CEAS Aeronautical Journal
https://doi.org/10.1007/s13272-024-00762-6

ORIGINAL PAPER

Model-driven development for functional correctness of avionics systems: a verification framework for SysML specifications

Hendrik Kausch[1] · Mathias Pfeiffer[1] · Deni Raco[1] · Bernhard Rumpe[1] · Andreas Schweiger[2]

Received: 30 January 2024 / Revised: 23 May 2024 / Accepted: 5 August 2024

Abstract
Currently, the most widespread software quality assurance methods in the avionics domain are semi-automated reviews and testing. However, their effort grows disproportionately to the size of the system under development. Also, these methods cannot achieve exhaustive coverage due to the complexity of today's avionics systems and their potentially infinite set of combinations of possible inputs and system states. Furthermore, the later software issues are detected in the development process, the more expensive it is to fix them. To overcome these issues, a model-driven verification approach for modeling and analyzing avionics systems in early phases of the development is presented. To this end, semantics is given to SysML v2 models by a mapping to a theorem prover encoding. The development of a dedicated SysML v2 profile supporting event-driven data flow specifications, the encoding of corresponding structures in the theorem prover Isabelle, and a generator creating theorems from SysML v2 models are presented. The approach is evaluated by formally proving a representative liveness property of a hierarchical system model from the avionics domain. Since liveness properties can be negated only by infinite data sequences and thus cannot be covered exhaustively by testing, this case study demonstrates the added value for meeting typical safety requirements in the avionics domain. The results can be transferred from avionics to other domains, as well.

Keywords Model-driven development · Safety · Avionics · Formal methods · Formal verification · Theorem prover

1 Introduction

Rising automation during ground and air operations of aircraft drive the complexity of avionics systems. Nowadays, the latter's development accounts for over 30% of the overall aircraft development costs [1]. The corresponding effort is mainly generated by the strict safety (EUROCAE ED-12C/RTCA DO-178C[1]) and security (EUROCAE ED-202A/RTCA DO-326A 1) demands. These are required by the certification authorities (such as EASA for Europe or FAA for the United States of America) and cover the complete development and maintenance process and the operational phase for both software (EUROCAE ED-12C) and hardware (EUROCA ED-80/RTCA DO-254 1) of the avionics system. The major part of the avionics' development costs is accounted for by the verification phase [2]. This is caused by the types of verification techniques in place in most of the cases, e.g., testing and semi-automated reviews. However, the effort for these methods grows disproportionately with the system's size [2]. In addition, testing fails to deliver exhaustive coverage in certain scenarios. This is due to the systems' complexity and their potentially infinite combinations of possible inputs and system states. In comparison, formal verification methods are able to verify correctness for all inputs and system states.

The later in the project errors are detected and fixed, the more extensive is the corresponding effort for correcting them [3, 4]. Fixing these errors requires additional effort or increasing the development pace. Though the latter contradicts [5] the agile principle of development at constant pace,[2] it is still widely seen in industry. NB, that agile methods can and should be applied also to the modeling phase [6], because they are compliant with avionics development in line with

✉ Deni Raco
raco@se-rwth.de

1 Chair of Software Engineering, RWTH Aachen University, Aachen, Germany

2 Airbus Defence and Space GmbH, Manching, Germany

[1] As the EUROCAE and RTCA documents are technically equivalent to each other, we use only the EUROCAE reference throughout the remainder of this article for improving the readability.

[2] https://agilemanifesto.org/principles.html, last access 03/03/2023.

the standards [7] mentioned above. In addition, Beizer [8] demonstrates, that the cumulative distribution of error discoveries related to development stages can be described as an S shape. That means, that more issues are detected, as later development stages are entered [9]. The combination of these two factors (costs for fixing and distribution of issues) drives the overall development costs.

To tackle the issues mentioned above, we propose the deliberate usage of EUROCAE ED-12C's supplements, which enable the deployment of formal (RTCA DO-333/EUROCAE ED-216 1) and model-based methods (RTCA DO-331/EUROCAE ED-218 1). Aerospace industry has successfully adopted formal methods [10] for verifying properties at the code level, using model checkers and abstract interpretation, e.g., for worst-case execution time analysis [10, 11], which is also regulated by EUROCAE ED-216, or for replacing[3] testing efforts for certain properties [10]. However, the application of formal methods typically requires highly skilled personnel. Furthermore, it is inherently hard to guarantee the successful application of formal methods [1], such as in the above example. Deploying a development methodology may alleviate some of these shortcomings. Such a methodology should channel methods towards their automated application and should drive success rates by applying tactics to the system engineering process. The objective is to increase the effectiveness of formal methods and to lower the barrier of entry, effectively increasing acceptance in the avionics domain.

Software quality is determined significantly by the quality of the corresponding models [12, p. 409] used in the development process. Thus, the quality of models in the context of model-based development needs to be verified. Our approach introduces explicit means for verifying the model quality [13], since there is no widespread and accepted approach for measuring and improving model quality [12, p. 409].

1.1 Structure

Section 1.2 outlines methods, formalisms, and modeling languages for formal verification. Section 1.3 compares related work and summarizes previous work. Section 1.4 sums up the contributions of this article regarding the challenges presented in Sect. 1. Section 2 introduces a typical avionics use case representing a class of modern, software-intensive avionics systems. Section 3 provides details of our semantic foundation, event-based specifications, theorem-prover encodings, engineering process methodology, and a SysML v2 frontend. Sections 4.1 and 4.2 evaluate the applicability of the methodology, while Sect. 4.3 evaluates the verifiability. Section 4.4 covers tool qualification considerations. Finally, Sect. 5 summarizes and discusses the findings.

[3] NB, that testing can never demonstrate the absence of issues, but only their presence.

1.2 Foundations of formal methods

Formal methods can tackle the deficits regarding testing mentioned in Sect. 1. The key classes of formal methods [10] for avionics development according to EUROCAE ED-216 are abstract interpretation, model checking, and deductive methods. Abstract interpretation is the least expressive and is targeted to very specific artifacts. It requires some expertise to discharge false positives. Model checking is less expressive than theorem provers. It is mostly automated, but still requires expertise to be used successfully. Deductive methods (e.g., theorem proving) are the most powerful and most expressive formal method. They require a strong expertise and continuous interaction to be used successfully. The mentioned classes are introduced in more detail below:

Abstract interpretation [14] abstracts from software source code notation into more abstract models, enabling reasoning about certain information regarding the execution of the software itself. To use abstract interpretation as a formal method, one can use over-approximation or under-approximation [15, p. 21]. Over-approximation is capable of demonstrating the absence of defects. However, over-approximations are not able to expose software defects. They usually generate a huge number of false positives, which have to be ruled out manually. In contrast, under-approximation identifies present bugs and raises corresponding issues, but is not able to demonstrate the absence of defects. To improve these limitations of under-approximation, O'Hearn suggested to use incorrectness logic [16]. However, Ascari et al. [15] demonstrate that this logic cannot rule out under-approximation's limitations completely.

Model checking is a formal method [17, 18] able to check whether a formal system model fulfills some property specification. It can provide counterexamples for property violation by giving an execution trace that reaches a state, where the property does not hold. Model checking can also be used to check semantic differences, e.g., between functional architectures of a system [19]. The main limitation of model checkers is, that they suffer from the explosion of the state space. One can try to exploit the state space's structural regularities, e.g., by using symbolic techniques and abstractions [20]. However, such exploits fail to cover the whole state space, so they come at the cost of giving up exhaustivity. Unfoldings [21] are yet another approach for reducing the size of the state space. However, the state explosion is still an issue with systems with sequential execution.

Theorem provers, as one representative of **deductive methods**, offer the highest assurance level and have been used for verifying important properties, e.g., the safety and security properties of the complete kernel of an operating system [22]. To this end, the formal semantics of a programming language (such as C) or a modeling language (such as SysML) needs to be encoded (eventually automatically by a code generator) into the language of the selected theorem prover. The added benefit is, that one can write more than just tests covering the program functionalities. Full proofs over each potential input are possible, as well. A key advantage of theorem proving compared to model checking is that the complexity of proofs grows only linearly with the system's complexity [23]. There exist multiple theorem provers, differing in capabilities such as size of their library, strength of their logic, and their level of automation [24].

1.3 Related work

Formal verification, reasoning, and theorem proving require well-defined semantics, matching the problem domain [25]. Formalisms such as Communicating Sequential Processes (CSP) ([26], as used in e.g., [27]), Calculus of Communicating Systems (CCS) [28], π-calculus [29], Ptolemy [30], Temporal Logic of Actions (TLA) [31], Petri Nets [32] or FOCUS [33, 34] are usually used as mathematical underpinning for reasoning. The reason is their support of non-determinism, underspecification, and a notion of behavioral refinement, time-sensitive specifications, and hierarchical decomposition. In particular, decomposition is badly needed in general, otherwise the verification of a complex atomic component can quickly become unfeasible. This in turn requires compositional[4] verification, which is provided by FOCUS and our Isabelle formalization of FOCUS. In FOCUS, distributed and interactive systems consist of components exchanging messages through unidirectional channels. The semantics of a component is a (set of) stream processing functions each of which representing a potential behavior. Behavioral refinement is then represented by set inclusion. Concurrency is represented by an appropriate composition operator connecting channels. The most important reason that FOCUS is used in this article is due to the fact, that its refinement mechanism is fully compositional [34, 36]. This means, that after decomposing a system, refining the components separately, and then composing them again, the composed system will be – by construction – a correct refinement of the one before its refinement, thus saving supplementary testing and integration costs.

Time-synchronous behavior specifications [36, 37] are known to be well-suited for hardware specification and verification [38]. Meanwhile, in software applications such as the increasingly software-intensive avionics domain, an event-driven paradigm is much more common for building scalable distributed systems [39].

For a user friendly interface, modeling languages can be used to hide the complexity of the mathematical formalisms. A number of synchronous data flow modeling languages such as Esterel [40] and Lustre [41] (and its dialect SCADE) have been created for the development of reactive systems. However, due to their time-synchronous paradigm, these are rather suited for the description of hardware systems.

Further modeling languages for specifying distributed systems have been developed, such as the Palladio Component Model [42], MechatronicUML [43], AutoFocus [44] or Ptolemy [30]. However, neither of them does support event-based specifications or the latest version of the de facto standard systems engineering modeling language, SysML (v1). This paper uses SysML [45], because it is prominently used in the aerospace and automotive industry for systems engineering. In particular, a profile enabling an event-based specification style was developed for this article, since it promises to be more scalable and flexible than synchronous communication [39].

By encoding FOCUS in the interactive theorem prover Isabelle [46] and defining a transformation from SysML models into FOCUS components in Isabelle, the behavior of SysML specifications is formally defined. Isabelle enables machine-supported and automated proof searches and allows for the generation and verification of machine-based and machine-checked formal proofs. The verification of communication in distributed systems using theorem provers has been demonstrated [47]. However, the protocol being verified was manually encoded directly in the theorem prover, which requires expertise and is more error prone compared to our model-driven generative approach.

Integrating formal verification, particularly deductive methods and modeling languages is not new. The authors of [48] combine the modeling language RSML^{-e} and the theorem prover PVS via a code generator similarly to our approach. However, the modeling paradigm is synchronous and not event-based. In addition, the modeling language is not an industry standard such as SysML. There is no automation, as the proving process is manual.

SysML-Sec [49] provides a SysML profile and a model-driven toolkit to develop and formally verify embedded systems w.r.t. to safety and security concerns. However, the approach is based on the previous version of SysML (v1), while the approach described in this article resorts to its successor version with considerably extended expressiveness.

For specifying distributed software systems through the approach presented in this article, we build on our previous works. These differ from the work in this paper in certain aspects. In [50, 51] a code generator encoding class diagram syntax and semantics for describing systems was introduced, but had no tool-support for verifying properties and focused on language variability instead. References [36, 52, 53] used a time-synchronous version of the architecture description

[4] Compositionality is introduced by Carnab as *Frege's principle* [35, p. 120–121].

language (ADL) MontiArc. MontiArc is a domain specific language (DSL) based on FOCUS built using the framework MontiCore [54–56]. In contrast, SysML v2 is used in this article. Compared to our SysML time-synchronous variant in [23] a profile of SysML was extended in this article, building on our previous works [57, 58] to support event-based processing. Furthermore, [59] 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. Reference [60] focuses on the signatures for timed event-based automata, which serve as a blueprint for the implementation presented in this article. Model analysis in [11] was performed in another previous work of ours using the alternative formal methods of model checking. These do help to reduce the complexity of the verification of the developed system's correctness. However, the system requirements of a representative avionics software system treated in [57] required the development of an infrastructure to model and reason over event-based processing systems. The work presented in this article extends and elaborates on that infrastructure.

1.4 Results

This article updates and continues our previous projects (German Federal Ministry for Economic Affairs and Climate Action ASSET-2 [61] and German Federal Ministry of Education and Research SPES series [62][5]) and research on model-based verification of safety-critical properties [23, 36, 52, 53, 59, 63]. To begin closing the gap between systems engineering and formal methods, we extend our previous work by the following key novel contributions:

- Introduction of event-driven modeling in SysML v2
- Provision of an event-driven reasoning infrastructure
- A semantical mapping from SysML v2 to a data flow formalism
- A development methodology to improve the rate of successful application of formal methods
- Evaluation in a use case from the avionics domain handling liveness properties
- Automation using a Formal Integrated Development Environment (F-IDE)

In summary, we present an approach levering generative model-based formal verification using event-driven specifications. It covers the early development phases and provides means to guarantee the (a) compliance, (b) consistency, (c) verifiability, and (d) traceability between system requirements (SRs), high-level requirements (HLRs), and the design of a software architecture as well as low-level requirements (LLRs), as shown in Fig. 1.

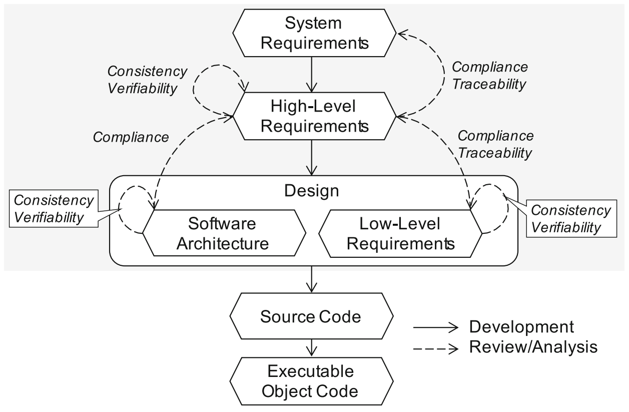

Fig. 1 The MontiBelle approach provides means to guarantee the **a** compliance, **b** consistency, **c** verifiability, and **d** traceability between SRs, HLRs, software architecture, and LLRs in the systematic design as suggested by EUROCAE ED-216

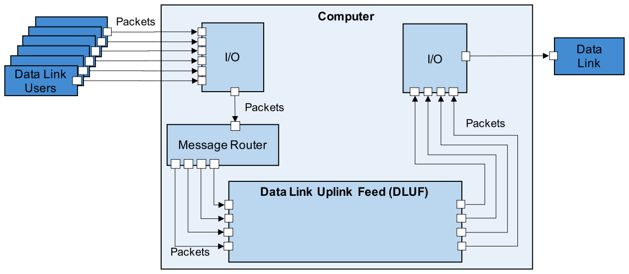

Fig. 2 Graphical representation of the DLUF system context[57]

2 Use case

As exemplary implementation used to evaluate the viability of the methods presented in this article, the development of a software functionality representative for the avionics domain (Data Link Uplink Feed (DLUF), see Fig. 2) is selected. In this system the users of a wireless connection (e.g., between an Unmanned Aerial Vehicle (UAV) and its ground station) need to transfer prioritized data packets. Table 1 introduces the SRs for the DLUF, the respective requirements type, and the corresponding verification method.

From these SRs we develop the system's boundaries, context, and data types and trace them to each SR. Figure 2 depicts the graphical overview of the system's boundaries and context. Packets are simple byte arrays of maximum size of 100 KByte (SRs 14, 17). The packets are received via the I/O element of the computer (SR 5). The packets are prioritized by a message router, i.e., forwarded to the appropriate queue according to the sender's respective prioritization (SRs 1, 4, 7). The incoming packets are processed by DLUF such

[5] https://spesml.github.io, last access 05/05/2024.

Table 1 SRs are, in accordance with EUROCAE ED-216, developed first and describe desired the architecture, functionality, and performance of DLUF

No.	Requirement	Type	Verification
1	The DLUF system shall provide a prioritization component processing messages	Functional	Test
2	This prioritization component shall ensure, that packets of all (potentially low-prioritized) users are transmitted time and time again (i.e., provision of non-starvation)	Functional	Formal method
3	The data link shall transmit packets of users with a data rate (i.e., budget) of 10 MByte/s	Performance	Test
4	Priorities between 1 and 4 shall be assigned to each user, where 1 denotes highest and 4 lowest priority	Functional	Test
5	An I/O event-based processing module shall receive the packets	Architecture	Review
6	The packets are labeled with priorities from the users	Structure	Review
7	A message router shall forward the packets to the corresponding buffer	Architecture	Review
8	Forwarding to the buffers shall depend on the message priority	Functional	Test
9	Forwarding shall be done in an event-based manner	Functional	Test
10	Packets shall be stored by the corresponding buffer to enable future retransmission	Functional	Test
11	Packets shall be attempted to be forwarded depending on the remaining capacity	Functional	Test
12	Another I/O processing module shall forward the selected packets to the data link	Architecture	Review
13	The I/O processing module shall forward the packets in an event-based manner	Functional	Test
14	The size of packets shall vary between 1 and 100,000 Bytes	Structure	Test
15	The DLUF system shall operate in cycles of length 100 ms	Performance	Test
16	Per cycle the capacities of each of the four priorities shall be 100 KByte, 200 KByte, 300 KByte, and 400 KByte	Performance	Test
17	The system shall not assume any defined packet format but treat the packets as byte arrays	Structure	Review
18	The system shall not allocate memory dynamically during runtime	Structure	Review

that higher priority packets have precedence over those of lower priority. At the same time, the balance of all priority classes has to be achieved: Higher prioritized messages shall not completely rule out the forwarding of messages of lower priority (i.e., there is no starvation of messages with low priority, SR 2). DLUF shall additionally ensure, that the maximum data rate of the link is 10 MByte/s (SR 3), which is implicitly ensured by assigning a capacity of 400 KByte, 300 KByte, 200 KByte, and 100 KByte respectively per cycle to each priority (SR 16) and enforcing the cycle length at 100 ms (SR 15). Packets also need to be stored immediately in buffers of fixed size (SRs 7–10). Packets forwarded by DLUF, i.e., that are viable to be transmitted within a transmission cycle, are finally sent via an I/O component to the $DataLink$ (SRs 12, 13).

To this end, it is required to formally verify (instead of just demonstrating the correct functionality with non exhaustive tests), that these properties hold for the overall system (instead of just subsystems) in every scenario (instead of just best-case scenarios). SRs 5–7, 12, and 17–18 are covered by appropriate system design and review. SRs 8–11, and 16 require careful design of the inner working of the DLUF system. However, SR 2, i.e., the non-starvation property, requires checking an unknown and potentially infinitely long time frame to ensure a correct DLUF system. Non-starvation is a liveness property.[6] Testing is used in industry for a lot of similar avionics properties and EUROCAE ED-12 defines several complementary certification objectives to ensure sufficient verification, when using tests, but to accomplish even higher certainty of correctness formal verification methods presented in EUROCAE ED-216 are advised [10]. It has to hold for the overall system and cannot be sufficiently verified by only checking properties of the system's parts, but requires the integration of all artifacts into a single coherent claim. SR 2 is selected for further investigation in Sect. 4, where we leverage formal methods according to EUROCAE ED-216 to achieve the safety requirements Design Assurance Level (DAL) A and show both the formal proof for this property, as well as demonstrate the application of our methodology and tool chain for making this proof feasible.

3 The MontiBelle approach

The MontiBelle approach is a collection of methods, methodologies, and tools that tackles the challenges outlined in Sect. 1. We will detail the improvements made compared to earlier publications, while mainly only referencing already published results. The order of issues presented in this section reflects the list of results presented in Sect. 1.4.

3.1 Semantic foundation

Based on a mathematical and logical foundation, Focus is a formal framework capable of specifying distributed systems at different abstraction levels. It is a methodology for the stepwise development and refinement of interactive systems [34], where streams represent communication histories between components. Furthermore, refinement is compatible with composition [65] as discussed in Sect. 1.3.

The property of Focus, that refinement is fully compositional, allows the following specification method to be used: A system can be decomposed into under-specified components. These components can then be refined individually, until an implementation is reached. After assembling these components into a complete system, the requirements of the original system automatically hold in the new system [34], as well. Thus, Focus allows for breaking down the proof complexity by applying verification at each granularity level (SRs, HLRs, LLRs, and implementation). This provides scalability as a benefit when compared to a monolithic verification of the complete system. Refinement is also transitive. One has to show, that HLRs are sufficient to satisfy the SRs, that LLRs refine the HLRs, and that the implementation fulfills the LLRs. The implementation satisfies the SRs then by transitivity. The following sections introduce the core concepts and a slightly simplified main encoding in the theorem prover Isabelle.

3.2 Timed stream bundle processing functions

This section introduces atomic components, i.e., the atomic building blocks of a system, as timed stream bundle processing functions. To this end, concepts, definitions and Isabelle encoding for (timed) streams, timed stream bundles (SBs) (a grouping concept) [34], and timed stream processing functions (SPFs) are presented, that build to a certain degree on results from [66]. An overview over the abbreviations and symbols is given in Sect. 3.4.

The most important data type in Isabelle is the **streams** domain. Streams are concatenations of messages over some alphabet and describe the history of communication channels in a system. With the keyword *domain* the stream data type in Isabelle is defined similar to the implementation of Haskell lists:

```
domain 'm stream = cons (head::"'m") (lazy rest::"'m stream")
```

An event in Isabelle is defined to allow reasoning over timed communication histories of systems in Isabelle. An event is either a message `Event 'm` or the progress of time $\surd$:

```
datatype 'm event = Event 'm | √
```

Components in distributed systems usually communicate with a multitude of other components via multiple input and output channels. A **stream bundle** is a mapping from channel names to streams and allows the association thereof. With the *pcpodef* keyword, the type of stream bundles is defined as the type of all well formed functions, i.e., mapping channels to streams, that only contain allowed messages. The notation C^Ω is used for stream bundles, where C is a set of channels and C^Φ form finite stream bundles containing only finite streams on each channel:

```
pcpodef 'cs bundle = "{f::('cs ⇒M stream). wellformed f}"
```

The behavior of a component is algebraically described by **SPFs**. An SPF has the signature $f : I^\Omega \rightarrow O^\Omega$, where I denotes the input channels and O the output channels. In Isabelle, the general type for SPFs is defined as a function mapping input bundles of a channel set $'I$ to output bundles of a channel set $'O$:

```
type_synonym ('I,'O) spf = "'I^Ω → O^Ω"
```

[6] Liveness properties prescribe, that the desired functionality is provided eventually [64].

To allow for underspecification, the behavior is specified as a stream processing specification (SPS), which is a set of SPFs representing all possible deterministic behaviors:

```
type_synonym ('I,'O) sps = "('I,'O) spf set"
```

Components defined descriptively are conform to their predicates by construction, but not always realizable. Defining contradictory requirements leads to an empty set of functions, hence, specifications can be inconsistent. For consistent specification, an automaton, that fulfills the requirements, can be defined. Since an SPS, whose elements are defined by an automata, is always consistent, a refinement relation between the automata and descriptive SPS shows consistency.

3.3 Event-driven processing

Compared to previous works [23, 67], where a time-synchronous paradigm more suited to hardware-verification is demonstrated, event-based systems are a closer match to the behavior of typical software systems, in particular in the context of cyber-physical systems. This means event-driven modeling of reactions to incoming events is a more natural fit for typical distributed software systems. Due to the time-sensitive environments found in the avionics domain, it is necessary for the correct specification of event-based systems to react to time passing. We thus propose a theory for event-based processing components. The described proposal preserves compositionality of refinement, because the underlying semantics corresponds to SPFs.

Behavior of event-driven components is modeled using state machines [68], that can react immediately to single events like incoming transmissions. After receiving such an input, the system can produce arbitrarily, but finitely many outputs and/or simultaneously change its internal state. Each transition models the immediate reaction of the automaton to incoming messages on either the data input i or the control command channel ctrl. Depending on the internal state, i.e., state of the internal memory, messages are stored and forwarded later. Events on different channels of a component might occur at the same time, i.e., within a single time frame. As event-based components react to single events, the ordering and subsequent processing of such simultaneous events is relevant to the semantics of the component. Underspecification of event order leads to underspecified semantics of event-based components. The resulting non-determinism is filtered out of the event automaton by adding a merge function, that sequences the input. A merge function produces all possible orders for multiple histories. The order of events on the merged stream determines the processing order for the event automaton and was defined in [60]. We define the signature of the event automaton as follows:

Definition 1 *[Timed Event Automaton]* The signature of a timed event automaton is a 5-tuple (S, {con}, O, δ, Init) with the following meaning:

- *S is the non empty set of states.*
- {con} is the set consisting of the single input channel with $cType(\text{con}) = (C \times M) := M_{\text{in}}$ and $C \times M$ is the set of tuples with channel name and message.
- *O is the set of output channels.*
- $\delta \subseteq S \times M_{\text{in}} \times S \times O^{\Phi}$ is the transition relation.
- Init $\subseteq S_0 \times O^{\Phi}$ 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 regular message. It is, however, fixed, that transitions with $\surd$ as input event start their output with a $\surd$ on all output channels. It holds

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

where $sbConc$ concatenates two stream bundles and $\surd^{\Omega}$ is the stream bundle "containing" just one tick on every channel. The denotational semantics of an event automaton can be represented by a (set of) SPFs. The semantics is given as a mapping to stream processing functions, was implemented according to [65], and embeds the automaton type fully in the existing Focus framework [59] in Isabelle. Behavioral refinement rules over the structure of automatons are given in [69].

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

Further implementation details, functions, and general theorems are introduced in [59].

3.4 Engineering distributed systems

By connecting components via communication channels, a distributed system can be engineered. There is three different kinds of composition types (parallel, sequential, feedback). By combining the different compositions kinds, complex distributed systems can be specified. A composition operator $\otimes$ composing SPFs and enabling sequential, parallel, and feedback compositions is defined. To handle feedback between components, the messages on feedback channels is iteratively calculated as a fixed point as defined in [34]. The signature elements $\cup$ and $-$ build the union or difference over the channel sets.

```
definition spfComp::"('I1^Ω → 'O1^Ω) → ('I2^Ω → 'O2^Ω)
→ ((('I1 ∪ 'I2) − ('O1 ∪ 'O2))^Ω → ('O1 ∪ 'O2)^Ω)"
```

The composition operator is easily lifted to SPSs by applying the composition operator in a pairwise way:

```
definition spsComp:: "('I1^Ω → 'O1^Ω) set ⇒ ('I2^Ω → 'O2^Ω) set
⇒ ((('I1 ∪ 'I2) − 'O1 ∪ 'O2)^Ω → ('O1 ∪ 'O2)^Ω) set"
(infixr '⊗') where "spsComp F G = {f⊗g | f g. f∈F ∧ g∈G }"
```

A significant challenge of applying theorem provers such as Isabelle for high-level proofs of system properties is, that these proofs usually rely on lower-level theories. It is therefore necessary to develop these lower-level theories such, that they can be used by engineers in multiple contexts. As such, constructs needs to be re-usable and operators modular (Table 2). The presented Isabelle formalizations are generic and have been successfully applied in different domains, e.g., in the verification of a door-light controller [52], a cruise control system [59], and a pilot flying system [58].

3.5 Methodology for correct dataflow architectures

The development of a system is carried out according to EUROCAE ED-12C and EUROCAE ED-216 with respect to the safety requirements level DAL A. The MontiBelle approach identifies three key classes of system models: (1) declarative specifications, (2) architecture, and (3) imperative specifications.

From SRs to Formal HLRs: To narrow the gap between typically informal SRs and formal LLRs, formal HLRs are introduced. These are formalized as declarative specifications over communication histories. For example, the formalized HLR of the non-starvation SRs, i.e., SR 2 in Table 1, can be seen in listing 1. In general, our specifications define (part of) the system's interface and give well-defined, but potentially largely underspecified formalizations of the system's behavior. Formalization of HLRs enables consistency checking in two ways. One might formally prove, that a realization exists, which matches the formulated requirements. This is done without explicitly defining the realization, but rather relying on reasoning to prove the absence of contradictions. The second way is to design a realization and show its compliance with the HLR. We propose to refine HLRs to LLR event automata, as they are consistent by construction. Showing compliance can be automated, as we'll show in Sect. 4.2.

Decomposing HLRs: To reach a feasibly fine-grained architecture, HLRs are decomposed into communication architectures of more detailed and specialized HLRs. Figure 3 shows this process exemplary for DLUF: From the HLR 2 formalizing the non-starvation SR, a decomposition of multiple schedulers, is created. These schedulers are then further decomposed into atomic blocks of buffers and capacity components. Decomposition levels are linked using refinement relations, ensuring traceability and enabling verifiability of compliance and consistency. Each refinement relation results in a proof obligation, i.e., an unfinished (yet to be proven) theorem. When refining declarative specifications to other declarative specifications or to architectures, then meeting the resulting proof obligation typically entails showing the implication between (potentially multiple) logic predicates. When refining between architectures, the compositionality of refinement in FOCUS drives automation.

Developing LLRs from HLRs: To formalize LLRs, an imperative and thus more implementation-oriented technique should be used. We propose the use of automata, specifically event-based automata for software-intensive systems. A requirement described by an automaton is consistent, as it itself describes one possible implementation. The modeling of event-based automata is done using a profile for SysML v2. Traceability and verifiability are once again achieved using refinement relations. The compliance is assured by resolving the resulting proof obligation, i.e., typically an inductive proof over (the length of) communication histories, i.e., streams.

Composing the System's LLRs: Combining architecture and LLRs into a system design is achieved by refining the final HLR architecture to an equally structured architecture composed from an LLR. The refinement of the decomposition of black boxes to the decomposition of event-automata, i.e., the refinement of the last HLR architecture to the LLR architecture, is then reduced to the compositionality of refinement in FOCUS. The development process of the architecture of LLRs is shown in Fig. 3 for DLUF.

3.6 SysML v2, code generator, and F-IDE

Developing and verifying systems in a theorem prover requires specific expertise, as does the application of an engineering process methodology. We therefore use a model-driven approach based on a SysML v2 profile. The profile implements the methodology, i.e., delivers rules and guidance for successful application. The textual SysML v2 profile is implemented using MontiCore [54–56], making models machine processable, ultimately enforcing valid models [70]. A generator automatically transforms system models and requirements into theorem prover encodings. This enables automated reasoning and reduces the risk of encoding errors. It also provides abstractions to the formal foundation by virtue of industry standard modeling languages. The tool chain is summarized in previous works [23, 57, 58].

Methodology through Modeling Language: MontiBelle ML is a SysML v2 profile dedicated to the modeling of verifiably safe and secure data-flow systems. MontiBelleML uses three steps of model conformity to guide the modeling process towards the successful application of the MontiBelle methodology. First, models are enforced to be valid SysML v2 models. This is achieved by implementing SysML v2 using MontiCore [54–56]. MontiCore is a

Table 2 Description of abbreviations and symbols

Abbreviation	Description	Abbreviation	Description
CS^{Ω}	(Timed) SB	CS^{Φ}	Finite (timed) SB
$\surd$	Progress of time	$\otimes$	Composition of SPFs
$\bigotimes$	Composition of SPSs	$\mathcal{P}$	Power set
$\subseteq$	Refinement relation	$[\![..]\!]$	Automata semantic mapping

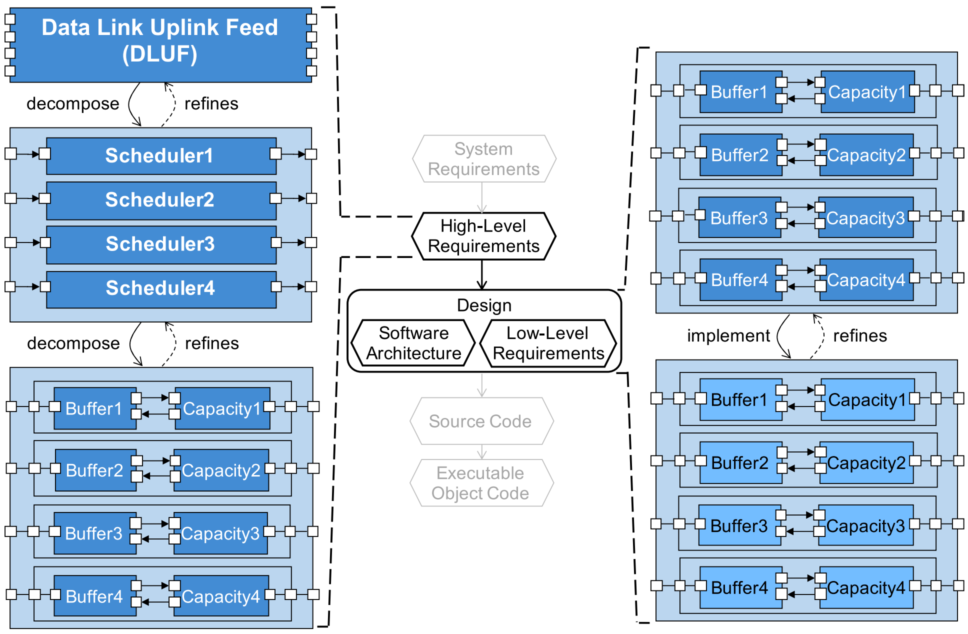

Fig. 3 HLR, LLR, and architecture development with the MontiBelle approach of DLUF

language workbench and is designed to facilitate the development of domain specific languages. The implementation of SysML v2 includes a parser and basic validation rules. Second, semantically well-founded models are enforced. Model elements are restricted to part definitions, state definitions, constraints, and composition. A rigorous type- and reference-checker using static analysis complements these restrictions. Static analysis finds errors before costly verification is attempted. Third, a methodologically sound development process is enforced. Formal refinement relations link model snapshots. Development from declarative specifications to (architectures of) imperative specifications is encouraged.

Automation through Generation: SysML models are consumed by a theorem generator. The parsed models are automatically transformed to their Focus representations, as explained in [58]. This gives formal semantics to the modeled system, enabling reasoning and deduction. The Focus representation is stored utilizing a common meta-model for all Focus based systems. This enables the re-use of a syntax-agnostic transformation. The meta-model was introduced in [23] and extended in [58]. We previously demonstrated the usefulness of an intermediary representation by adapting our transformation to other modeling languages [60].

A Formal Integrated Development Environment (F-IDE): The F-IDE prototype handles modelling, navigation, visualization, interactions with the formal backend, and automated formal verification. For model creation and editing, a language server implementation[7] was automatically generated from the SysML v2 implementation using MontiCore. The language server provides syntax highlighting, model navigation, auto completion, and error reporting. The F-IDE uses the theorem generator in the background to generate theorem provers encodings. The proof obligations are summarized in an interactive list. Verification of proof obligations are attempted by the click of a button. A color-coded light indicates their status: Verified goals are green, counterexamples red. Users are able to hand-craft theorems and hook those hand-crafted theorems into the generation process. No inconsistencies are introduced thanks to Isabelle's conservative extension mechanism.

[7] See https://microsoft.github.io/language-server-protocol/, last access 30/06/2023.

4 Evaluation

In this section, the presented MontiBelle approach to formally verify the correctness of distributed, event-based systems is evaluated. We begin by validating the correctness of our FOCUS encoding in Isabelle in Sect. 4.1. We then demonstrate the successful application of the model-driven MontiBelle methodology and tool chain to the DLUF case study in Sect. 4.2. Lastly and most importantly, we demonstrate the successful verification of the non-starvation property in Sect. 4.3.

4.1 The encoding of FOCUS

A mathematical framework could be unsound, i.e., could include errors or might even be constructed from false claims. If that were the case, it would not be fit for the development of correct systems. We thus encoded FOCUS and our extension for event-driven processing formally into a theorem prover. The formalizations are built on well-established formalizations of the Higher Order Logic of Computable Functions (HOLCF) [71, 72]. The implementation is a conservative extension without any gaps, meaning no axioms were introduced and all theorems are successfully proven based on the HOLCF. This proves, that FOCUS and the extensions for event-driven processing are sound.

To assure we **encoded FOCUS accurately**, we encoded and verified key theorems from literature [33] formally. We call a FOCUS function (e.g., the composition operator) sufficiently accurately encoded, if the theorem prover accepts the proof of key properties over it (such as commutativity). The encoded theorems also provide valuable abstraction layers for foundational FOCUS definitions. For example, the theorem for commutativity of composition allows the re-ordering of (sub-)systems, without unfolding their definitions. These abstractions allow for more effective and efficient proofs, increasing automation. For instance, the mentioned commutativity enables the re-use of proofs for systems composed of the same parts in a different order. One of the key theorems proven for our encoding of FOCUS is the **compositionality of refinement**. Both the step-wise decomposition of HLRs into an architecture and final composition of all LLRs (Fig. 3) into a coherent system requires the guarantee, that no incorrect behavior is introduced in the process, i.e., a refinement relation holds. This can be verified fully automatically. Additionally, **refinement in FOCUS is transitive**. An evolving system specification might be continuously refined. By transitivity, the final specification is a refinement of the original specification. All theorems were encoded and verified in the theorem prover Isabelle [59].

4.2 The MontiBelle approach

We demonstrate the applicability of the MontiBelle approach by virtue of modeling both a typical avionics system, as well as a liveness requirement. This demonstrates the modeling power, i.e., the ability to accurately represent typical avionics systems and their requirements. The generalizability of this approach, specifically the language-agnostic transformation backend, was recently demonstrated by implementing a language-specific frontend for an ADL [60]. This article shows the use of a different ADL, namely SysML v2. This demonstrates the generalizability to a class of modeling languages.

First, we develop a formal HLR from SR 2 using the textual notation of SysML v2. We thereby express a highly under-specified behavior specification. The result is the system model in listing 1. The syntax is described in [58]. We refer to this first layer of HLRs as HLR 1 and call this specification style a black box specification.

```
part def DLUF_black {
 port input: ~Packets[4]; port output: Packets[4];
 satisfy requirement 'non-starvation' {
  assumes 'infinitely long timeframe' { ∀i∈{1,2,3,4}.
   input[i].length() = ∞ }
  assumes 'message in each interval' { ∀i ∈ {1,2,3,4},
   ∀t:nat: input[i].atTime(t).length() > 0 }
  assumes 'size below max. capacity' { ∀i ∈ {1,2,3,4}:
   ∀v ∈ input[i].values(): v < maxCap[i] }
  require 'infinitely many outputs' { ∀i ∈ {1,2,3,4}:
   output[i].messages().length() = ∞ } } }
```

Listing 1 HLR 1 "non-starvation" formally modeled in textual SysML v2

Next, the development engineer can either directly provide an LLR or decompose the HLR further. Decomposition is motivated by the complexity of the specification. The creation of a compliant LLR and the formal verification of its compliance is hard, if the HLR is complex and multi-facetted. Note, that creating such an LLR without further decomposition might be possible, but challenging. Additionally, the direct development of an LLR is not parallelizable. Due to the complexity of HLR 1, we choose to decompose DLUF's HLR into four scheduler subsystems. The decomposition is summarized in Fig. 3, top left corner. Each scheduler is specified using SysML v2's textual notation, similar to listing 1. The composition of schedulers is referred to as HLR 2. A refinement relation establishes a verifiable trace between HLR 1 and 2. Each scheduler is further decomposed into message buffers and capacity gates. Each of these in total eight subsystems is specified analogously to HLR 1 as a black box specification. A refinement relation links each buffer capacity subsystem to the black box scheduler. The result is an architecture composed of HLRs.

Once the development of LLRs from decomposed black box specifications is reasonably achievable and verifiable, the LLR of all four buffers and capacity gates are specified. LLRs are specified using SysML v2 state machines. Black box buffer and capacity gate specifications are traced to the developed state machines using refinement relations. The syntax for state machines was given in [57, 58]. It is important to note, that state machines are consistent [65]. This means, there exists a function, and thus an implementation, that satisfies the requirements of the state machine. By refining to a state machine, the consistency of the black box specifications and their composition is verified, as well. The refinement chain also ensures the LLRs and architecture to be correct w.r.t. HLR 1.

4.3 Verifiability

In this section, the applicability of the engineering process methodology in combination with our Focus encoding in Isabelle is evaluated by verifying properties and refinement relations between different development artifacts in Isabelle. To ensure, that even low priority messages are transmitted again and again, we verify the non-starvation property (listing 1) formally for the DLUF system. Formal refinement and refactoring techniques from [73–75] are leveraged to achieve higher automation for the verification. Leveraging the compositionality and transitivity of the refinement relation of Sect. 4.1, there is two main proof obligations for the compliance. First, the composition of the four HLR Schedulers refines the DLUF HLR (see Fig. 3). Second, the composition of an HLR buffer and HLR capacity component refines an HLR scheduler component.

1. The DLUF decomposition into four Schedulers
The refinement relation between the *DLUF_HLR2* and *DLUF_HLR1* architectures is proven:

```
theorem shows "(Scheduler_HLR2 400 ⊗ Scheduler_HLR2 300
    ⊗ Scheduler_HLR2 200 ⊗ Scheduler_HLR2 100) ⊆ DLUF_HLR1"
```

Listing 2 Refinement proof between *Scheduler* composition and DLUF. The maximal capacity of a Scheduler is given by a parameter in KByte. Here, the Papameters are 400, 300, 200, and 100.

Proof sketch. The HLR of the Schedulers logically imply the HLR of DLUF. Isabelle's automatic prover tools find a proof.

2. The Scheduler Decomposition into Buffer and Capacity
The composition of the buffer and capacity HLR components refines the scheduler components of the *DLUF_HLR2* architecture:

```
theorem shows "(Buffer_HLR3 ⊗ (Capacity_HLR3 cap)) ⊆
        (Scheduler_HLR2 cap)"
```

Listing 3 Refinement proof between *Scheduler* composition and DLUF.

The proof functions and is automated analogously to the proof of listing 2. However, compliance to the DLUF black box requirement is not enough. Additionally, two more proof obligations are necessary, to confirm the consistency of DLUF and the compliance of the *DLUF_LLR* system. First, the LLR buffer component must refine the HLR buffer component. Second, the LLR capacity component must refine the HLR capacity component.

LLR of the Buffer Component: The buffer HLRs must be fulfilled by the Buffer LLR, i.e., the buffer automaton. To satisfy the first requirement, the buffer automaton's *output shall contain packets only, that were obtained as input*. This way, the buffer is restricted from creating packets, that never existed, e.g., packets, that are greater than the capacity limit and prevent the DLUF system from transmitting data. To realize the second requirement, the buffer *shall send output messages, when input messages exist and the* capacity *feedback provides correct acknowledgments for transmitted messages*. Since the buffer component has two input channels, the refinement relation is shown for every possible sequencing of input messages (see listing 4).

```
theorem shows "Buffer_LLR ⊆ Buffer_HLR3"
```

Listing 4 The LLR buffer component refines the HLR buffer component.

Proof sketch. Proof by induction over the input stream of the buffer automaton. For the empty input stream, both requirements hold trivially. Assume the property holds for an arbitrary stream s. Show, that it holds for every expanded stream of s by processing the additional stream element by the transition function of the buffer automaton. Since the transitions do not contradict the requirements, e.g., the transitions only output messages, that were obtained as input messages, the induction is proven and the refinement relation holds.

LLR of the Capacity Component: For the capacity component, analogous requirements were proven by induction. Compared to the buffer refinement, an additional challenge is, that the refinement is proven for every possible maximum capacity, e.g., for every parameter *cap*. Additionally, it must be ensured, that, *when transmitting a message, the correct acknowledgment is produced* for the buffer component. Secondly, *when the input messages exist and do not exceed the capacity limit, they are transmitted*. As a result, a refinement relation is concluded (see listing 5).

```
theorem shows "Capacity_LLR cap ⊆ Capacity_HLR3 cap"
```

Listing 5 The LLR capacity component refines the HLR capacity component.

LLR of the DLUF System: The final proven refinement relation shows, that the composition of the LLR components fulfills the DLUF requirement, e.g., SR 2.

```
theorem shows
  "(Buffer_LLR ⊗ (Capacity_LLR 400)) ⊗
   (Buffer_LLR ⊗ (Capacity_LLR 300)) ⊗
   (Buffer_LLR ⊗ (Capacity_LLR 200)) ⊗
   (Buffer_LLR ⊗ (Capacity_LLR 100)) ⊆ DLUF_HLR1"
```

Listing 6 The LLR architecture fulfills the non-starvation property and refines DLUF.

Proof Sketch. First, we know, that the *DLUF_LLR* architecture is a refinement of the *DLUF_HLR3* architecture from the compositionality of the refinement relation [59] and previous proofs over Buffer and Capacity HLR refinements (listing 5 and listing 4). Using the same compositionality argument and the scheduler refinement (listing 3), the refinement relation between the *DLUF_HLR3* architecture and the *DLUF_HLR2* architecture follows. At last, using the transitivity of the refinement relation [59] and the refinement between the *DLUF_HLR2* and *DLUF_HLR1* architecture (listing 2), the theorem holds. In conclusion, the non-starvation property for the DLUF system holds.

4.4 Tool qualification

The objective of the presented approach is to replace some of the testing effort by formal verification. This in turn requires the presented tool to be qualified according to RTCA DO-330/EUROCAE ED-215 1. Reference [76] describes, how Isabelle as a tool for proofs for functional correctness of DLUF can be qualified according to EUROCAE ED-215. Isabelle is based on a very small and trusted kernel of peer-reviewed axioms. The definitional approach used for the conservative extensions of Isabelle HOLCF [72] in Sect. 3 and for generated DLUF theories is checked by and derived from this kernel. Thus, no inconsistencies are introduced [46]. NB, that tool qualification according to EUROCAE ED-215 of the generator mapping SysML v2 models to Isabelle theories is needed, as well.

EUROCAE ED-12C requires test coverage analysis to take into account (1) requirements-based coverage analysis[8] and (2) structural coverage analysis.[9] To achieve similar coverage by formal methods, the following objectives need to be met: For (1), the full coverage of HLRs and LLRs is needed. This can be achieved by the presented approach, since it ensures the traceability between HLRs and LLRs (Sect. 3.5). Missing requirements can thus be detected by identifying broken traceability links. For (2), the verification coverage of the software structure is required, which can be achieved through the following means: (i) The complete coverage of each single requirement can be checked by the traceability provided by the approach (Sect. 3.5). (ii) The completeness of the system's requirements can be achieved by using Focus, which offers the semantic foundation created by mathematical and logical means. Unintended dataflow relationships are avoided by means of code generation (Sect. 3.6). (iii) Extraneous and deactivated code can be achieved by review or not formal analysis. Since in our case the code is generated (Sect. 3.6), no unnecessary code is introduced. This property can be demonstrated by the corresponding qualification.

5 Conclusion

The article raised the need for means for detecting issues early in avionics development processes. These are assumed to reduce costs for fixing defects considerably. At the same time model-based development can increase the software

[8] The coverage ensures that there is verification evidence available for the complete set of the system's requirements.

[9] Since exhaustive testing is usually not achievable, adequate metrics assess the degree, to which testing provides good enough confidence for product safety.

quality, if the created models meet the necessary quality. Furthermore, formal methods can reduce the test effort and in particular prove the correctness of software, while testing can demonstrate only the absence of defects. As a result of all of these improvements, the development can be performed at the same pace during all stages.

We described, which formal methods can be deployed in line with EUROCAE ED-216 and explained relevant and related work in this context. As a foundation for this article we have selected theorem provers as a representative of deductive methods, because they are the most powerful and most expressive formal method tool. However, they require a strong user expertise and continuous interaction. Thus, we have developed a corresponding methodology easing their successful application in industry projects.

A relevant avionics use case is presented to demonstrate the viability of the methods and methodology. The SRs for the DLUF system are listed and one key requirement, a liveness property, is identified for detailed treatment. This property cannot be exhaustively tested and thus requires formal methods. While the case study is small, it is also archetypal and one representative requirement is specifically formally verified.

The methodology demonstrated in this article consists of a model-driven verification framework enabling event-driven system specifications and reasoning. It enables a verified design and a correct refinement of safety-critical systems. The designer can either directly specify the system using a logic language such as Isabelle, or using an architecture description language such as SysML as a user-friendly way for describing the interface, behavior, and interaction between components. The system model and any desired properties can then be translated to equivalent specifications in a theorem prover. FOCUS as semantical foundation was chosen due to its compositionality of refinements.

We have introduced means for modeling time-critical software systems efficiently and effectively and for verifying properties formally. This includes the FOCUS data types for monolithic definitions of timed components and systems and their encodings in Isabelle. These data types are *stream*, *sb*, *spf*, and *sps*. Furthermore, causality concepts and the semantics of components are formalized. Event-driven processing components are introduced. To describe such components, a merge specification, which describes possible processing orders, and event automata, that define the event-driven behavior of the component, are formalized and encoded in Isabelle. Next, a composition operator capable of parallel, sequential, and feedback composition was introduced and a corresponding Isabelle encoding has been provided. The approach offers decompositional specification of systems using (1) declarative specifications, (2) architecture, and (3) imperative specifications. The refinement relations ensure traceability and enable the automated formal verification of compliance, compatibility, and consistency between an HLR and its corresponding LLRs. The developed methodology is complemented by a tool chain comprising a SysML v2 profile for modeling, a code generator for automatic theorem encoding, and an F-IDE. The modeling language profile supports system development along methodological recommendations. The code-generator drives automation, while the F-IDE enables the intuitive and integrated use of the tool chain.

The methodology and tool chain are applied to the verification of the liveness property of the DLUF case study. To this end, the generated Isabelle theories for the SysML models of the DLUF system are formally checked regarding their compliance and consistency by proving corresponding refinement relations in Isabelle. This demonstrates applicability of the MontiBelle approach.

As introduced in Sect. 1, the substitution of certain tests and manual reviews related to the mentioned objectives is possible. It also helps with requirements demanding properties being true always or never, which generally cannot be fully verified by testing. Note however, that certain sets of tests and reviews can only be complemented by this approach, but not completely replaced. In this context the following aspects are relevant:

- The correctness of formalization of the requirements need to be checked.
- Justification and appropriateness of the methodology needs to be checked.
- Compatibility with the target computer needs to be checked (unless the target environment is formally modeled).
- Completeness of requirements needs to be checked.
- Identifying dead or disabled code is covered by established tools.

In general, we observe an increasing maturity and feasibility in the application of formal methods in safety-critical systems, as it is possible by following the EUROCAE ED-216 standard, which can help to replace or complement many tests. NB, that the formal specification might create some additional effort, when considering the overall benefits over testing. However, they usually overcompensate later significantly, since technical flaws at the beginning may result in highly expensive corrections of deficits identified later in the development process, and the later the errors are corrected, the more costly they are to correct. In a future case-study, a more precise evaluation of costs might be performed alongside the formal correctness verification using the presented approach.

To counter the claim of increased effort required for early application of formal methods, we proposed a language-agnostic code generator and presented an industry standard modeling language as front end in this article. As industry proven and approved tools for systems engineering and especially for system modeling exist, we deem it feasible and maybe even necessary to **integrate** the MontiBelle framework into such tools and demonstrate the unobtrusive nature of additional checks, reports, and safety guarantees emerging from formal verification. We also strongly believe in the benefits provided by a unified yet customizable industry standard modeling language, such as the SysML v2 is aiming at to become. We belief, that all users, be it requirements stakeholders, system engineers, or quality control, could greatly benefit from accessible, integrated, and transparent application of formal methods.

Concerning the **scalability** of the approach, in this case study we dealt with a model consisting of ca. 40 component specifications. Future work needs to verify properties of a much larger industry model. We still expect the verification complexity to be well manageable by leveraging compositionality of refinement, and by following the proposed methodological way of designing the system. If, e.g., we propose a design recommendation, that each component shall be decomposed into up to ten sub-components, then step-wise refinement of properties will involve at most ten components, where this number is supposed to be small enough for the verification to be fully automatic. There exist larger case studies, reaching well above 1000 components. We recommend using decomposition of systems into up to ten components to manage this complexity. This way we could handle magnitudes of up to 1000 components using just three decomposition layers.

The presented approach enables considerable **model quality** by ensuring, e.g., model correctness and consistency. To provide a more holistic analysis of the effects of the MontiBelle approach regarding the model quality, an in-depth analysis including multiple case studies is necessary. Such an analysis will then be able to demonstrate, which particular model quality attributes [12] are or might not be covered. For model quality attributes, that might not yet be covered, the approach might be extended or adapted accordingly.

Funding Open Access funding enabled and organized by Projekt DEAL. German Federal Ministry for Economic Affairs and Climate Action, AMoBaCoD-Project (Grant no. 20X2201C).

Data availability Not applicable.

Declarations

Conflict of interest The authors have no conflict of interest to declare.

References

1. Annighoefer, B., Halle, M., Schweiger, A., Reich, M., Watkins, C., van der Leest, S., Harwarth, S., Deiber, P.: Challenges and ways forward for avionics platforms and their development in 2019. In: 38th Digital Avionics System Conference (DASC), San Diego, California, USA (2019)
2. Brahmi, A., Delmas, D., Essoussi, M.H., Randimbivololona, F., Atki, A., Marie, T.: Formalise to automate: deployment of a safe and cost-efficient process for avionics software. In: 9th European Congress on Embedded Real Time Software and Systems (ERTS 2018), Toulouse, France (2018)
3. Baziuk, W.: Bnr/nortel: path to improve product quality, reliability and customer satisfaction. In: Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95, pp. 256–262, Toulouse, France (1995)
4. Boehm, B.W.: Software engineering. IEEE Trans. Comput. C **25**(12), 1226–1241 (1976)
5. Cockburn, A.: Agile Software Development: The Cooperative Game, 2nd edn. Addison-Wesley Professional, Upper Saddle River (2006)
6. Rumpe, B.: Agile Modeling with UML: Code Generation, Testing, Refactoring. Springer, Berlin (2017)
7. Marsden, J., Windisch, A., Mayo, R., Grossi, J., Villermin, J., Fabre, L., Aventini, C.: ED-12C/DO-178C vs. Agile Manifesto—a solution to agile development of certifiable avionics systems. In: 9th European Congress Embedded Real-time Software and Systems (ERTS2 2018), Toulouse, France (2018)
8. Beizer, B.: Software System Testing and Quality Assurance. Van Nostrand Reinhold Co., New York (1984)
9. Rivers, A.T., Vouk, M.A.: Resource-constrained non-operational testing of software. In: Proceedings Ninth International Symposium on Software Reliability Engineering (Cat. No.98TB100257), pp. 154–163 (1998)
10. Moy, Y., Ledinot, E., Delseny, H., Wiels, V., Monate, B.: Testing or formal verification: Do-178c alternatives and industrial experience. IEEE Softw. **30**(3), 50–57 (2013)
11. Schöpp, U., Schweiger, A., Reich, M., Chuprina, T., Lúcio, L., Brüning, H.: Requirements-based code model checking. In: 2020 IEEE Workshop on Formal Requirements (FORMREQ), pp. 21–27. IEEE Computer Society, Los Alamitos (2020)
12. Fieber, F., Huhn, M., Rumpe, B.: Modellqualität als Indikator für Softwarequalität: eine Taxonomie. Inform. Spektrum **31**(5), 408–424 (2008)
13. Kausch, H., Pfeiffer, M., Raco, D., Rumpe, B., Schweiger, A.: Enhancing system-model quality: evaluation of the MontiBelle approach with the avionics case study on a data link uplink feed system. In: Software Engineering 2024—Companion Proceedings (AvioSE), pp. 119–138. Gesellschaft für Informatik e.V., Linz (2024)
14. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming

Languages. POPL '77, pp. 238–252. Association for Computing Machinery, New York (1977)
15. Ascari, F., Bruni, R., Gori, R.: Limits and difficulties in the design of under-approximation abstract domains. In: Foundations of Software Science and Computation Structures, Editors: Patricia Bouyer, Lutz Schröder, pp. 21–39. Springer, Cham (2022)
16. O'Hearn, P.W.: Incorrectness logic. Proc. ACM Program. Lang. 4(POPL) (2019)
17. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Grumberg, O., Veith H. (Eds.) Logics of Programs, pp. 52–71. Springer, Berlin (1982)
18. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in cesar. In: International Symposium on Programming, pp. 337–351. Springer, Berlin (1982)
19. Maoz, S., Ringert, J.O., Rumpe, B.: CDDiff: Semantic Differencing for Class Diagrams. In: ECOOP 2011—Object-Oriented Programming, pp. 230–254. Springer, UK (2011)
20. Baier, C., Katoen, J.-P.: Principles of Model Checking. The MIT Press, Cambridge (2008)
21. Esparza, J., Heljanko, K.: Unfoldings—A Partial-Order Approach to Model Checking. Springer, Berlin (2008)
22. Klein, G., Andronick, J., Elphinstone, K., Murray, T., Sewell, T., Kolanski, R., Heiser, G.: Comprehensive formal verification of an os microkernel. ACM Trans. Comput. Syst. **32**(1), 1–70 (2014)
23. Kausch, H., Pfeiffer, M., Raco, D., Rumpe, B.: Model-based design of correct safety-critical systems using dataflow languages on the example of SysML architecture and behavior diagrams. In: Proceedings of the Software Engineering 2021 Satellite Events, vol. 2814. CEUR, Online (2021)
24. Wiedijk, F.: Comparing mathematical provers. In: Asperti, A., Buchberger, B., Davenport J.H. (Eds.) Mathematical Knowledge Management, pp. 188–202. Springer, Berlin (2003)
25. Harel, D., Rumpe, B.: Meaningful modeling: what's the semantics of "Semantics"? IEEE Comput. J. **37**(10), 64–72 (2004)
26. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall International, Englewood Cliffs (1985)
27. Murray, T., Lowe, G.: On refinement-closed security properties and nondeterministic compositions. Electr. Notes Theor. Comput. Sci. **250**, 49–68 (2009)
28. Milner, R.: A Calculus of Communicating Systems. Springer, Berlin (1982)
29. Parrow, J.: An introduction to the pi-calculus. In: Bergstra, J.A., Ponse A., Smolka, S.A. (Eds.) Handbook of Process Algebra, pp. 479–543. Elsevier Science, Amsterdam (2001)
30. Lee, E.: Fundamental limits of cyber-physical systems modeling. ACM Trans. Cyber Phys. Syst. **1**, 1–26 (2016)
31. Abadi, M., Lamport, L.: Open Systems in TLA. In: Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing-PODC '94, pp. 81–90. ACM Press, New York (1994)
32. Reisig, W.: Petri Nets: An Introduction. Springer, Berlin (1985)
33. Broy, M., Stølen, K.: Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement. Springer, New York (2001)
34. Broy, M., Rumpe, B.: Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. Inform. Spektrum **30**(1), 3–18 (2007)
35. Carnab, R.: Meaning and Necessity: A Study in Semantics and Modal Logic. The University of Chicago Press, Chicago (1947)
36. Kausch, H., Pfeiffer, M., Raco, D., Rumpe, B.: MontiBelle—toolbox for a model-based development and verification of distributed critical systems for compliance with functional safety. In: AIAA Scitech 2020 Forum. AIAA, Orlando (2020)
37. Grosu, R., Rumpe, B.: Concurrent timed port automata. Technical Report TUM-I9533, TU Munich, Germany (1995)
38. He, J., Turner, K.J.: In: Specification and Verification of Synchronous Hardware using LOTOS, pp. 295–312. Springer, Boston (1999)
39. Kounev, S., Rathfelder, C., Klatt, B.: 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, Tallinn, Estonia (FESCA)
40. Berry, G., Bouali, A., Fornari, X., Ledinot, E., Nassor, E., de Simone, R.: Esterel: a formal method applied to avionic software development. Sci. Comput. Program. **36**(1), 5–25 (2000)
41. Caspi, P., Pilaud, D., Halbwachs, N., Plaice, J.: Lustre: a declarative language for programming synchronous systems. In: POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, Munich, West Germany (1987)
42. Becker, S., Koziolek, H., Reussner, R.: The Palladio component model for model-driven performance prediction. J. Syst. Softw. **82**, 3–22 (2009)
43. Dziwok, S., Pohlmann, U., Piskachev, G., Schubert, D., Thiele, S., Gerking, C.: 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. Version 1.0 (2016)
44. Voss, S., Zverlov, S.: Design Space Exploration in Auto FOCUS 3—an Overview. In: IFIP First International Workshop on Design Space Exploration of Cyber-Physical Systems, Berlin, Germany (2014)
45. Object Management Group (OMG), SysML v2 Submission Team (SST): OMG Systems Modeling Language (SysML) Version 2.0 Beta 2 (Release 2024-03). https://github.com/Systems-Modeling. Accessed 28 Sep 2024
46. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic. lecture notes in artificial intelligence, vol. 2283. Springer, Berlin (2002)
47. Paul, S., Agha, G., Patterson, S., Varela, C.: Eventual consensus in Synod: verification using a failure-aware actor model. Innov. Syst. Softw. Eng. **19**(4), 395–410 (2023)
48. Rayadurgam, S., Joshi, A., Heimdahl, M.P.E.: Using PVS to prove properties of systems modelled in a synchronous dataflow language. In: Formal Methods and Software Engineering, pp. 167–186. Springer, Berlin (2003)
49. Apvrille, L., Roudier, Y.: SysML-Sec: a SysML environment for the design and development of secure embedded systems. In: APCOSEC 2013, Yokohama (2013)
50. Cengarle, M.V., Grönniger, H., Rumpe, B.: Variability within modeling language definitions. In: Conference on Model Driven Engineering Languages and Systems (MODELS'09). LNCS 5795, pp. 670–684. Springer, USA (2009)
51. Grönniger, H., Rumpe, B.: Modeling Language Variability. In: Workshop on modeling, development and verification of adaptive systems. LNCS 6662, pp. 17–32. Springer, USA (2011)
52. Kriebel, S., Raco, D., Rumpe, B., Stüber, S.: Model-based engineering for avionics: will specification and formal verification e.g. Based on Broy's Streams Become Feasible? In: Proceedings of the workshops of the software engineering conference. Workshop on Avionics Systems and Software Engineering (AvioSE'19). CEUR Workshop Proceedings, vol. 2308, pp. 87–94. CEUR Workshop Proceedings, Online (2019)
53. Kausch, H., Pfeiffer, M., Raco, D., Rumpe, B.: An approach for logic-based knowledge representation and automated reasoning over Underspecification and refinement in safety-critical cyber-physical systems. In: Combined Proceedings of the Workshops at

Software Engineering 2020, vol. 2581. CEUR Workshop Proceedings, Online (2020)

54. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: MontiCore 1.0: Ein Framework zur Erstellung und Verarbeitung domänspezifischer Sprachen. Informatik-Bericht 2006-04, CFG-Fakultät, TU Braunschweig (2006)
55. Hölldobler, K., Rumpe, B.: MontiCore 5 Language Workbench Edition 2017. Aachener Informatik-Berichte, Software Engineering, Band 32. Software Engineering. Shaker Verlag, Germany (2017)
56. Hölldobler, K., Kautz, O., Rumpe, B.: MontiCore language workbench and library handbook: edition 2021. Aachener Informatik-Berichte, Software Engineering, Band 48. Software Engineering. Shaker Verlag, Düren (2021)
57. Kausch, H., Pfeiffer, M., Raco, D., Rumpe, B., Schweiger, A.: Correct and sustainable development using model-based engineering and formal methods. In: 2022 IEEE/AIAA 41st Digital Avionics Systems Conference (DASC). IEEE, USA (2022)
58. Kausch, H., Michael, J., Pfeiffer, M., Raco, D., Rumpe, B., Schweiger, A.: Model-based development and logical AI for secure and safe avionics systems: a verification framework for SysML behavior specifications. In: Aerospace Europe Conference 2021 (AEC 2021). Council of European Aerospace Societies (CEAS), Warsaw, Poland (2021)
59. Bürger, J.C., Kausch, H., Raco, D., Ringert, J.O., Rumpe, B., Stüber, S., Wiartalla, M.: Towards an Isabelle Theory for Distributed, Interactive Systems—The Untimed Case. Aachener Informatik Berichte, Software Engineering, Band 45. Software Engineering. Shaker Verlag, Germany (2020)
60. Kausch, H., Pfeiffer, M., Raco, D., Rath, A., Rumpe, B., Schweiger, A.: 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, pp. 169–188. Gesellschaft für Informatik e.V., Bonn (2023)
61. Reich, M., Schweiger, A., Lorenz, J., Margull, U.: Experience Report on Reuse in Avionics. In: IBS Workshop Micro Air Vehicle Technologie—Konzepte und Anwendungen 2019, vol. 9, pp. 28–35. TUDpress THELEM Universitätsverlag GmbH und Co. KG, Dresden (2020)
62. Böhm, W., Broy, M., Klein, C., Pohl, K., Rumpe, B., Schröck, S. (eds.): Model-Based Engineering of Collaborative Embedded Systems. Springer, Cham (2021)
63. Ringert, J.O., Rumpe, B.: A little synopsis on streams, stream processing functions, and state-based stream processing. Int. J. Softw. Inform. **5**, 29–53 (2011)
64. Alpern, B., Schneider, F.B.: Defining liveness. Inf. Process. Lett. **21**(4), 181–185 (1985)
65. Rumpe, B.: Formale Methodik des Entwurfs verteilter objektorientierter Systeme. In: Ausgezeichnete Informatikdissertationen 1997. B. G. Teubner, Stuttgart (1997)
66. Rumpe, B., Klein, C., Broy, M.: Ein strombasiertes mathematisches Modell verteilter informationsverarbeitender Systeme—Syslab-Systemmodell. Technischer Bericht TUM-I9510, TU München, Deutschland (1995)
67. Grosu, R., Klein, C., Rumpe, B., Broy, M.: State Transition Diagrams. Technical report, TU Munich (1996)
68. Rumpe, B.: Formale Methodik des Entwurfs Verteilter Objektorientierter Systeme. Herbert Utz Verlag Wissenschaft, Munich (1996)
69. Paech, B., Rumpe, B.: A new concept of refinement used for behaviour modelling with automata. In: Proceedings of the Industrial Benefit of Formal Methods (FME'94). LNCS 873, pp. 154–174. Springer, Spain (1994)
70. Hölldobler, K., Rumpe, B., Wortmann, A.: Software language engineering in the large: towards composing and deriving languages. J. Comput. Lang. Syst. Struct. **54**, 386–405 (2018)
71. Regensburger, F.: HOLCF: Eine konservative Erweiterung von HOL um LCF. PhD thesis, Technische Universität München, Munich, Germany (1994)
72. Huffman, B.C.: HOLCF '11: A Definitional Domain Theory for Verifying Functional Programs. Portland State University, Portland (2012)
73. Philipps, J., Rumpe, B.: Refinement of information flow architectures. In: ICFEM'97 Proceedings. IEEE CS Press, Hiroshima, Japan (1997)
74. Philipps, J., Rumpe, B.: Roots of refactoring. In: Tenth OOPSLA Workshop on Behavioral Semantics. Tampa Bay, Florida, USA, October 15. Northeastern University, USA (2001)
75. Andronick, J.: Please check my 500K LOC of Isabelle. In: Cofer, D., Klein, G., Slind, K., Wiels V. (Eds.) Qualification of Formal Methods Tools—Report from Dagstuhl Seminar 15182 (2015)
76. Andronick, J.: Please check my 500K LOC of Isabelle. In: Qualification of Formal Methods Tools—Report from Dagstuhl Seminar 15182 (2015)