ms and $t = 7$ ms in the figure), except for mode transitions from C to B. Here, the specified duration is 2 ms (at $t = 17$ ms). In the lower part, an example trace of a reaction pattern with a latency of 2 ms and fine-grained post-phases is shown. All post-phases are highlighted in orange. The pattern specifies that for every event e a corresponding event f occurs. This applies even if the mode in which the pattern is active has already been left, provided that the post-phase duration has not yet expired (at $t = 5$ ms). Recall that the post-phase duration for reaction patterns is ∞ by default. As a result of the post-phase duration of 1 ms specified for transitions from modes B to C, reactions to trigger events may be suppressed (at $t = 14$ ms).

The extensions still allow specifying common pre- and post-phase durations D_{pre} and D_{post} without using M_{pre} and M_{post}, respectively. If there are no other elements specified in the list, the duration applies to all modes in $M_c \setminus M$, and thus $M_{pre} = M_c \backslash M$ and $M_{post} = M_c \backslash M$ are implicitly defined. Otherwise, it applies to all modes in M_c for which no other M_{pre_i} (or M_{post_i}) is specified in the list, and M_{pre} (or M_{post}) is assumed to be set accordingly. Finally, the following conditions must be fulfilled by all $M_{x_{i/j}}$, which represent either pre- or post-phase mode sets $M_{pre_{i/j}}$ or $M_{post_{i/j}}$, respectively:

$$\begin{aligned} &M_{x_i} \subseteq M_c \\ &M \cap M_{x_i} = \emptyset \\ &\forall (i \neq j) : M_{x_i} \cap M_{x_j} = \emptyset \end{aligned} \tag{18}$$

The required extensions are reflected in the semantics by replacing D_{pre} in Eq. (8) by D'_{pre}, i.e.,

$$T_{pre} = [t_j, \min(t_j + D'_{pre}, t_k \mid (m_k, t_k)_{M_c \setminus M, [t_{j+1}, \infty)}))\ (19)$$

with

$$D'_{pre} = \begin{cases} D_{pre_1} & \text{if } m_{j-1} \in M_{pre_1} \\ \quad\vdots & \\ D_{pre_i} & \text{if } m_{j-1} \in M_{pre_i} \\ D_{pre} & \text{else} \end{cases} \tag{20}$$

and D_{post} in Eq. (9) by D'_{post}, i.e.,

$$T_{post} = [t_j, \min(t_j + D'_{post}, t_k \mid (m_k, t_k)_{M, [t_{j+1}, \infty)}))\,. \tag{21}$$

with

$$D'_{post} = \begin{cases} D_{post_1} & \text{if } m_j \in M_{post_1} \\ \vdots & \\ D_{post_i} & \text{if } m_j \in M_{post_i} \\ D_{post} & \text{else} \end{cases}. \tag{22}$$

6 Application example

To illustrate the approach for handling operating modes, we consider an *Adaptive Cruise Control with Collision Avoidance* (*ACCwCA*). The system has been considered and implemented in the context of the Step-Up!CPS project, and is detailed in [1]. The main focus was on the timing behavior of the system, and thus, the overall architecture and time values were previously fixed. An issue that became evident in the previous work was the missing ability to specify operating modes in order to reflect the system behavior more accurately. Hence, the example presented here can be considered as an attempt to fix this issue.

A component model of the system is depicted in Fig. 7. The example focuses on the early phases of system design, where the functional viewpoint is in focus. However, the concepts are not limited to these phases or this particular perspective, but can also be used in later development stages down to the hardware/software design.

The *ACCwCA* is an advanced driver assistance system that possesses a number of modes, namely *Idle* (`I`), *Cruise* (`C`), *Follow* (`F`), and *Evade* (`E`). Initially, the system is in *Idle* mode. It switches into *Cruise* mode when being activated, where it maintains a constant speed set by the driver. If a slower vehicle is ahead of the ego vehicle, the *Follow* mode is activated, and the *ACCwCA* adapts the speed to the leading vehicle and maintains a safe minimum distance. Additionally, the component provides a collision avoidance functionality. If the distance between the leading and the ego vehicle falls below a safety-critical threshold, the *Evade* mode is activated to perform an emergency braking maneuver.

The top-level specification of the *ACCwCA* is shown in Table 1. An upstream sensor processing unit periodically provides the distance *D* to the vehicle ahead and the velocity *LV* of the leading vehicle (line 1). The ego velocity *EV* is received from another external system (line 2). With the occurrence of a *Req* event in mode *Idle*, the system is activated (lines 3–4). After activation, the *ACCwCA* computes regular updates of the control values *Ctrl* as well as *Switch* values that indicate how the values are to be interpreted. A transition from *Idle* to *Evade* is automatically taken in safety-critical situations, i.e., without the need for manual activation (line 5). Depending on the active mode, the system shows different timing behaviors. This is partly to save resources and partly to avoid unnecessary restrictions in the specifications. In *Cruise* mode, the system reacts to its inputs within 245 to 250 ms (line 6). In modes *Follow* and *Evade*, the reaction requires a shorter period of time (lines 7–9).

Taking into account the traffic situation, the *ACCwCA* switches between *Cruise*, *Follow*, and *Evade* until it is deactivated by a driver request (lines 10–11). To ensure continuous control, *Ctrl* and *Switch* are provided every 3 to 7 ms in all operating modes other than *Idle* (line 12). When switching from *Idle* to *Cruise*, *Follow*, or *Evade*, the internal components need some time to adapt their behavior to the requested mode. Since a maximum delay of 30 ms in providing the first *Switch* and *Ctrl* events is acceptable, a pre-phase duration of 30 ms is defined (line 12).

When the component switches to *Idle* mode, it should not produce any further *Ctrl* and *Switch* events. To this end, we specify a fine-grained post-phase duration of 0 ms for transitions to the *Idle* mode (line 6–9). However, a post-phase duration of 25 ms remains for the periodic outputs events *Ctrl* and *Switch* (line 12), which does not comply with the previous requirements. Note that this situation arises for consistency reasons and is explained in more detail in Sect. 6.2.

6.1 Component-level specifications

The *ACCwCA* consists of three sub-components, namely *Adaptive Cruise Control* (*ACC*), *Collision Avoidance* (*CA*), and *Switching Logic* (*SL*). All components have their own internal modes as depicted in Fig. 7. Based on the definitions from Sect. 4, the top-level modes are mapped to combinations of the composed lower-level modes by using the mode mapping function $\mu_{ACCwCA} : M_{acc} \times M_{ca} \times M_{sl} \rightarrow M_{ACCwCA}$. The detailed mapping is shown in Table 2. Each line specifies a combination of modes of the *ACC*, *CA*, and *SL* components as well as the resulting mode of the top-level component.

The combinations show that the mode of the overall system primarily depends on the lower-level modes of the *ACC* and *SL* components. The *CA* influences the mode of the *ACCwCA* only indirectly by inputs that cause mode changes of the *SL*. The interaction of these components is detailed in Sects. 6.1.2 and 6.1.3. As a result of the combinatorics, mode combinations that lead to inconsistencies between different specifications may arise, especially in transition phases. However, since these combinations are transient, a careful definition of suitable pre-and post-phase durations D_{pre} and D_{post} effectively prevents inconsistent behavior. The effects are explained in more detail in the following subsections.

As indicated in Sect. 4, the mode mapping and the corresponding mode transitions as depicted in the top-right part of Fig. 7 could also be modeled as a dedicated *Mode Logic* component. Such a component would be equipped with input ports that are connected with the mode ports of the *ACC* and *SL* components. An output port would present the aggregated

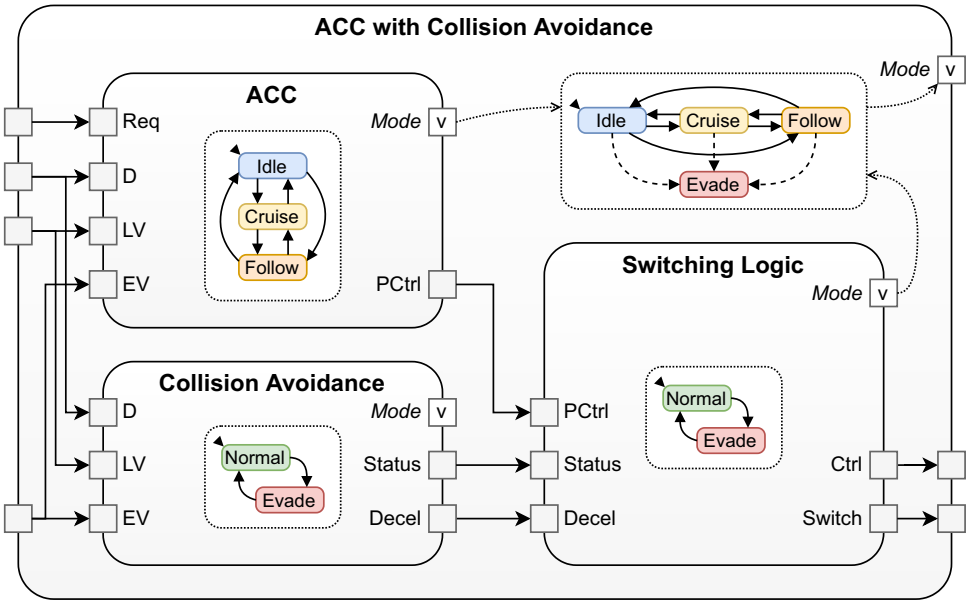

Fig. 7 Component model of the *Adaptive Cruise Control with Collision Avoidance*

Table 1 Top-level specification of the *Adaptive Cruise Control with Collision Avoidance*

A	*{D,LV}* occurs every 40 ms with jitter 5 ms.	1
	EV occurs every 10 ms with jitter 2 ms.	2
	Req occurs every $(0, \infty)$ ms.	3
G	Reaction(*Req*, set(*Mode*)) within (0, 5] ms in mode *I*.	4
	Reaction((*EV,{D,LV}*), set(*Mode,E*)) within (0, 5] ms in mode *I*.	5
	Reaction((*EV,{D,LV}*), *{Ctrl,Switch}*) within [245, 250] ms in mode *C* post 0 ms {*I*}.	6
	Reaction((*EV,{D,LV}*), *Ctrl*) within [195, 200] ms in mode *{F,E}* post 0 ms {*I*}.	7
	Reaction((*EV,{D,LV}*), *Switch*) within [195, 200] ms in mode *F* post 0 ms {*I*}.	8
	Reaction((*EV,{D,LV}*), *Switch.2*) within [195, 200] ms in mode *E* post 0 ms {*I*}.	9
	Reaction((*EV,{D,LV}*), set(*Mode*)) within (0, 5] ms in mode *{C,F,E}*.	10
	Reaction(*Req*, set(*Mode,I*)) within (0, 5] ms in mode *{C,F,E}*.	11
	{Ctrl,Switch} occurs every [3, 7] ms in mode *{C,F,E}* pre 30 ms post 25 ms {*I*}.	12

mode of the *ACCwCA* component. In turn, the mapping of the input modes and the aggregated top-level mode could be specified by means of a specification similar to those of the other components. An excerpt from an example *Mode Logic* specification is included in Table 3. In this case, the guarantee contains four reaction patterns that forward mode changes of the *ACC* component to the output (line 1–3), as well as a change of the *SL* to the *Evade* mode (line 4). However, if the *SL* component switches from *Evade* mode back to *Normal*, the mode previously set by the *ACC* component must be reestablished as the new top-level mode. This would require to store the *ACC* mode into an internal state of the imagined *Mode Logic* component. Conceptually, this could be achieved by introducing a variable in the component that stores the mode, which is supported by the framework through variable ports. Note that corresponding specification patterns are needed, which have not (yet) been defined.

Table 2 Mode combinations and top-level modes

M_{acc}	M_{ca}	M_{sl}	M_{ACCwCA}
Idle	∗	*Normal*	*Idle*
Cruise	∗	*Normal*	*Cruise*
Follow	∗	*Normal*	*Follow*
∗	∗	*Evade*	*Evade*

∗ = "any value"

As a final remark, there is a fundamental difference between the two approaches. While mapping functions "realize" mode changes instantly, components with timing specifications do not allow for instantaneous reactions. This has an impact on the top-level specification, as mode changes become visible to the environment only with a delay.

Table 3 Excerpt from a *mode logic* specification as an alternative for the explicit mode mapping

G	`Reaction(`change(*ACC.Mode*,*I*), set(*Mode*,*I*)) `within` ...	1
	`Reaction(`change(*ACC.Mode*,*C*), set(*Mode*,*C*)) `within` ...	2
	`Reaction(`change(*ACC.Mode*,*F*), set(*Mode*,*F*)) `within` ...	3
	`Reaction(`change(*SL.Mode*,*E*), set(*Mode*,*E*)) `within` ...	4

6.1.1 Adaptive cruise control

The *Adaptive cruise control* (*ACC*) component calculates control values in terms of *PCtrl* events. Its specification is shown in Table 4. The component has *Idle* (`I`), *Cruise* (`C`), and *Follow* (`F`) modes and is activated and deactivated by *Req* events (line 4–5). Situation-dependent changes between *Cruise* and *Follow* are possible at any time (line 6). In *Cruise* mode, the component takes 220 to 225 ms to react to inputs, whereby it has to react faster in *Follow* mode to respond timely to the behavior of the vehicle ahead (lines 7–8). When the *ACC* switches out of *Cruise* mode, we limit the reaction to already received input events to a maximum duration of 200 ms (line 7). If the *ACC* switches to *Idle* mode, no further *PCtrl* values are to be calculated based on input events received (line 7–8). *PCtrl* events are provided every 5 ms in both modes (line 9).

6.1.2 Collision avoidance

The *Collision Avoidance* (*CA*) component, whose specification is given in Table 5, checks whether the execution of an emergency braking maneuver is required. It possesses *Normal* (`N`) and *Evade* (`E`) modes. As long as there is no hazardous situation, the *CA* is in *Normal* mode. If a violation of the critical distance is detected, it switches to *Evade* (line 3) and transmits a *Status* event with value 1 to inform the *SL* component (line 4). The *CA* requires 170 to 175 ms to react to inputs (line 5). After completing of the braking maneuver, it switches back to *Normal* mode and emits a *Status* event with a value of 0 (line 6). To prevent the system from running into inconsistent states, additional *Decel* events are provided for 5 ms after leaving the *Evade* mode (line 7).

6.1.3 Switching logic

The specification of the *Switching Logic* (*SL*) is presented in Table 6. The component has *Normal* (`N`) and *Evade* (`E`) modes. It receives *PCtrl*, *Decel*, and *Status* events as inputs (lines 1–3). Based on these values, it switches between *Normal* and *Evade* modes. In *Normal* mode, it forwards the received *PCtrl* and provides corresponding *Switch* events to enable a correct interpretation of the values (line 4). A mode change to *Evade* is triggered by receiving a *Status* event with value 1 (line 5). In *Evade* mode, it passes the *Decel* values with a *Switch* value of 2 (line 6). The occurrence of a *Status* event with a value of 0 triggers a transition to *Normal* mode (line 7).

6.2 Resulting system behavior

Table 2 shows that the mode of the top-level component *ACCwCA* is depends on the mode of the *SL* component. Based on the description in Sects. 6.1.2 and 6.1.3, it is evident that the individual modes of the components have different criticality for the system behavior. If the safety-critical distance is violated, the highest priority is that an emergency braking maneuver is performed to prevent the system from a collision. Hence, by changing the mode of the *SL* from *Normal* to *Evade* mode, only the *CA* values are output and the *ACC* component values are ignored. This mode change propagates up to the top-level component, as can be seen in Fig. 8. The priority between the output values thus results indirectly from the combinatorics and the mapping between top-level and lower-level modes.

Figure 8 shows example traces that illustrate a possible system behavior resulting from the given specifications. Here, we focus on the evolution of component modes as well as the propagation of mode changes through the overall system. Again, active pre- and post-phases are highlighted in green and orange color. The system is activated at $t = 0$ ms and deactivated at $t = 400$ ms. The top-level mode, which is presented in the bottom line, directly results from the composed lower-level modes as described in Sect. 6.1 (see Table 2). Note that *Cruise* and *Follow* modes are ignored, if the high-critical *Evade* mode is active. In addition, the need for defining suitable pre- and post-phases is demonstrated. In the top-level specification, for example, a pre-phase is required to account for delays in providing initial results (see Table 1, line 12). Moreover, post-phases in $\mathscr{C}_{acc}$ ensure that the control values calculated in *Follow* mode are not overwritten by outdated values from the previous *Cruise* mode (see Table 4, line 7).

In the following, we will examine two significant non-trivial details of the system behavior more closely. Let us first take a look at the parts of Fig. 8 that are highlighted by the overlays ①, ②, and ③. Based on the specification of the *ACC* and *CA* components, which generate periodic output events every 5 ms in their respective modes (see Table 4, line 9 and Table 5, line 7), and the non-varying delay of 25 ms introduced by the *SL* component (see Table 6, lines 4 and 6), we can conclude that the output of the *ACCwCA* also

Table 4 Specification $\mathscr{C}_{acc}$ of the *Adaptive Cruise Control* component

A	*{D,LV}* occurs every 40 ms with jitter 5 ms in mode *{C,F}*.	1
	EV occurs every 10 ms with jitter 2 ms in mode *{C,F}*.	2
	Req occurs every $(0, \infty)$ ms.	3
G	Reaction(*Req*, set(*Mode*,*{C,F}*)) within $(0, 1]$ ms in mode *I*.	4
	Reaction(*Req*, set(*Mode*,*I*)) within $(0, 1]$ ms in mode *{C,F}*.	5
	Reaction(*(EV,{D,LV})*, set(*Mode*)) within $(0, 1]$ ms in mode *{C,F}*.	6
	Reaction(*(EV,{D,LV})*, *PCtrl*) within $[220, 225]$ ms in mode *C* post 200 ms, 0 ms {*I*}.	7
	Reaction(*(EV,{D,LV})*, *PCtrl*) within $[170, 175]$ ms in mode *F* post 0 ms {*I*}.	8
	PCtrl occurs every 5 ms in mode *{C,F}*.	9

Table 5 Specification $\mathscr{C}_{ca}$ of the *Collision Avoidance* component

A	*{D,LV}* occurs every 40 ms with jitter 5 ms.	1
	EV occurs every 10 ms with jitter 2 ms.	2
G	Reaction(*(EV,{D,LV})*, set(*Mode*)) within $(0, 1]$ ms *{E,N}*.	3
	Reaction(change(*Mode*,*E*), *Status.1*) within $(0, 1]$ ms in mode *N*.	4
	Reaction(*(EV,{D,LV})*, *Decel*) within $[170, 175]$ ms in mode *E* post 0 ms.	5
	Reaction(change(*Mode*,*N*), *Status.0*) within $(0, 1]$ ms in mode *E*.	6
	Decel occurs every 5 ms in mode *E* post 5 ms.	7

Table 6 Specification $\mathscr{C}_{sl}$ of the *Switching Logic* component

A	*PCtrl* occurs every $(0, \infty)$ ms.	1
	Status occurs every $(0, \infty)$ ms.	2
	Decel occurs every 5 ms in mode *E* pre 5 ms.	3
G	Reaction(*PCtrl*, *{Ctrl,Switch}*) within 25 ms in mode *N*.	4
	Reaction(*Status.1*, set(*Mode*,*E*)) within $(0, 1]$ ms in mode *N*.	5
	Reaction(*Decel*, *{Ctrl,Switch.2}*) within 25 ms in mode *E*.	6
	Reaction(*Status.0*, set(*Mode*,*N*)) within $(0, 1]$ ms in mode *E*.	7

occurs periodically every 5 ms. Looking at the highlighted parts of the *Ctrl* and *Switch* lines, it can be seen that both outputs do not occur periodically every 5 ms as expected, but later or earlier. This behavior is due to mode changes in the *SL* component. The mode change in the *SL* and the resulting change of the forwarded control values of the *ACC* and the *CA* lead to a discrepancy in the output behavior. Based on this possible temporal variation, the specification of the *ACCwCA* for the occurrence of the output does not result in a 5 ms period as assumed, but in an interval of $[3, 7]$ ms.

The second detail refers to the system when switching to the *Idle* mode. The corresponding part is highlighted by annotation ④ in Fig. 8. As described in Sect. 6.1, the system should not provide any further outputs when being switched to *Idle* mode. Contrary to the intended behavior, some trailing *Ctrl* and *Switch* events occur at the outputs after switching to the *Idle* mode. According to the specification, this happens within a maximum time span of 25 ms. Due to its function as a simple arbiter, the output behavior of the *SL* cannot be restricted by refining the post-phase durations in the specification. To this end, the integration of a mechanism to disable the *SL* component would be required. An alternative solution would be to integrate an additional component that filters the outputs of the *SL* depending on the modes of the internal components.

7 Discussion

The application example illustrates the expressiveness of the extensions and outlines possible application scenarios of mode-dependent specifications. In this context, the two patterns considered in Sect. 5 already enable a detailed formulation of real-time requirements that address operating modes and mode transitions. Due to the diversity of conceivable applications and the complexity of the combinatorics of inter-mode relations, the use of mode-dependent specifications requires a thorough consideration of resulting proof obligations.

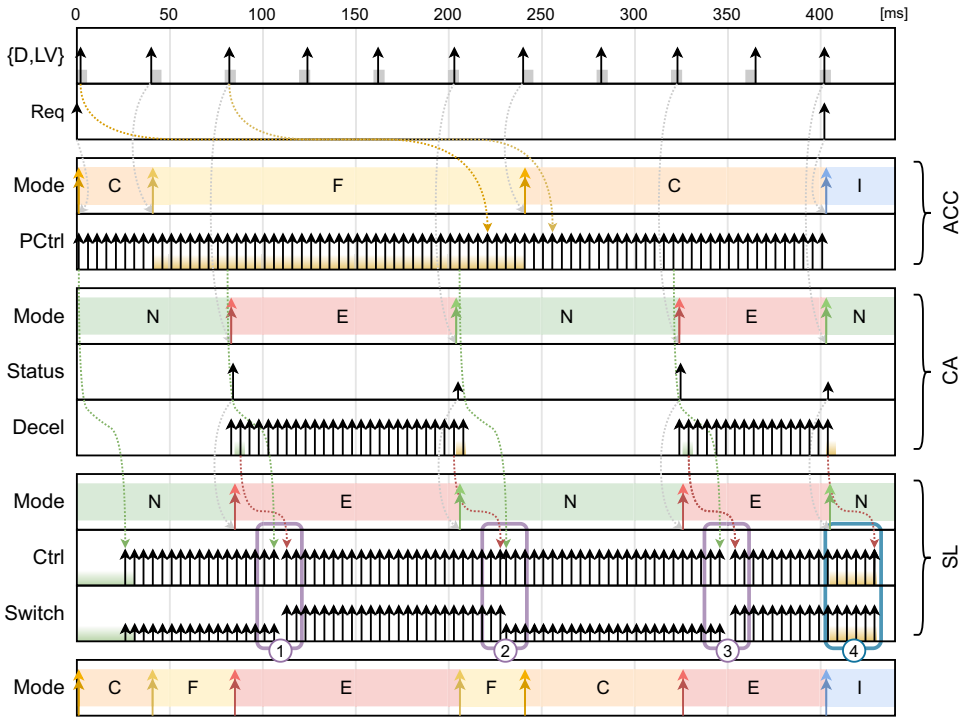

Fig. 8 Example traces of the *Adaptive Cruise Control with Collision Avoidance*

For components that interact with each other, it must be ensured that operating modes takes place in a coordinated manner. Here, it is particularly important to ensure that the pre- and post-phase durations of all transitions defined in different specifications comply with each other.

When considering mode relations across multiple hierarchy levels, the definition of a meaningful mapping is crucial to enable error-free composition. In order to achieve a consistent specification of the behavior of the overall system, all mode mapping functions must be well-defined, i.e., they have to fulfill the following two conditions:

1. The mode mapping μ must be *complete*, i.e., the modes of the higher-level component must result from the cross product of the modes of the respective lower-level components.
2. The mode mapping μ must be *unambiguous*, i.e., it must indeed specify a function from the combination of subcomponent modes to a well-defined output mode.

If both conditions are met, the mode-dependent specification of a higher-level component can be designed in the same way as the corresponding specifications of the lower-level components. Checking the above conditions for mapping function can easily be achieved.

Concerning the consistency between specifications at different hierarchy levels, the usual proof obligations for contract-based design [4] can be applied. The specified behavior in lower-level modes must refine the behavior of their associated higher-level modes. As a consequence, it is required to prove that the behavior in all operating modes is correct with respect to the assumed environments and that valid transitions are taken whenever the environment changes. The underlying contract theory allows to check the validity of refinement relations by means of VITs [12].

Virtual integration testing can also be used to check proper interaction of components at the same level with respect to their specifications. For *compatibility*, we can check whether the composition of multiple components refines the contract $(false, true)$. If so, then there is no environment in which the corresponding components interact with each other in a "correct" way. *Consistency* can be tested by checking refinement of the contract $(true, false)$, meaning that the composition does not yield a valid implementation.

In order to support engineers in these tasks, automated tool support would be desirable. Therefore, mode-aware analyses are an important prerequisite. To close this gap, the extension of an existing tool prototype [11] that performs simulation-based VITs for SysML models annotated with A/G contracts is work in progress.

Viewed from a certain distance, it becomes clear that the concepts are not only suited for handling timing specifications, but can also be used to formalize safety properties. A first step in this direction was the use of mode-dependent specifications for modeling failure modes and mitigation as part of the requirements of the *Model-based Safety Assessment and Traceability* (MobSTr) dataset [30], which is

available on GitHub.[1] The results have recently been published in [2] together with the presentation of prototypical tool support for this kind of specifications.

For most specification formalisms hold that verification is expensive for larger system designs. Contract-based design, on the other hand, enables managing complexity through a divide-and-conquer strategy along component hierarchies. Virtual integration testing ensures that higher-level specifications are satisfied if their composites do so, thus allowing to split verification tasks into smaller pieces. Contracts as a mean of requirement specification can be combined with modeling languages such as MATLAB/Simulink and statecharts [21, 23] in order to check contract satisfaction. Contracts as formal specifications are able to cover different functional, timing, and safety aspects in industrially relevant applications [17, 24, 25]. Besides tool support in the form of generalized verification engines such as UPPAAL [3] and SPIN [16], specialized tools like OCRA [9] exist.

8 Conclusion

The presented approach enables a consistent handling of operating modes, mode transitions, and mode-dependent behavior in contract-based design. In contrast with related work, we performed the extensions on pattern level, which enables the use of timed traces to characterize system behavior and facilitates seamless integration into an existing framework. The concepts allow a comprehensive consideration of timing aspects of mode-based systems and help to reduce verification complexity. In the future, we aim to investigate additional specification patterns and provide tool support for automated analyses.

Acknowledgements This work has been partially funded by the *Federal Ministry of Education and Research* (BMBF) as part of *Step-Up!CPS* (01IS18080B) and *PANORAMA* (01IS18057G) as well as by the *Ministry of Science and Culture* (MWK) *of the State of Lower Saxony* as part of the *Zukunftslabor Mobilität*.
An earlier version of this article was published in the conference proceedings of the *15th International Conference on Verification and Evaluation of Computer and Communication Systems* (VECoS'21) [19]. We would like to thank the organizers for the opportunity to present our results more comprehensively in this article.

Funding Open Access funding enabled and organized by Projekt DEAL.

[1] https://github.com/panorama-research/mobstr-dataset.

References

1. Bebawy Y, Guissouma H, Vander Maelen S, et al (2020) Incremental contract-based verification of software updates for safety-critical cyber-physical systems. In: 2020 international conference on computational science and computational intelligence (CSCI). IEEE. https://doi.org/10.1109/CSCI51800.2020.00318
2. Becker JS, Koopmann B, Stierand I, et al (2023) Providing evidence for correct and timely functioning of software safety mechanisms. In: Groher I, Vogel T (eds) Software engineering 2023 workshops. Gesellschaft für Informatik, pp 66–77. https://doi.org/10.18420/se2023-ws-09
3. Bengtsson J, Larsen K, Larsson F, et al (1996) UPPAAL—a tool suite for automatic verification of real-time systems. In: Alur R, Henzinger TA, Sontag ED (eds) Hybrid systems III, lecture notes in computer science, vol 1066. Springer, Berlin, Heidelberg, pp 232–243. https://doi.org/10.1007/BFb0020949
4. Benveniste A, Caillaud B, Nickovic D et al (2018) Contracts for system design. Found Trends Electron Des Autom 12(2–3):124–400. https://doi.org/10.1561/1000000053
5. Böde E, Büker M, Damm W, et al (2017) Design paradigms for multi-layer time coherency in ADAS and automated driving (MULTIC). In: FAT series, Research Association for Automotive Technology, vol 302. https://www.vda.de/vda/de/aktuelles/publikationen/publication/fat-schriftenreihe-302
6. Böde E, Damm W, Ehmen G, et al (2019) MULTIC-tooling. In: FAT series, Research Association for Automotive Technology, vol 316. https://www.vda.de/vda/de/aktuelles/publikationen/publication/fat-schriftenreihe-316
7. Champion A, Gurfinkel A, Kahsai T, et al (2016a) CoCoSpec: a mode-aware contract language for reactive systems. In: De Nicola R, Kühn E (eds) Software engineering and formal methods, lecture notes in computer science, vol 9763. Springer, Cham, pp 347–366. https://doi.org/10.1007/978-3-319-41591-8_24
8. Champion A, Mebsout A, Sticksel C, et al (2016b) The kind 2 model checker. In: Chaudhuri S, Farzan A (eds) Computer aided verification, lecture notes in computer science, vol 9780. Springer, Cham, pp 510–517. https://doi.org/10.1007/978-3-319-41540-6_29
9. Cimatti A, Tonetta S (2015) Contracts-refinement proof system for component-based embedded systems. Sci Comput Program 97(3):333–348. https://doi.org/10.1016/j.scico.2014.06.011
10. Damm W, Dierks H, Oehlerking J, et al (2010) Towards component based design of hybrid systems: safety and stability. In: Manna Z, Peled DA (eds) Time for verification: essays in memory of Amir Pnueli, lecture notes in computer science, vol 6200. Springer, Berlin, Heidelberg, pp 96–143. https://doi.org/10.1007/978-3-642-13754-9_6
11. Damm W, Ehmen G, Grüttner K, et al (2019) Multi-layer time coherency in the development of ADAS/AD systems: design approach and tooling. In: Proceedings of the workshop on design automation for CPS and IoT. ACM, pp 20–30. https://doi.org/10.1145/3313151.3313167
12. Damm W, Hungar H, Josko B, et al (2011) Using contract-based component specifications for virtual integration testing and architecture design. In: 2011 design, automation and test

in Europe. IEEE, pp 1023–1028. https://doi.org/10.1109/DATE.2011.5763167
13. Firus V, Becker S, Happe J (2005) Parametric performance contracts for QML-specified software components. Electron Notes Theor Comput Sci 141(3):73–90. https://doi.org/10.1016/j.entcs.2005.04.036
14. Harel D (1987) Statecharts: a visual formalism for complex systems. Sci Comput Program 8(3):231–274. https://doi.org/10.1016/0167-6423(87)90035-9
15. Henzinger TA, Horowitz B, Kirsch CM (2001) Giotto: a time-triggered language for embedded programming. In: Henzinger TA, Kirsch CM (eds) Embedded software, lecture notes in computer science, vol 2211. Springer, Berlin, Heidelberg, pp 166–184. https://doi.org/10.1007/3-540-45449-7_12
16. Holzmann GJ (2004) The SPIN model checker: primer and reference manual, vol 1003. Addison-Wesley
17. Kaiser B, Weber R, Oertel M et al (2015) Contract-based design of embedded systems integrating nominal behavior and safety. Complex Syst Informat Model Q 4:66–91. https://doi.org/10.7250/csimq.2015-4.05
18. Koymans R (1990) Specifying real-time properties with metric temporal logic. Real-Time Syst 2(4):255–299. https://doi.org/10.1007/BF01995674
19. Kröger J, Koopmann B, Stierand I, et al (2022) Handling of operating modes in contract-based timing specifications. In: Nouri A, Wu W, Barkaoui K, et al (eds) Verification and evaluation of computer and communication systems, lecture notes in computer science, vol 13187. Springer, Cham, pp 59–74. https://doi.org/10.1007/978-3-030-98850-0_5
20. Kugele S, Marmsoler D, Mata N, et al (2016) Verification of component architectures using mode-based contracts. In: 2016 ACM/IEEE international conference on formal methods and models for system design (MEMOCODE). IEEE, pp 133–142. https://doi.org/10.1109/MEMCOD.2016.7797758
21. Latella D, Majzik I, Massink M (1999) Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. Formal Aspects Comput 11:637–664. https://doi.org/10.1007/s001659970003
22. Maraninchi F, Rémond Y (1998) Mode-automata: about modes and states for reactive systems. In: Hankin C (ed) Programming languages and systems, lecture notes in computer science, vol 1381. Springer, Berlin, Heidelberg, pp 185–199. https://doi.org/10.1007/BFb0053571
23. Nejati S, Gaaloul K, Menghi C, et al (2019) Evaluating model testing and model checking for finding requirements violations in Simulink models. In: Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering. ACM, pp 1015–1025. https://doi.org/10.1145/3338906.3340444
24. Nuzzo P, Xu H, Ozay N et al (2013) A contract-based methodology for aircraft electric power system design. IEEE Access 2:1–25. https://doi.org/10.1109/ACCESS.2013.2295764
25. Nuzzo P, Sangiovanni-Vincentelli AL (2018) Hierarchical system design with vertical contracts, Lecture notes in computer science, vol 10760, Springer, Cham, pp 360–382. https://doi.org/10.1007/978-3-319-95246-8_22
26. Pnueli A (1977) The temporal logic of programs. In: 18th annual symposium on foundations of computer science. IEEE, pp 46–57. https://doi.org/10.1109/SFCS.1977.32
27. Reussner RH, Becker S, Firus V (2004) Component composition with parametric contracts. In: Tagungsband der Net.ObjectDays, pp 155–169. https://sdqweb.ipd.kit.edu/publications/pdfs/reussner2004f.pdf
28. Reussner RH, Happe J, Habel A (2005) Modelling parametric contracts and the state space of composite components by graph grammars. In: Cerioli M (ed) Fundamental approaches to software engineering, Lecture notes in computer science, vol 3442. Springer, Berlin, Heidelberg, pp 80–95. https://doi.org/10.1007/978-3-540-31984-9_7
29. Sljivo I, Gallina B, Carlson J, et al (2013) Strong and weak contract formalism for third-party component reuse. In: 2013 IEEE international symposium on software reliability engineering workshops (ISSREW). IEEE, pp 359–364. https://doi.org/10.1109/ISSREW.2013.6688921
30. Steghöfer JP, Koopmann B, Becker JS, et al (2021) The MobSTr dataset—an exemplar for traceability and model-based safety assessment. In: 2021 IEEE 29th international requirements engineering conference (RE). IEEE, pp 444–445. https://doi.org/10.1109/RE51729.2021.00062
31. von der Beeck M (1994) A comparison of statecharts variants. In: Langmaack H, de Roever WP, Vytopil J (eds) Formal techniques in real-time and fault-tolerant systems, lecture notes in computer science, vol 863. Springer, Berlin, Heidelberg, pp 128–148. https://doi.org/10.1007/3-540-58468-4_163