[KPR+23] H. Kausch, M. Pfeiffer, D. Raco, A. Rath, B. Rumpe, A. Schweiger:
A Theory for Event-Driven Specifications Using Focus and MontiArc on the Example of a Data Link Uplink Feed System.
In: Software Engineering 2023 Workshops, I. Groher, T. Vogel (Eds.),
pp. 169-188, Gesellschaft für Informatik e.V., Bonn, Februar 2023.
www.se-rwth.de/publications/

A Theory for Event-Driven Specifications Using Focus and MontiArc on the Example of a Data Link Uplink Feed System

Hendrik Kausch[1], Mathias Pfeiffer[1], Deni Raco[1], Amelie Rath[1], Bernhard Rumpe[1] and Andreas Schweiger[2]

Abstract:
The development of avionics message communication systems is expensive due to their complexity and the need to get them accepted by the certification authorities. We need to develop high-integrity software, but we also face cost pressure. For managing complex large systems, several time-synchronous modeling languages have been proposed. While these are appropriate for hardware specifications, when it comes to specifying distributed software systems, an event-based specification style is better suited. We present an event-based specification theory based on the framework Focus by giving the signatures and data types for specifications using event automata. For this, we capture message processing order as a further dimension of non-determinism by specifying a general timed merge component. These event automata can represent underspecification of behavior, and a refinement calculus can be applied to these for a stepwise reduction of non-determinism. Furthermore, we present the necessary concepts for enabling a user-friendly specification and simulation of event-based systems by using the architecture description language MontiArc. Finally, we evaluate our approach by performing a top-down architecture design of an avionics case study and demonstrating event-based specifications of requirements in MontiArc. The presented methodology improves the management of complexity, reduces costs, and increases the system quality.

Keywords: event-based components; automata; formal verification; model-driven development; underspecification; distributed systems; avionics

1 Introduction

Safety-criticality of avionics systems increases their complexity. In particular, in message communication systems the complexity needs to be managed, which is not only required during the system's development but also during the product's maintenance and update phases. This is of uttermost importance, since the life cycle of aircraft encompasses usually several decades.

1.1 Related Work

Synchronous dataflow modeling languages like Esterel [Be00] and Lustre [Ca87] (and its dialect SCADE) have been created for the development of such reactive systems, in order to

[1] RWTH Aachen University
[2] Airbus Defence and Space GmbH.

doi:10.18420/se2023-ws-17

handle their complexity. However, due to their time-synchronous paradigm, these are rather suited for the description of hardware systems. A number of further approaches for specifying distributed systems have been developed, such as the Palladio Component Model [BKR09], MechatronicUML [Dz16], AutoFocus [VZ14] or the Ptholemy Project [Le16].

Event-based communication promises to be more scalable and flexible than synchronous communication [KRK13]. For specifying distributed software systems through the approach presented in this paper, we build on our previous works. These differ from the work in this paper in certain aspects. [Kr19, Ka20] used a time-synchronous version of the architecture description language (ADL) MontiArc [RWa, RWb]. MontiArc is built using our framework MontiCore [RHK21, RWc, RWd] for creating domain specific languages (DSLs) based on FOCUS semantics [Br92]. In comparision, MontiArc is extended in this paper to allow event-based processing. Also, the modeling language in the previous works [Ka21b, Ka22, Ka21a] was the alternative SysML. [Bu20] presented an encoding of streams and stream processing functions in the theorem prover Isabelle, but not for (event-based) automata, and covered only the untimed streams. [Ra22] on the other hand focuses on the signatures for timed event-based automata. The event-based approach can if necessary emulate the time-synchronous behavior (which would include using internal component memory).
Model analysis in [Sc20] was performed using the alternative formal method of model-checking. These methods do help to reduce complexity and make the systems safer, however the system requirements of a representative avionics software system treated in this paper (i.e. event-based processing) demanded extensions of our methodology in a number of aspects.

Regarding semantical foundations, CSP [Ho78], CCS [Mi89], *pi*-calculus [Mi99], TLA [AL94], and Petri Nets [Re12] are often used for distributed systems. In comparison, the mathematical formalism of FOCUS we use in this paper as semantical underpinning has a unique property, that its refinement mechanism is fully compositional.

1.2 Structure of the Paper

Until now we have mentioned the context of safety-critical avionics systems. Next, we present the problem statement using a running example. Then, we mention the challenges that arise and list the contributions of this paper regarding how we dealt with the challenges. Afterwards, in section 2, we will present in more details the mathematical foundations of our formalism and the necessary extensions. Furthermore, we will introduce the ADL used as frontend and the necessary adaptions, in order to specify components in an event-based style. In section 3, we show how the methods and languages are used by specifying event-driven components of the case study. Finally, we conclude by summarizing this paper and giving key takeaways.

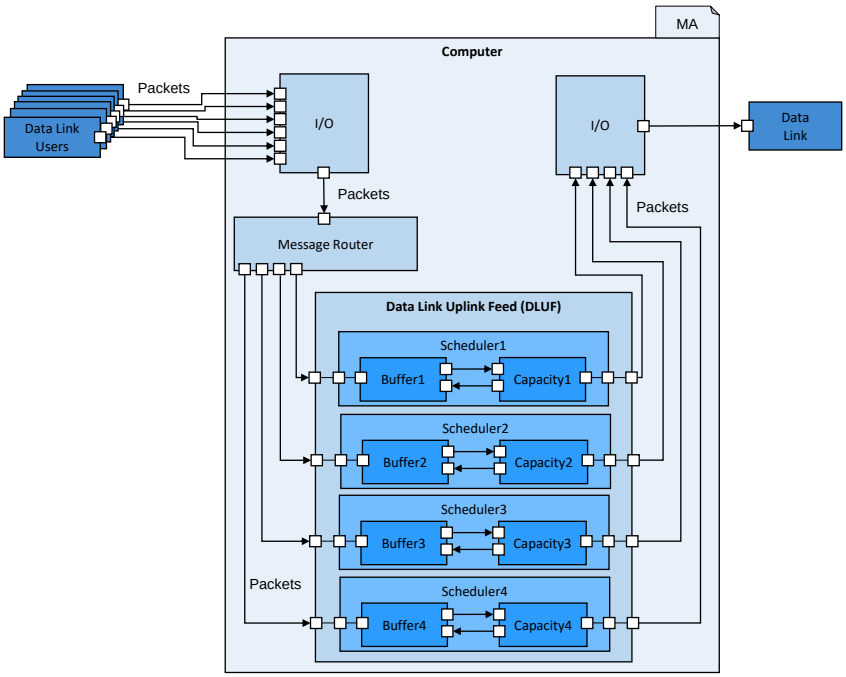

Fig. 1: The design in the graphical representation of the ADL MontiArc (MA) as presented in Sect. 2.6.

1.3 Problem Statement

Often, time-dependent avionics software systems are to be implemented, which perform operations as reactions to incoming events. While for hardware systems a time-synchronous paradigm is well-suited, in distributed software systems (as encountered in projects such as [Bo21, Sp22]) an event-driven specification way is much more commonly used. Event-based systems have the advantage of allowing the specification of behavior where events from different input channels arrive in arbitrary intervals and the component can react to them directly. One case study consisting in a transmission system including its core subsystem Data Link Upload Feed (DLUF) (see Fig. 1) [Ka22] was chosen to be implemented, which needed to fulfill system requirements such as:

- The data link shall transmit packets of users with a data rate (i.e. budget) of 10 MByte/s.
- A reaction and possible output should occur directly as a result of a packet-income event. Note, this behavior would not be possible in a time-synchronous system. Removing this requirement on the other hand, would reduce the performance as packets could not be sent immediately.
- Priorities between 1 and 4 shall be assigned to each user.

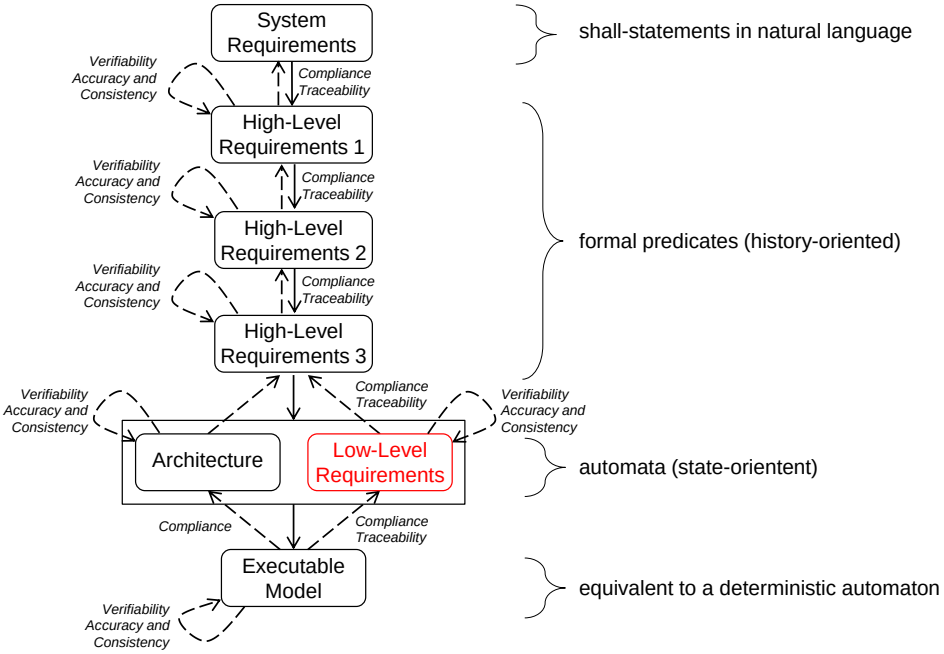

Fig. 2: Systematic design based on RTCA DO-178C (more precisely its supplement DO-333). Implementing the methodology presented here in a theorem-prover like Isabelle [Ka22] allows replacing cost-intensive tests with formal proofs. In this paper we focus on low-level requirements.

- The system shall implement a prioritization processing concept.
- However, this prioritization concept must ensure that packets of all (potentially low-prioritized) users can be transmitted time and time again (non-starvation).
- Packets shall be stored in buffers before being scheduled for transmission.
- Size of packets varies between 1 and 100.000 Bytes.
- The system works in cycles of length 100 ms.
- Per cycle the capacities of the priorities are: 100 KByte, 200 KByte, 300 KByte, 400 KByte. Because of this requirement, an untimed event-based system is not sufficient because no timeout could be modeled to reset the capacities.
- The system shall be performance-optimized.
- The system shall not assume any defined packet format, but treat the packets as byte arrays.
- The system shall not dynamically allocate memory during runtime.

For designing an architecture of the system on different abstraction layers, a scheme based on the RTCA DO-178C/DO-333 was used as reference (see Fig. 2). According to DO-178C,

we go from system requirements to high-level requirements to low-level requirements to an implementation. The chosen systematic design is explained in Sect. 2.1. FOCUS was used as formal foundation. In FOCUS, a component maps input streams into output streams (stream processing function). When developing this case study with FOCUS, a number of technical challenges arise. These are described in the next section together with how to deal with them.

1.4 Proposed solution and contributions

In [Bo21, Sp22] a model-based method is being developed for modeling and analyzing cyber-physical systems. FOCUS is used as a semantics domain and architecture description languages are used as user-friendly frontends for engineers. For an event-driven development of time-dependent avionics software systems, a high-confidence development can be achieved by building on an established sound theory and providing necessary extensions, and by providing an ADL and appropriately extending it for user-friendly modeling. The contributions of this paper address occurring technical challenges as follows:

1. Stateful components occur, so the infrastructure for a state-oriented specification style has to be provided. In Sect. 2.3.2 the data structures of event-based components are introduced, in particular event automata represent a state-oriented specification to model stateful behavior (i.e. a component can react to the same input by a different output depending on its current state).
2. Restricted expressive power of untimed streams: Cannot model timeouts, which are needed, since the system uses cyclic calls and resets counters (needs being able to react to the ending of a time interval). In Sect. 2.3.1 the stream data type is extended to timed streams by a special symbol “tick”, which enables reaction to no message income (when two “ticks” occur subsequently), and thus enabling the modeling of timeouts by counters.
3. Since feedback loops occur, a precise semantics for streams flowing in such loops needs to be established by a delay-concept. In Sect. 2.3.3 strong causal functions are defined over timed streams, which enable a delayed reaction on an input message, thus enabling a well-defined semantics of (streams flowing in) feedback loops.
4. Limitations of time-synchronous state-based specifications using port-automata [Bu19]: An incoming input message on one channel is not (but should be) sufficient to cause a reaction (without waiting for other channels). In Sect. 2.4 the specification method of event-based components is introduced, by presenting the signatures of the merge component and of event automata. Compared to time-synchronous specifications, an incoming input message on one channel is then sufficient to cause a reaction.
5. The order of message processing needs to be treated explicitly as additional dimension of non-determinism (for the same input streams on different input channels, the output produced may differ depending on the order the messages are processed), showing the necessity for the specification of a stream-merge component. In Sect. 2.4.1 the

specification of a merge component is given, representing all possible message processing orders.

6. When giving a specification for a merge component, the old known problem of the "merge anomaly" [BA81] needs to be considered. The merge component in Sect. 2.4.1 is not specified in the intuitive way as relational underspecification (one input stream mapped to multiple output streams), but instead the underspecification is represented through sets of (stream processing) functions (as introduced in section Sect. 2.3.3), each of which describes a possible processing order. Strong causality ensures then a delay. This is then a sufficient solution to circumvent the merge anomaly [BA81,Br93], avoiding scenarios as in Sect. 2.4.3.
7. Also, when specifying a merge component, the fact needs to be considered, that a fair (no channel is completely ignored), non-strict (if one input source hangs, e.g. due to an infinite computation, the component does not just stop functioning), untimed merge specification is not realizable (or using the wording of [Br93] "no prefix monotonic implementation exists"). The specification of the merge component in Sect. 2.4.1 was defined over timed streams, which is sufficient to make the specification realizable [Br93] (meaning the specification is consistent, i.e. the set of stream processing functions is not empty).
8. Stream tuples are not sufficient for modeling a general (with respect to arbitrary interface sizes) merge component (since tuples describe only a fixed number of inputs). In Sect. 2.3.2 the precise definition of the structure of (timed) stream bundles is given (isomorphic to stream tuples), which solves the problem of defining a generalized (with respect to an arbitrary number of channels) merge specification.
9. Multiple hierarchy levels of abstraction occur and stepwise refinement is used to reduce underspecification (meaning multiple behaving differently implementations are allowed). The presented event-based infrastructure needs to properly handle behavioral non-determinism (further dimension of non-determinism apart from the processing order) and refinement. In Sect. 2.5 it is shown, that the refinement calculus of [Ru96] can be carried over to event automata, since the automaton signatures are preserved.
10. A user friendlier domain-specific language for event-based systems is needed to hide mathematical constructs from the modeling engineer. However, this language does need clear semantics [HR04] (NB, that e.g. [vdB94] counted over 30 different semantics of statecharts). In Sect. 2.6 the architecture description language MontiArc (see [Bu19] for a time-synchronous specification style) is adapted to enable event-driven specifications and has clear FOCUS based semantics.
11. Since testing still is dominant in quality assurance, a concept for simulating deterministic executable event-driven automata models needs to be presented. In Sect. 2.6.1 a simulation method for event-based system specifications is presented.

2 Towards a Theory for Event-Based Processing Components

In this section we start by describing in more detail Focus and the top-down design according to RTCA DO-178C/DO-333. Afterwards, we proceed by presenting the data structures and concepts needed for the lower-level abstraction layer, where the event automata will play the main role.

2.1 Focus and Top-Down Design

Based on a mathematical and logical foundation, Focus is a formal framework capable of representing distributed systems at different abstraction levels. It is used for the step-wise development and refinement of interactive systems [BR07] and works with streams representing the history of communication between components. A component is either specified as a black box, capturing only the relation of input and output channels, or as a white box. In the latter case, a component is specified as atomic building block or as several sub-components. Thereby Focus has the unique property, that refinement is compatible with composition [Br92, Ru96], i.e. if one decomposes a system, refines each component independently, and then assembles the system from the components, the composed system is guaranteed to be a refinement of the original one [BR07]. That refinement is fully compositional, allows the following specification method to be used: The subsystem of DLUF (see Fig. 1) Scheduler1 can be decomposed into the components Buffer1 and Capacity 1. The under-specified Buffer1 can be refined individually from a history-oriented specification to an automaton (see Fig. 6). Putting the refined Buffer1' and Capacity1 back together in Scheduler1', the requirements of the original Scheduler1 automatically hold in the new Scheduler1' as well.
This method allows us to break down the proof complexity of mathematical software verification with Focus into the levels of RTCA DO-178C:

1. The level of System Requirements (SRs)
2. The level of High-level requirements (HLRs)
3. The level of Low-level requirements (LLRs)
4. The level of implementation

For our methodology, we use the design in Fig. 2 which is based on RTCA DO-178C/DO-333. In our case, the system requirements are given as shall-statements in natural language. HLRs are specified as formal predicates. We write them as history-oriented specifications in Focus. The specifications are (sets of) functions that state properties on the complete history (infinite) input and output stream. For more information on histroy-oriented specification we refer to [Ka20]. Such a specified component is highly underspecified. Going closer to the implementation, we reach the LLRs. Component specifications are given as concrete automata or as composition of sub-components. The last step of the LLRs and closest to the implementation is the executable model. In the executable model, every atomic component is specified by a deterministic automaton and all non-atomic components are specified by composition. Per generator, such a model can be transformed directly into source code in, e.g., Java or Python [HRR12]. The generated code identifies as implementation according to DO-178.

We break down the HLRs and LLRs in so many sub-levels until we reach a granularity with only identified, usual refinement steps (see Sect. 2.5) left. Due to our case study, we have three HLR levels and one LLR level here. This level of granularity is sufficient to automate the verification.

Formal verification can then be applied in many small steps, see Fig. 2. This method scales far better than to verify in one big step that an implementation fulfills the system requirements. The latter is even often not automatable at all. The goal of the verification based on FOCUS is to show on a high and less detailed level, that HLRs are sufficient to satisfy the SRs. After that, it only has to be proven, that the LLRs refine the HLRs and that the implementation fulfills the LLRs. If that is the case, then the implementation also satisfies the SRs.

2.2 Abstraction Levels on the Running Example

The SRs of the case study were written in Sect. 1.3 informally, where the Computer component of Fig. 1 is a black box. One step further, at the most abstract high-level requirements (HLR 1), the Computer component was decomposed into I/O components, the Message-Router component, and the DLUF component. The DLUF component, which at this level is still a black box, is specified by a history-oriented specification style. Afterwards, at the level of HLR 2, the DLUF component was decomposed into a composition of four Scheduler components. Then, at the level of HLR 3, each of the Scheduler components was decomposed into compositions of Buffer components and Capacity components. Decomposing Buffer and Capacity further, would be possible but not helpful in managing the complexity of the system. Thus, the architecture will be fixed at this point (meaning that no further decomposition will occur). The specification of each non-atomic component (such as Scheduler1) is given through the composition of its sub-components, and the specification of the atomic components (such as Capacity1) is given by history-oriented specifications. Only on the low-level requirement layer we have the atomic components specifications refined by writing these as (potentially still non-deterministic) state-oriented specifications. Finally, at the executable model level, we have the specifications of atomic components written as deterministic automata (such as Fig. 6) (as refinements of the non-deterministic automata of the layer above). These automata are deterministic with regard to behavior specification, but still non-deterministic with regard to processing order (this further dimension of non-determinism is made deterministic only afterwards during simulation, see Sect. 2.6.1).

2.3 Semantic Domain of Interactive Components

2.3.1 Streams

To prepare the introduction of event-based specifications, we recall that FOCUS builds on streams representing the history of communication between components. A stream is a finite or infinite sequence of messages.

Definition 2.1 (Untimed Streams)

The set of all untimed streams is defined as $M^\omega := M^* \cup M^\infty$. *With* $M^* := \bigcup_{n \in \mathbb{N}}([1, ..., n] \rightarrow M)$ *describing all finite streams and* $M^\infty := \mathbb{N}_+ \rightarrow M$ *being the set of all infinite streams.*

We differentiate between untimed and timed streams. The latter expands a sequence of messages by time ticks. A tick ($\surd$) is used to represent time progression. In between two ticks any finite number of messages is possible. Thus, ticks model a discrete notion of time. We use these to define the version of timed streams we find most suitable to use. The added value of timed streams is the possibility of modeling timeouts. And time modeling is necessary in our case study, since the capacities for the priority levels need to be reset after each cycle.

Definition 2.2 (Timed Streams)
Timed streams are defined as: $M^{\underline{\omega}} := (M \cup \{\surd\})^\omega$. $M^{\underline{\omega}}$ *has a prefix order. The subset of finite timed streams is defined as* $M^{\underline{*}} := (M \cup \{\surd\})^*$.

2.3.2 Stream Bundles

Stream tuples describe fixed component interfaces and are thus not sufficient to specify components with arbitrary interfaces (such as the merge specification in Sect. 2.4.1). We present so-called stream bundles as an isomorphic structure, that allows such generalizations. A stream bundle is a mapping from channel names to streams and thus allows the association of streams and channels. A small example is the component shown in Fig. 3. The component has two input streams, one containing boolean values and the other natural number values. An isomorphic structure to a stream tuple as input is the following:

$$\mathbb{B}^\omega \times \mathbb{N}^\omega \cong \{sb : \{c_1, c_2\} \rightarrow \mathbb{B}^\omega \cup \mathbb{N}^\omega | sb(c_1) \in \mathbb{B}^\omega \wedge sb(c_2) \in \mathbb{N}^\omega\}$$

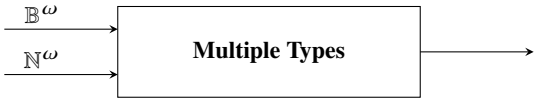

Fig. 3: Component with two input streams of different types .

The stream bundle may only map to type correct streams. In this example, $sb(c_1)$ may never contain any natural numbers and vice versa. This leads to the final formula for stream bundles. The notation C^Ω is used for stream bundles, where C is a set of channels.

Definition 2.3 (Timed Stream Bundles)
Let the function $sValues : M^{\underline{\omega}} \rightarrow P(M)$ *return the set of messages of a stream. And let* $cType : C \rightarrow P(M)$ *return the type of a channel. Let* C *be a subset of all channel names and* $M := \bigcup_{c \in C} cType(c)$. *Timed stream bundles are defined as* $C^{\underline{\Omega}} := \{sb \in C \rightarrow M^{\underline{\omega}} | \forall c \in C : sValues(sb(c)) \subseteq cType(c)\}$. *The set of finite timed stream bundles* $C^{\underline{\Phi}} \subset O^{\underline{\Omega}}$ *is defined as* $C^{\underline{\Phi}} := \{sb \in C \rightarrow M^{\underline{*}} | \forall c \in C : sValues(sb(c)) \subseteq cType(c)\}$.

2.3.3 Stream Bundle Processing Functions

The behavior of a component is algebraically described by stream processing functions (SPFs). An SPF has the signature $f : I^\Omega \rightarrow O^\Omega$, where I notates the input channels and

O the output channels. SPFs are bundle-to-bundle functions with fixed input and output channels. The behavior of a component is deterministic, if it is described by exactly one SPFs. To allow underspecification, the behavior is specified as stream processing specifications (SPSs), which is a set of SPFs representing all possible specifications. If an SPS contains no SPF, it is inconsistent and not realizable.

A function should represent realistic behavior. It is not allowed to look into the future. This means an SPF should be weak causal, i.e. the input until the ith time unit defines the complete output until that time unit. The following definition expresses weak causality, whereby $sb \downarrow i$ returns the first i time slices of the stream bundle sb.

Definition 2.4 (Weak causality)
A function f is weak causal, iff $\forall sb_1, sb_2 \in C^{\underline{\Omega}} : i \in \mathbb{N} : sb_1 \downarrow i = sb_2 \downarrow i \Rightarrow (f\ sb_1) \downarrow i = (f\ sb_2) \downarrow i$

To enforce, that every output is produced with some delay (thus giving precise meaning to feedback loops), weak causality can be strengthened into strong causality:

Definition 2.5 (Strong causality)
A function f is strong causal, iff $\forall sb_1, sb_2 \in C^{\underline{\Omega}} : i \in \mathbb{N} : sb_1 \downarrow i = sb_2 \downarrow i \Rightarrow (f\ sb_1) \downarrow i + 1 = (f\ sb_2) \downarrow i + 1$

The input until time unit i completely determines the output until $i + 1$. Any strong causal function is clearly weak causal. The semantics of event components is defined as SPFs.

Proposition 2.1 (Semantics of timed event-based processing components)
The semantics of timed event-based processing components are (sets of) SPFs of the type $TSPF := \{f \in I^{\underline{\Omega}} \rightarrow O^{\underline{\Omega}} | f \text{ is weak causal}\}$ *for* I, O *being the input and output channels.*

2.4 Signatures of Atomic Event-Based Processing Components

We identified as a necessity, that event-based processing components work on timed streams. Thereby, one event is a message on a channel or a tick. By receiving an event as input, a reaction to exactly this event is triggered at the component. Ticks are synchronized and are processed, when there is a tick on every input channel. An event component reacts to each event. That includes messages and ticks. This is contrary to the port automata presented in [Bu19], where the processing works time-slice-based. Time-slice-based components do not react to each event individually. Instead, they react only to whole time slices and output only whole time slices. The possibility to react event-based allows for a more intuitive way of modeling automata. Also, whereas time-slice-based systems, especially the time-synchronous ones, depict hardware systems, event-based systems depict software systems.

This section proposes a theory for event-based processing components. The focus lies on the signatures and infrastructure of atomic event-processing components. The shown proposal is compatible with composition, because the underlying semantics corresponds to SPFs.

Atomic event-based processing components are specified by automata. The semantics of components in FOCUS are mappings of input histories to output histories. For the same input histories different outputs can be produced depending on the order of arrival of input. Because the order of arrival is underspecified, this also holds for the semantics of event-based components.

Example 1 *An identity component with two input channels and one output channel, that just forwards all input messages, gets the streams* $\langle a \rangle$ *and* $\langle b \rangle$ *as inputs. Possible outputs are* $\langle a, b \rangle$ *and* $\langle b, a \rangle$.

This non-determinism is filtered out of the event automaton by sequentially composing a merge component with it. The merge specification produces all possible orders for multiple histories. The order of the merged stream is the processing order for the event automaton. It follows, that an atomic $m \times n$ event component consists of a $m \times 1$ merge component sequentially concatenated with a $1 \times n$ event automaton as shown in Fig. 4.

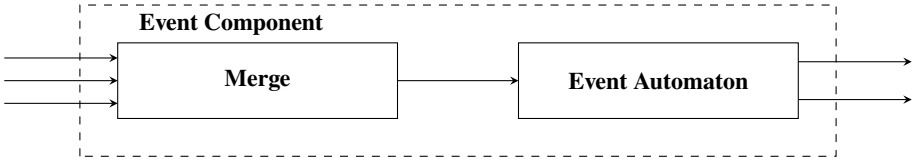

Fig. 4: Structure of an Event Component .

2.4.1 Merge Specification

The merge specifications merges messages from different channel histories into one history, thus, giving a process order for the event automaton. The merge process is subject to multiple requirements: The minimal number of time intervals over all input streams must be preserved and all transmitted messages of any channel must maintain their original order and be transmitted in the same time interval. Since there are multiple merge possibilities, the merge component is highly underspecified.

We can then specify the $Bundle\_Merge$ as follows: Input is the stream bundle over the set of channel names I. Output is the stream bundle over the channel con, which identifies the channel connecting the merge component with the subsequent event automaton. The specification contains all the functions, that fulfill the tick count property [3] and the prefix property[4].

Following functions ease the specification: The *timedFilter* function filters the input stream consisting of 2-tuples (channel, message) by a given channel. The *tick count* function $\#_{\surd} :: I^{\underline{\Omega}} \rightarrow \mathbb{N} \cup \{\infty\}$ returns the minimal count of ticks of a stream in a timed stream bundle.

[3] The tick count property states, that all complete time intervals are transmitted by the merge function.
[4] The prefix property requires, that messages from each channel are transmitted in the correct order.

$$Bundle\_Merge :: P(I^{\underline{\Omega}} \to \{con\}^{\underline{\Omega}})$$
$$Bundle\_Merge := \{f \in I^{\underline{\Omega}} \to \{con\}^{\underline{\Omega}} | \forall sb \in I^{\underline{\Omega}} : \#_{\surd}(f(sb)) = \#_{\surd}(sb) \wedge \forall sb \in I^{\underline{\Omega}}, c \in I : timedFilter(f(sb)(con), c) \sqsubseteq sb(c)\}$$

2.4.2 Signatures of Event Automata

We can now define the signatures of the event automaton.

Definition 2.6
The syntax of a timed event automaton is a 5-tuple (S, {con}, O, δ, Init).

- *S is the not empty set of states.*
- *{con} is the set consisting of the single input channel with* $cType(con) = (C \times M) =: M_{in}$ *and* $(C \times M)$ *is the set of tuples with channel name and message.*
- *O is the set of output channels.*
- $\delta \subseteq S \times \{con\}^{\underline{\Theta}} \times S \times O^{\underline{\Phi}}$ *is the transition relation.*
- $Init \subseteq S_0 \times O^{\underline{\Phi}}$ *and* $S_0 \subseteq S$ *is the set of initial states with initial output.*

Because the input stream bundle has only one channel and per transition only one event is read by the automaton, the $\surd$ can be interpreted as a normal message. It is however fixed, that transitions with $\surd$ as input event start their output with a $\surd$ on all output channels. It holds

$$(s, \surd, t, out) \in \delta \Rightarrow out = sbConc(\surd^{\underline{\Omega}}, out') \wedge out' \in O^{\underline{\Phi}},$$

where *sbConc* concatenates two stream bundles and $\surd^{\underline{\Omega}}$ is the stream bundle containing one tick on every channel.

The semantics can be represented as (set of) SPFs. The transformation is defined as a (greatest) fixed point calculation of the corresponding functional. The semantics is given with the mapping to stream processing functions

$$[\![...]\!] : (S, \{con\}, O, \delta, Init) \to P(\{con\}^{\underline{\Omega}} \to O^{\underline{\Omega}}).$$

Reacting to ticks allows to model timeouts, which is relevant for many systems. One such component is a warning module, that outputs a warning message after not receiving system critical input between any two ticks.

2.4.3 Causality in Feedback Loops

A weak causal event component allows produced output to depend on input received during the same interval. Event components may react instantaneously, which is not always desired.

One problem for feedback components is the merge anomaly as presented in [BA81]. A feedback message may get processed before its corresponding input message, which obviously should be impossible behavior. In DLUF, this would correspond to a system run, where a packet gets acknowledged before it is even sent.

Strong causality enforces a delay of at least one time slice for the output of components. Then, in a feedback system, a $\surd$ separates the arrival of the input from the arrival of the corresponding feedback. Thus, the merge anomaly cannot occur.

2.5 Underspecification and Refinement

Underspecification and refinement are important for the reusability of components and are used in many development processes. One form of underspecification is non-determinism of the processing order. Another form of underspecification is behavior underspecification. By removing underspecification, a component is refined. The following techniques cover the usual refinement steps encountered when specifying using automata. Since automaton signatures are preserved from [Ru96], its refinement calculus can be transferred, enabling:

- Removing initial states, as long as it does not result in the empty set.
- Reducing transition options reduces non-determinism, as long as there is at least one transition for every state and input.
- Adding transitions to remove partiality. An automaton is partial if for some state and some input there exists no transition.
- Adding or removing unreachable states. This is a refinement step preserving the semantics.
- A state can be split into other states to create a new state set for the refined event automaton. This refinement step allows cloning states.

This syntactical changes have a corresponding semantical impact. Consider for example removing one of the transition options of a non-deterministic automaton (starting in a state, the same input message can lead to different outputs or different new states). Then, after applying the presented semantical mapping of Sect. 2.4.2 to both automata, the new resulting set of stream processing functions will be a subset of the previous one [Ru96].

2.6 MontiArc for a User-Friendly Frontend

FOCUS is powerful, but has a complex syntax, whose formulae are hard to read and write for those not intimately familiar. Model-based approaches can alleviate those shortcomings by providing a DSL to intuitively describe systems and properties. We propose the use of an ADL called MontiArc [HRR12]. MontiArc is a particularly good fit, as its underlying semantics is FOCUS and it has already been demonstrated to model FOCUS systems in a time-synchronous mode [Bu19]. MontiArc has also been integrated into a formal verification framework for the time-synchronous case [Ka20].

MontiArc has a textual and also a graphical representation (see the architecture in Fig. 1). To cope with the challenges presented in Sect. 1.3, particularly the event-driven nature of the problem domain (software), we propose an adaptation of the modeling language MontiArc to facilitate modeling of event-based systems. The original syntax of MontiArc was presented in [Bu19] and most recently detailed for the time-synchronous case in [Ka20]. In this work, MontiArc is extended such, that transitions react to events arriving on a particular port. The example in Fig. 5 neatly demonstrates this new capability by modeling

a one-time random number generator. The event-transition in line 9 denotes, that when the automaton is still in its initial state *Start* and an event arrives on the *signal* input port, no further guard must be satisfied ("[ ]") to switch to state *End* and a random number on output port *number* is sent.

```
1  component OneTimeRandom {
2    port in unit signal, out int number;
3
4    automaton {
5      initial state Start, state End;
6
7      Start -> End signal [ ] / { number = random(); }
8    }
9  }
```

Fig. 5: MontiArc Model of an event-based one-time random number generator.

New restrictions apply to this event-based extension of MontiArc. Each transition is only able/allowed to query and process data from its designated event port. The guard of a transition may not reference any port except the event port of that transition. The effects may not use any port besides the event port to calculate their values. Transitions may further denote no event port. Omission of an event port denotes a reaction to time passing (“tick” event). Such transitions may not reference *any* input port in their guard or effects block.

It is also important to note, that the MontiArc user does not explicitly need to model the merge (see Sect. 2.4), but only the automaton. The choice for simultaneous arrivals is left non-deterministic. A scheduler is only implemented, when the model gets converted into code by a code generator.

2.6.1 Simulation and Testing

While formal verification gives highest assurance, it is also very costly and prohibitively expensive, if one erroneously attempts to prove a false statement. Simulation and testing provide tools to give confidence, before a formal verification is attempted. They can, for example, help find counterexamples, before verification is even attempted. Executable components are thus essential to a successful industry application of formal verification and we propose the following blueprint for an executable event-based infrastructure.

Total non-deterministic automaton can be made executable through use of arbitration. For every scenario, where non-determinism occurs, i.e., where multiple initial states or transitions are active, one selects a single (or a finite subset) state/transition based on some arbitration mechanism (select the first, use random seeds, etc.).

Then, test engineers or (automated) counterexample finders can **test an event automaton** by providing a sequence of tuples containing channel name and message each. This effectively provides the input via the bridge channel between merge component and event automaton

(see Sect. 2.4). Scheduling, i.e., merge, is not part of the simulation here. Instead, the test input is representative of several scenarios.

For **simulating non atomic event-components**, the merge specification must be refined to a deterministic merge. The merge should mirror the non-deterministic behavior to improve the chances of finding counterexamples, e.g., by employing different arbitration methods as explained before. With this blueprint, one is now able to simulate, test, and find counterexamples for MontiArc event components via a mapping to FOCUS.

3 Application to the Case Study

The DLUF system [Ka22] can now be modeled as an event-based processing component in a timed environment, where one time slice symbolizes the length of one transmission cycle. To model a complex system, first, the SRs get specified. Then, stepwise the HLRs and eventually the LLRs are worked out. During the process, the system gets hierarchically decomposed and the sub-components are refined. Fig. 2 depicts the three hierarchical layers. The HLRs specify each of the non atomic components as compositions of the sub-components.

3.1 Implementation of Capacity and Buffer

DLUF has multiple abstraction levels and there are several forms of representation for one component. But because our presented theory focuses on the event automata whereas [Ka22] deals more with handling HLR, it is sufficient here to only look at the atomic components Buffer and Capacity in detail to demonstrate the methodology. A implementation or LLR are provided by modeling a *Scheduler* component for each priority level. The *Scheduler* consists of a *Buffer* and a *Capacity* component and specifies, that only a certain total amount of data can be sent in each time slice. Packets, that cannot be sent anymore, are saved in the *Buffer* component and will be sent again to *Capacity* in the next time slice. The MontiArc realization of both *Buffer* and *Capacity* is shown in Fig. 6.

Capacity models, that a given maximum transmission capacity *max* is not overstepped in a time interval. It is a 1×2 component and has an internal variable to store the current available *capacity* in a time slice. Packets are forwarded from *input* to the output port *output*, if the size of the packet is still within the available capacity. If so, the capacity gets reduced by the size of the packet and an acknowledgment is sent on the port *status* to the *Buffer*. Otherwise, the component sends a not-acknowledgment and ignores the packet and every further incoming packet in that time slice. At the beginning of each time slice, the internal variable is reset to the maximum capacity. Noteworthy is the need for a reaction to a timeout. The *Buffer* component stores incoming messages and forwards them to *Capacity* via the connecting channel. It is a 2×1 component with the input ports *input* of type Packet and *status* to receive acknowledgment from the *Capacity* component. The output port *output* is connected to the port *input* of *Capacity*. The incoming messages on *input* get stored in an internal list *b*, until the respective acknowledgment is received. If the acknowledgment to a message does not arrive at *Buffer* in the next time slice, the message is retransmitted.

```
component Capacity(KByte max) {
  port
    in  Packet input,
    out Packet output,
    out STATUS status;

  Kbyte capacity;

  automaton {
    state S;
    state Block;
    initial S / {
      capacity = max;
    }

    S input [
      input.size()≤capacity] /
    {
      output = input;
      status = ACK;
      capacity -= input.size();
    }
    S→Block input [
      input.size()>capacity] /
    {
      status = NAK;
    }
    Block input;
    S       / capacity = max;
    Block→S / capacity = max;
  }
}
```

```
component Buffer {
  port
    in  Packet input, STATUS status,
    out Packet output;

  List<Packet> b; Integer cnt;

  strong automaton {
    state Send, Wait;

    Send input / {
      b=b.append(input);
      output=input; cnt++;
    }
    Wait input / b = b.append(input);
    Wait→Send status [status = NAK] /
      { output = b; cnt = b.size(); }
    Wait status [cnt>1&&status==ACK] /
      { b = b.tail(); cnt=cnt-1; }
    Wait→Send status [cnt <= 1
      && b.size()>1 && status=ACK] /
    {
      b=b.tail(); output=b.tail();
      cnt=b.size();
    }
    Wait→Send status [status==ACK
      && cnt≤1 && b.size()≤1] /
      { b=b.tail(); cnt=0; }
    Send       [cnt == 0];
    Send→Wait [cnt > 0];
  }
}
```

Fig. 6: LLR of the *Buffer* (left) and *Capicity* (right). Both are specified as MontiArc event automata.

Buffer is a strong causal event component. This is necessary for the implementation of the composition, as it forms a feedback loop between *Buffer* and *Capacity*.

3.2 Evaluation

A prototype tool for the automatic formal verification of such an event-based specified system is presented in [Ka22] with SysML as an alternative to MontiArc. With help of such a tool, the liveness and fairness property of the system can be proven formally. Our system meets the SRs stated in Sect. 1.3. Messages of all users are transmitted time and time again, i.e., no user gets ignored forever. This is guaranteed by the liveness property of the system. Each priority level has a certain capacity, that can be used in each cycle. Thus, it is not possible, that low prioritized packets starve, because of too many incoming high priority packets. One cycle is represented by exactly one tick, which simulates the passing of 100

ms. With the capacities set to 100, 200, 300, and 400 KBytes, the system has a maximum data rate of 10 MByte/s. Because the capacity limit of a higher priority is larger than the ones of all lower priorities, they are handled prioritized. Thus, DLUF fulfills the important prioritization and fairness properties. Furthermore, the packets can have any format and the priorities can be assigned to the user. DLUF is also performance optimized, because if there is enough capacity in a cycle, the messages can be sent directly, independent of the priority level. The Buffer components handle the storing of packets, until they are sent successfully. We specified a MontiArc system with FOCUS semantics, that sufficiently satisfies all SRs.

4 Conclusion and Takeaways

For event-based safety-critical software systems a high-confident development can be reached by building on an established sound theory and providing necessary extensions, and by providing an ADL and appropriately extending it for a user-friendly modeling. Therefore, data structures of event-based components were introduced, in particular event automata represent a state-oriented specification to model stateful behavior. The stream data type is extended to timed streams by a special symbol, which enables reaction to absence of messages and modeling timeouts. By using strong causal functions, which are defined over timed streams, a delayed reaction on an input message is enabled, thus giving a well-defined semantics of feedback loops. Building on these, the specification method of event-based components is introduced, by presenting the signatures of the merge component and of event automata. A timed specification of the merge component does increase the complexity of the specification, but it circumvents a known implementation issue of [Br93] of untimed merging. Using (strong causal) SPFs (as opposed to relational specifications) circumvents the merge anomaly. Specifying components as functions over (timed) stream bundles (instead of stream tuples) solves the problem of defining a general (with respect to number of channels) merge specification. Since signatures were extended in a structure-preserving way, a well-established refinement calculus of [Ru96] can be carried over to event automata. The frontend DSL MontiArc was adapted to enable an event-driven specification and simulation and has clear FOCUS based semantics.

Event-based systems provide major advantages compared to time-synchronous ones. They can react to every incoming event individually and immediately. This is demonstrated in the case study, where modeling reactions in a time-synchronous setting would have been tedious and error prone: One would have to specify the system's reaction for every combination of incoming messages. The *Buffer* would need to declare 7 additional transitions. This further underlines the point of user-friendly architecture description of systems, and in particular demonstrates the potential industry applicability of this method. The formal semantics of our approach allow to create tool-chains, as in [Ka22], that generate formal proofs for event-based systems. We can replace or complement costly tests by formal verification and thus lower the certification costs.

Finally, the specification of DLUF contains multiple refinement steps. The correctness of these has to be shown to verify the system. For this, an encoding of the complete event-based

theory in the theorem prover Isabelle building up on [Ka22] is ongoing work. Furthermore, a generator translating MontiArc event automata into Isabelle event automata would be a next step towards automatizing event-based model analysis.

Acknowledgments

This work was supported by the German Ministry for Education and Research (BMBF) in the SpesML project (https://spesml.github.io/index.html/, last access 14/12/2022, grant number 01IS20092B).

Bibliography

[AL94] Abadi, Martín; Lamport, Leslie: Open Systems in TLA. In (Anderson, James; Peleg, David; Borowsky, Elizabeth, eds): Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing - PODC '94. ACM Press, New York, New York, USA, pp. 81–90, 1994.

[BA81] Brock, J Dean; Ackerman, William B: Scenarios: A Model of Non-Seterminate Computation. In: International Colloquium on Formalization of Programming Concepts. Springer, pp. 252–259, 1981.

[Be00] Berry, Gérard; Bouali, Amar; Fornari, Xavier; Ledinot, Emmanuel; Nassor, Eric; de Simone, Robert: ESTEREL: a formal method applied to avionic software development. Science of Computer Programming, 36(1):5 – 25, 2000.

[BKR09] Becker, Steffen; Koziolek, Heiko; Reussner, Ralf: The Palladio Component Model for Model-Driven Performance Prediction. Journal of Systems and Software, 82:3–22, 01 2009.

[Bo21] Boehm, Wolfgang; Broy, Manfred; Klein, Cornel; Pohl, Klaus; Rumpe, Bernhard; Schröck, Sebastian, eds. Model-Based Engineering of Collaborative Embedded Systems : Extensions of the SPES Methodology. Springer, Cham, Switzerland, 2021.

[Br92] Broy, Manfred; Dederichs, Frank; Dendorfer, Claus; Fuchs, Max; Gritzner, Thomas F.; Weber, Rainer: The Design of Distributed Systems - An Introduction to FOCUS. Citeseer, 1992.

[Br93] Broy, Manfred: Functional specification of time-sensitive communicating systems. ACM Trans. Softw. Eng. Methodol., 2:1–46, 1993.

[BR07] Broy, Manfred; Rumpe, Bernhard: Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. Informatik-Spektrum, 30(1):3–18, 2007.

[Bu19] Butting, Arvid; Kautz, Oliver; Rumpe, Bernhard; Wortmann, Andreas: Continuously analyzing finite, message-driven, time-synchronous component & connector systems during architecture evolution. Journal of Systems and Software, 149:437–461, 2019.

[Bu20] Buerger, Jens Christoph; Kausch, Hendrik; Raco, Deni; Ringert, Jan Oliver; Rumpe, Bernhard; Stüber, Sebastian Wolfram; Wiartalla, Marc: Towards an Isabelle Theory for distributed, interactive systems : the untimed case. Technical report, Düren, 2020. Weitere Reihe: Technical report / RWTH Aachen University, Software Engineering Group. - Zweitveröffentlicht auf dem Publikationsserver der RWTH Aachen University 2022.

[Ca87] Caspi, Paul; Pilaud, Daniel; Halbwachs, Nicolas; Plaice, John: Lustre: A Declarative Language for Programming Synchronous Systems. In: POPL. 1987.

[Dz16] Dziwok, Stefan; Pohlmann, Uwe; Piskachev, Goran; Schubert, David; Thiele, Sebastian; Gerking, Christopher: The MechatronicUML Design Method: Process and Language for Platform-Independent Modeling. Technical Report tr-ri-16-352, Software Engineering Department, Fraunhofer IEM / Software Engineering Group, Heinz Nixdorf Institute, Zukunftsmeile 1, 33102 Paderborn, Germany, December 2016. Version 1.0.

[Ho78] Hoare, Charles Antony Richard: Communicating sequential processes. In: The origin of concurrent programming, pp. 413–443. Springer, 1978.

[HR04] Harel, David; Rumpe, Bernhard: Meaningful modeling: What's the semantics of Semantics? Computer, 37:64 – 72, 11 2004.

[HRR12] Haber, Arne; Ringert, Jan Oliver; Rumpe, Bernhard: MontiArc - Architectural modeling of interactive distributed and cyber-physical systems, volume 2012,3 of Technical report / Department of Computer Science, RWTH Aachen. RWTH and Technische Informationsbibliothek u. Universitätsbibliothek and Niedersächische Staats- und Universitätsbibliothek, Aachen and Hannover and Göttingen, 2012.

[Ka20] Kausch, Hendrik; Pfeiffer, Mathias; Raco, Deni; Rumpe, Bernhard: An Approach for Logic-based Knowledge Representation and Automated Reasoning over Underspecification and Refinement in Safety-Critical Cyber-Physical Systems. In: SE-WS 2020: Software Engineering workshops 2020 : combined proceedings of the workshops at Software Engineering 2020, co-located with the German Software Engineering Conference 2020 (SE 2020) : Innsbruck, Österreich, March 05, 2020. volume 2581 of CEUR workshop proceedings, 17. Workshop Automotive Software Engineering, Innsbruck (Austria), 24 Feb 2020 - 25 Feb 2020, [RWTH Aachen], [Aachen, Germany], Feb 2020.

[Ka21a] Kausch, Hendrik; Michael, Judith; Pfeiffer, Mathias; Raco, Deni; Rumpe, Bernhard; Schweiger, Andreas: Model-Based Development and Logical AI for Secure and Safe Avionics Systems: A Verification Framework for SysML Behavior Specifications. Aerospace Europe Conference 2021, Warsav (Poland), 23 Nov 2021 - 26 Nov 2021, pp. [1]–9, Nov 2021. Hybride Konferenz.

[Ka21b] Kausch, Hendrik; Pfeiffer, Mathias; Raco, Deni; Rumpe, Bernhard: Model-Based Design of Correct Safety-Critical Systems using Dataflow Languages on the Example of SysML Architecture and Behavior Diagrams. In: SE-SE 2021 : Software Engineering 2021 Satellite Events - Workshops and Tools & Demos : Braunschweig/Virtual, Germany, February 22 - 26, 2021 : Proceedings of the Software Engineering 2021 Satellite Events / Edited by Sebastian Götz, Lukas Linsbauer, Ina Schaefer, Andreas Wortmann. volume 2814 of CEUR workshop proceedings, 3. Workshop on Avionics Systems and Software Engineering, online, 22 Feb 2021 - 26 Feb 2021, RWTH Aachen, Aachen, Germany, pp. 1–22, Feb 2021.

[Ka22] Kausch, Hendrik; Pfeiffer, Mathias; Raco, Deni; Rumpe, Bernhard; Schweiger, Andreas: Correct and Sustainable Development Using Model-based Engineering and Formal Methods. In: 2022 IEEE/AIAA 41st Digital Avionics Systems Conference (DASC). pp. 1–8, 2022.

[Kr19] Kriebel, Stefan; Raco, Deni; Rumpe, Bernhard; Stüber, Sebastian: Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy's Streams Become Feasible? In: [Software Engineering (SE) und Software Management (SWM), SE SWM, 2019-02-18 - 2019-02-22, Stuttgart, Germany]. BMW Group, Chair of Software Engineering at RWTH Aachen, pp. 87–94, Feb 2019.

[KRK13] Kounev, Samuel; Rathfelder, Christoph; Klatt, Benjamin: Modeling of Event-based Communication in Component-based Architectures: State-of-the-Art and Future Directions. Electronic Notes in Theoretical Computer Science, 295:3–9, 2013. Proceedings the 9th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA).

[Le16] Lee, Edward A.: Fundamental Limits of Cyber-Physical Systems Modeling. ACM Trans. Cyber-Phys. Syst., 1(1), nov 2016.

[Mi89] Milner, R.: Communication and Concurrency. Prentice-Hall, Inc., USA, 1989.

[Mi99] Milner, Robin: Communicating and mobile systems: the pi calculus. Cambridge university press, 1999.

[Or22] Ortner, Philipp; Steinhöfler, Raphael; Leitgeb, Erich; Flühr, Holger: Air Traffic Simulation and Modeling Framework. In: 2022 International Conference on Broadband Communications for Next Generation Networks and Multimedia Applications (CoBCom). pp. 1–5, 2022.

[Ra22] Rath, Amelie: A Theory of Event-Based Processing and an Application on an Aerospace Case Study. Bachelor's thesis, RWTH Aachen University, 2022.

[Re12] Reisig, Wolfgang: Petri nets: an introduction, volume 4. Springer Science & Business Media, 2012.

[RHK21] Rumpe, Bernhard; Hölldobler, Katrin; Kautz, Oliver: MontiCore Language Workbench and Library Handbook : Edition 2021, volume 48 of Aachener Informatik-Berichte, Software Engineering. Shaker, Düren, 2021. Zweitveröffentlicht auf dem Publikationsserver der RWTH Aachen University 2022.

[Ru96] Rumpe, Bernhard: Formale Methodik des Entwurfs verteilter objektorientierter Systeme. PhD thesis, Zugl.: München, Techn. Univ, Zugl. München, 1996.

[RWa] RWTH, SE Chair: , Modeling Software Architecture & MontiArc. https://se-rwth.github.io/research/Software-Architecture/.

[RWb] RWTH, SE Chair: , MontiArc. https://github.com/MontiCore/montiarc.

[RWc] RWTH, SE Chair: , MontiCore - Language Workbench and Development Tool Framework. https://monticore.github.io/monticore/.

[RWd] RWTH, SE Chair: , MontiCore Repository. https://github.com/orgs/MontiCore/repositories.

[Sc20] Schopp, U.; Schweiger, A.; Reich, M.; Chuprina, T.; Lucio, L.; Bruning, H.: Requirements-based Code Model Checking. In: 2020 IEEE Workshop on Formal Requirements (FORMREQ). IEEE Computer Society, Los Alamitos, CA, USA, pp. 21–27, sep 2020.

[Sp22] SpesML (BMBF): , Official SpesML Project Site. https://spesml.github.io, 2022.

[vdB94] von der Beeck, Michael: A comparison of Statecharts variants. In (Langmaack, Hans; de Roever, Willem-Paul; Vytopil, Jan, eds): Formal Techniques in Real-Time and Fault-Tolerant Systems. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 128–148, 1994.

[VZ14] Voss, Sebastian; Zverlov, Sergey: Design Space Exploration in AutoFOCUS 3 – An Overview. 01 2014.