AerospaceEurope Conference2021
WARSAW, POLAND | 23 -26 NOVEMBER 2021
CEAS Council of European Aerospace Societies

MODEL-BASED DEVELOPMENT AND LOGICAL AI FOR SECURE AND SAFE AVIONICS SYSTEMS: A VERIFICATION FRAMEWORK FOR SYSML BEHAVIOR SPECIFICATIONS

Hendrik Kausch[1], Judith Michael[1], Mathias Pfeiffer[1], Deni Raco[1], Bernhard Rumpe[1] & Andreas Schweiger[2]

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

Abstract

In this paper a model-based verification approach for detecting potential safety or security vulnerabilities in cyber-physical systems is presented. It addresses the question of how model-driven specifications (in particular state-based style) combined with code generation and logic-based AI can be used for assuring safety and security early in design. Currently, the common verification method in industry still remains testing and reviews, but their costs grow overproportionally with the system size and they cannot achieve exhaustive coverage. To overcome this, an extension of a model-based verification framework for safety-critical cyber-physical systems is presented. A SysML profile is extended for supporting event-driven state-based specifications, including corresponding encodings of the key structures in the theorem prover Isabelle and a code generator from SysML to Isabelle. An evaluation on an avionics case study indicates that model-based approaches and logic-based AI can be applied for lowering certification costs.

Keywords: logical AI, model-driven development, security, safety, avionics

1. Introduction

Currently, avionics accounts for over 30% of the aircraft development costs [1]. About 75% (of these 30%) goes into verification [4]. Safety considerations during design are necessary according to the required certification, which is approved by EASA and FAA. As baselines for this, standards such as RTCA [26] DO-178C or RTCA DO-254 are used during the development process.

But designing mainly with safety and not so much security in mind leads to one of the key system design issues of nowadays. It is important to recognize that security and safety are strongly connected and have to be carefully considered in advance. This means that not only safety, but also security needs to be designed into avionics to demonstrate not just the initial airworthiness, but also the maintenance of continuous airworthiness and protection from unauthorized interaction.

Organizations such as RTCA, the company ARNIC [2], or SAE International are responsible for creating and maintaining standards dealing with some of the most relevant aspects of our time such as Cyber Physical Systems Security [30] or Artificial Intelligence in Aviation [29] etc. Seven levels of security trustworthiness (Evaluation Assurance Level) [10] are used for system classification with respect to criticality. A high criticality rises the demands on the depth to which the manufacturer must describe and test his product. The highest level (so called level 7/7+) requires formal correctness arguments, which is used in extremely high risk situations, in particular when the high value of the goods justifies the higher costs.

Companies such as Airbus have successfully adapted formal methods for verifying properties on the code level, using model-checkers and abstract interpreters e.g. for worst-case execution time

analysis [22, 31], which is also regulated by the standard DO-333 formal methods supplement of DO-178C. But when refining requirements from the system-level to higher-level abstract requirements and then further to lower-level requirements, the most used technique continues to be testing and reviews. Their costs grow overproportionally with the system size [4] though, and besides tests can show the presence but not the absence of errors. So the question we will address is how can model-based specifications (in particular a state-based style) and logic-based AI be leveraged to improve safety and security of safety-critical systems.

Formal methods tackle some deficits mentioned about testing: high costs and exhaustive coverage. Theorem proving, as the formal method offering the highest assurance, has been successfully used to completely verify security and safety properties, such es the prominent example of exhaustively verifying the entire kernel of an operating system [19], or other security-critical components [23]. One can verify that a certain claim holds in the model or use the counterexample finder to generate an attack scenario against the network that shows how the attacker penetrates the system. By constructing a threat model, security-critical honest components are composed with maximally hostile untrusted environmental intruder-components in their most unrefined variant. Any refinement of an untrusted component of the threat model represents the behaviour of a real component with which the security-critical components may be composed in practice. Formalisms such as CSP (as was used in [23]), CSS [21], pi-calculus [25], or FOCUS [6, 5] greatly assist this process, since they offer the concepts of nondeterminism and underspecification, a notion of behavioral refinement, time-sensitive specifications and hierarchical decomposition. In FOCUS, distributed systems consist of components exchanging messages through unidirectional channels. The semantics of a component is a (set of) stream processing functions. The most important reason that FOCUS is used in this paper is due to the compatibility of refinement with composition. This means that when component B refines component A (all behaviours of B are also behaviours of A), a system built by placing B in a particular context (environment) will always refine the system built by placing A in that same context, for all possible contexts. Hence, if it is proven that a security property holds for the composition of a security-critical component with a maxically hostile component, it will also hold for any possible real instantiation of hostile components. Secrecy (certain messages should not occur) or authentication (should occur only under particular circumstances) properties can e.g. thus be checked. In the evaluation chapter we will show for instance how the violation of a property of a component can be automatically checked (the property states that certain messages should never occur in the output). Also, theorem provers have an advantage (compared to the model-checking approach on CSP models in [23]) in particular when verifying software, since, despite some progress by so-called partial-order approaches [11], model-checkers run into the well-known state-space-explosion, whereas proof complexity using theorem provers grows only linearly with system complexity [18].

The contribution in this paper updates our previous works [27, 18, 17, 8, 16, 20], where the model-based verification of several safety-critical properties can be found. In our latest work [18], we used a time-synchronous behavior specification paradigm [12, 17], known to be well-suited for hardware specification and verification [14], where model-checking techniques, unlike in software, can usually achieve an excellent exhaustive coverage. Meanwhile, in software, an event-driven paradigm is much more common. Hence, in this paper, event-based automata are introduced for capturing non-determinism and underspecification.

We thus extend our work by the following key novel contributions:

- SysML-Profile [32] for event-automata
- Event-automata encoding in the theorem prover Isabelle [24]
- Extending the code generator from SysML to Isabelle to support event-automata
- Evaluation on an avionics case study, by showing how properties of components specified by event-automata can be checked.

This paper is structured as follows: The second chapter describes the basics of the event-driven methodology and an avionics running example. In chapter three the frontend of the framework consisting in the modeling language and the generator to the theorem prover is described. Then chapter

four presents the encoding in the theorem prover of the basic dataflow structures, and of the new introduced event-driven state machines. Chapter five evaluates the methodology by verifying a property of the frontend model in the theorem prover, thus giving helpful information to the developer about potential safety or security vulnerabilities. Finally, chapter six presents a conclusion of this work.

2. Designing Event-Based Systems: An Introduction to Event-Automata

Specifying a system or a component in an event-driven way is a simple and understandable specification style. It allows reactive description for any single input event instead of needing to specify reactions for any possible combination of input events arriving in a complete time-slice. The pilot flying system (PFS) from Nasa and Rockwell Collins [9] (see fig. 1) is such an event-driven system. It is comprised of redundant guidance systems on each side of the cockpit. The control of the plane can be transferred across by flicking a switch. But the transfer may be subject to a errors such as hardware fault of the communication bus or external disturbances of the guidance systems.
We use event-driven automata to define the behavior of atomic event-driven components. Unlike timed port automata [12, 17] that react to the input on all channels once per time unit, an event-automata reacts to every single input event arriving on any input channel. This matches the intention of event-driven systems perfectly. Using this execution model, the event-automata for a flight controller can then be constructed as shown in fig. 2.

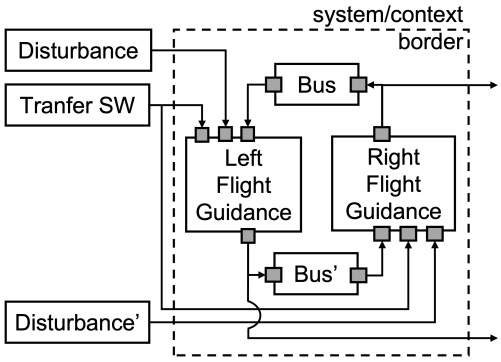

Figure 1 – PFS System

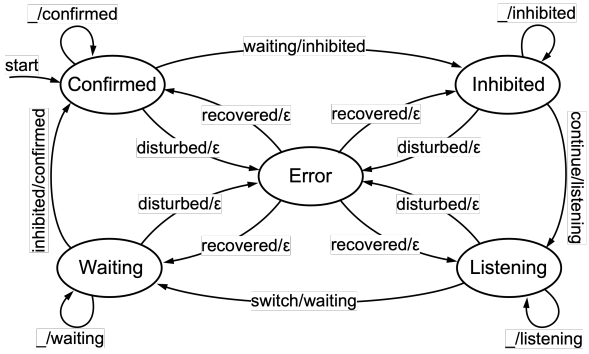

Figure 2 – Left Flight Guidance

Using event-driven automata for component behavior specification leads to a difficulty arising from the component's input interface, namely how does an automaton react, if two events on different channels arrive at the same time. Since any event-driven automata reacts on any single input event is can only react to two events in a serialized fashion. A serializing component for the input interface is necessary, which forwards the input to the automaton. For a component with one input channel this is trivial because the internal order of a history is well defined. For components with two or more input channels like the guidance system the input histories must be merged into one history. To allow such a serialization for bundles of input histories a generalized bundle merge component which merges an arbitrary but finite amount of input histories is necessary. Underspecifying such a bundle merge component allows it to produce any possible merged output stream and thus includes any possible prioritization of channels as a refinement. Then, each event-based component specified by an event-automata consists internally of a bundle-merge component and the event-automata.
To form a complete system, all its components must be composed regarding their channel connections. Using FOCUS [6] as the underlying methodology, a systems composition maintains all its sub-components properties. Especially noteworthy is that refinement is compositional. Thus, a refinement of a sub-component automatically implies the refinement of the complete system [5].

3. Model-driven and Generative Approach: The Frontend

Systems and their requirements are typically formulated as (structured) text. From this, one can derive models in some modeling language (for the purpose of this paper we assume the correctness of this step). Our framework then transforms those models into a knowledge base representation suited

for automated reasoning and verification [18]. Previously, the only state-based behavior specificaton we supported were timed port automata. To enable the use of event-driven behavior, we extended our model-driven approach in three key areas: 1) recognizing event triggers in the modeling language (SysML), 2) transforming into an event-driven intermediary representation (build on [18]), and 3) appropriate encoding in the knowledge base (Isabelle).

3.1 Intermediary Representation & Encoding

Our framework is designed to be modeling language agnostic. Concepts, mapping, and encoding are shared between different languages. To achieve this, model artifacts describing architecture, behavior, and requirements are first transformed into a common intermediary representation [18]. Our representation for timed port automata based behavior consisted of the following: a statespace, a set of intial configurations, and a set of transitions. Event-driven automatas introduce two types of events. Message events represent the receipt of a message on a particular input channel. Time events represent the passing of time. Time passes on all channels equally. These two event types lead to two types of transitions required for event-automata: *messageTransitions* and *timeTransitions*.

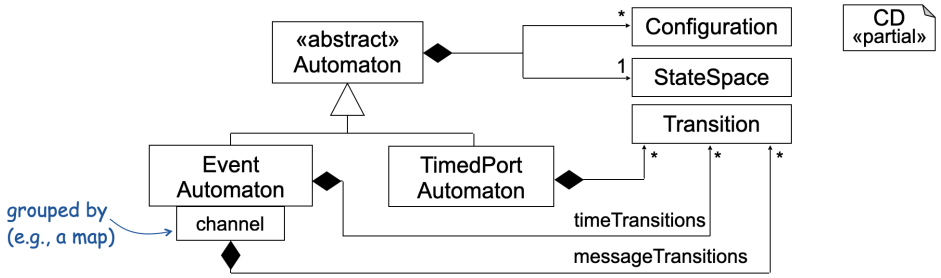

Figure 3 – Main concepts of the intermediary representation for automata

The intermediary representation was extended by an abstract concept *Automaton* that re-uses existing *StateSpace* and *Configuration*s. Both the *TimedPortAutomaton* and *EventAutomaton* are concrete instances of this abstract concept. The *TimedPortAutomaton* employs a simple list of *Transition*s [18]. *EventAutomaton*s manages two transition types: *messageTransitions* are triggered by message events on a particular channel and can only process that message. *timeTransitions* are triggered by time passing on all channels and cannot process any message.
The intermediary representation is then encoded into the knowledge base implemented in Isabelle.

3.2 Modeling Language

To model systems and requirements, we use a final draft version of the SysML v2 [32] in accordance with the most recent publications of the SysML Submission Team (SST) [33]. SysML v2 provides so called state definitions for state-based behavior descriptions. State defintions mainly consist of states and transitions. Each transition defines a starting state, an optional guard, an event trigger, an action, and a target state [18]:

```
1 transition first      Transmitting                 /* starting state        */
2            accept     input                        /* message event trigger */
3            do action  { send input to output; }    /* action                */
4            then       Error                        /* target state          */
```

Triggers allow us to specificy the type of the event and on what input channel it occured. A message event on a particular channel is denoted by the channel name. A time event is denoted by –.

4. Semantic domain in Isabelle: The Backend

Formally analyzing a system is only possible by giving it clear semantics [13]. To automate this semantics giving process, FOCUS based structures are implemented in the theorem prover Isabelle [8]. This encoding of a semantic domain acts as a target for the system model transformation process. The following listings show important parts of the Isabelle structures slightly simplified.

The most important datatype is the domain of streams. A stream is a sequence of elements over an alphabet and describe the history of channels. Every stream is either empty or defined as head element and its rest of the stream, similar to lists in Haskell. The keyword *domain* defines the stream datatype in Isabelle and associates it a chain-complete partial order (the prefix order) with a smallest element [15]. This ensures the existence of a least fixed point (the infinite stream is thus approximated by its finite prefixes). This in turn enables us to define a semantic for iterative processing of infinite streams, e.g., event-automata. Infinite streams are included via the well-known concept of *lazy* evaluation.

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

The event datatype is defined as a message or the passage of time modeled by a tick ($\surd$). With the stream and event datatypes it is now possible to define event streams.

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

A component often has more than one input and output channel. A tuple representation of component interfaces hinders the formalization of a composition operator which should work with arbitrary component interfaces. Instead, a function mapping channel labels to streams associates specific histories to components inputs or outputs is a better choice. This is called bundling streams. The function representation is also necessary for defining an input stream merging component for general component interfaces introduced in section 2. But since channels allow only certain messages, a function is not necessarily a wellformed stream bundle. The messages of each channel inside the bundle must be a subset of the allowed messages on that channel.

```
definition wellformed :: "('cs ⇒ M stream) ⇒ bool" where
"wellformed f = ∀channel. messagesOf(f channel) ⊆ allowedOn channel"
```

Using the *wellformed* predicate, the pcpo of bundles is defined using the *pcpodef* keyword. This lifts the prefix order on streams point-wise to bundles and allows a composition operator definition as a fixed point calculation.

```
pcpodef 'channel bundle("(_^Ω")
          = "{f::('channel ⇒ M stream). wellformed f}"
```

Having defined bundles for describing input and output interfaces and histories of components, any deterministic component is a continuous function that maps input bundles to output bundles. These functions are called stream processing functions (SPFs) in the semantic domain.

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

Underspecified components have many possible implementations. It is fittingly defined as a set of SPFs called stream processing specification (SPS). A singleton SPS is equivalent to a SPF and thus deterministic. An empty SPS is called an inconsistent (not implementable) specification.

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

An event-automaton then consists of a state-space, an input message type, an output channel set, a transition function and initial configurations. Sets are used to encode nondeterminism in initial configuration and transition results.

```
  record ('state,'message,'outchannel)
    Automaton =
      transitions :: "'state ⇒ 'message ⇒ (('state × 'outchannel^Ω) set)"
      configurations :: "('state × 'outchannel^Ω) set"
```

We have encoded the bundle merge component in Isabelle. It is defined as a nondeterministic component that maps bundles to event streams. Since event-based automata may react differently on events from different channels the merged output stream additionally transmits its messages channel origin. Each possible bundle-merge component has to fulfill two properties: 1) It must transmit no more and no less than all complete time-slices of the input bundle, and 2) filtering the output stream for messages of a specific channel must result in a prefix of the channels history.

Notation	Signature [1]	Functionality
$\#_{\surd}$	streamTicks:: $M^{\omega} \Rightarrow \mathbb{N}_{\infty}$	Returns number of $\surd$s occuring in a stream
$\#_{\surd}$	minBundleTicks:: $CS^{\Omega} \Rightarrow \mathbb{N}_{\infty}$	Minimum number of $\surd$s occuring in any stream
	map:: $(M_1 \Rightarrow M_2) \Rightarrow M_1^{\omega} \Rightarrow M_2^{\omega}$	Applies function to every event of a stream
	filter:: $(M \Rightarrow \mathbb{B}) \Rightarrow M^{\omega} \Rightarrow M^{\omega}$	Filters events of a stream according to a predicate
$\sqsubseteq$	prefixOrder:: $M^{\omega} \Rightarrow M^{\omega} \Rightarrow \mathbb{B}$	Checks if first stream is prefix of second stream
	messagesOf:: $M^{\omega} \Rightarrow \mathscr{P}(M)$ **or** $CS^{\Omega} \Rightarrow \mathscr{P}(M)$	Set containing all occurring messages
.	bundleGetCh:: $CS^{\Omega} \Rightarrow CS \Rightarrow M^{\omega}$	Returns stream from a specific channel of a bundle

[1]: $\mathbb{N}_{\infty}$ = Natural numbers inclusive infinity, M^{ω} = event stream over messages M, CS^{Ω} = bundle over channel set CS

Table 1 – Functions implemented in Isabelle

```
definition bundleMerge::"('cs^Ω ⇒('cs × M) event stream) set" where
"bundleMerge ≡ {f | f. ∀bundle. #√bundle = #√(f bundle) ∧
      (∀channel. map snd (filter (λ(c,_). c=channel)(f bundle) ⊑ bundle.c}"
```

Each event-automaton can be mapped to an SPS by effectively applying fixed point iterations over the transition function [28]. The event-automata and bundle merge implementations in Isabelle define the semantics of any component realized by an event-driven automata in SysML. The next step is to evaluate the encodings of event-based components and systems in Isabelle and the automatic transformation process to Isabelle.

5. Integration and Validation

To demonstrate the generative model-based approach, we model the PFS using SysML v2 from which the Isabelle code is automatically generated (for core Isabelle source code please see [8] and for more on the generator please see Appendix of [18]). We show the effectiveness of our encoding by formally analyzing a property of the avionic bus which is described by an event-automaton.
First, we exemplarily show a possible SysMLv2 state definition of the avionic bus. The bus automaton has one input and one output port. Furthermore it has two states named *Transmitting* and *Error*. After every received input event the underspecified bus decides nondeterministicly to assure or disturb the next transmission. This decision is specified by having transitions leading to the different bus states.

```
state def Bus(in input: Status, out output: Status) {
  entry; state Transmitting;
  /* shorthand where source state is the lexically previous one (Trans.)  */
  transition accept input do action { send input to output; }
                       then Transmitting;
  transition accept input do action { send input to output; }
                       then Error;
  transition accept --    then Transmitting;
  transition accept --    then Error;

  state Error;
  transition accept input do action { send Confirmed to output; }
                       then Error;
  transition accept input then Error;
  transition accept --    then Error;
  transition accept input then Transmitting;
  transition accept --    then Transmitting;
}
```

Listing 1: A bus event-automaton SysML specification

The bus component should never alter messages, despite being disturbed. The property to be checked restricts the buses output to only contain messages received as input. Please note that proving this property would provide us a necessary but not sufficient information, since the bus might still alter messages by exchanging them with previously received messages. To show this necessary

requirement, a theorem is formulated in Isabelle. It states that for every possible SPF (implementation) in the buses SPS (specification) and for every input bundle (contains only one stream), the messages of the output bundle occur in the messages of the input bundle.

```
1  theorem "∀ spf_bus ∈ sps_bus , input∈I^Ω.
2           let output = spf_bus input in
3           messagesOf output ⊆ messagesOf input"
```

The automatical provers and counterexample-finders are then left to run in parallel for a few seconds. It turns out that this property does not hold. By automatically checking for counterexamples, the couterexample tool "Quickcheck" [3, 7] provides the user with an example input stream $< [Inhibited, Inhibited] >$ where the output is evaluated to $< [Inhibited, Confirmed] >$. This violates the theorem statement. The transition in line 12 & 13 of listing 1 is faulty and allows sending message *Confirmed* if the automaton is in state *Disturbed*.
After deleting the faulty transition, the property can be shown. Furthermore, because of the compositionality of FOCUS the property immediately holds for any system the bus might be used in. Thus, the complete PFS system is validated in regards to the bus theorem.

6. Conclusion

We extended a model-based verification framework by allowing event-driven system specifications and reasoning. By detecting a non-conformance between the implementation and the abstract specification using formal counterexample finding, one can provide hints for potential developer mistakes or detect potential security vulnerabilities early at design time. 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 RTCA DO-333 standard, which can help to replace or complement many tests. Please note that the formal specification might create some additional effort when considering the overall benefits over testing. However, they overcompensate later significantly, since technical flaws at the beginning may result in highly expensive deficits, and the later the errors are corrected, the more expensive they are.
Acknowledgements: Thanks to Sebastian Stüber for helping with the encoding of event-automata in Isabelle.

7. Contact Author Email Address

mailto: raco@se-rwth.de

8. Copyright Statement

References

[1] Annighoefer, B., Halle, M., Schweiger, A., Reich, M., Watkins, C., van der Leest, S., Harwarth, S., and Deiber, P. Challenges and ways forward for avionics platforms and their development in 2019. In *38th Digital Avionics System Conference (DASC)* (2019).

[2] ARNIC Standards https://www.aviation-ia.com/content/arinc-standards.

[3] Berghofer, S., and Nipkow, T. Random testing in isabelle/hol. In *SEFM* (2004), vol. 4, Citeseer, pp. 230–239.

[4] Brahmi, A., Delmas, D., Essoussi, M. H., Randimbivololona, F., Atki, A., and 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, Jan. 2018).

[5] Broy, M., and Rumpe, B. Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. *Informatik Spektrum 30*, 1 (2007), 3–18.

[6] BROY, M., AND STØLEN, K. *Specification and development of interactive systems: Focus on streams, interfaces, and Refinement*. Springer, New York, 2001.

[7] BULWAHN, L. The new quickcheck for isabelle. In *Certified Programs and Proofs*, vol. 7679 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2012, pp. 92–108.

[8] BÜRGER, J. C., KAUSCH, H., RACO, D., RINGERT, J. O., RUMPE, B., STÜBER, S., AND WIARTALLA, M. Towards an Isabelle Theory for distributed, interactive systems - the untimed case. Tech. Rep. AIB-2020-02, RWTH Aachen University, March 2020.

[9] COFER, D., AND MILLER, S. Do-333 certification case studies. In *NASA Formal Methods* (Cham, 2014), J. M. Badger and K. Y. Rozier, Eds., Springer International Publishing, pp. 1–15.

[10] COMMON CRITERIA FOR INFORMATION TECHNOLOGY SECURITY EVALUATION (ISO/IEC 15408) https://www.commoncriteriaportal.org/files/ccfiles/CCPART3V3.1R4.pdf.

[11] ESPARZA, J., AND HELJANKO, K. Unfoldings - a partial-order approach to model checking. In *Monographs in Theoretical Computer Science. An EATCS Series* (2008).

[12] GROSU, R., AND RUMPE, B. Concurrent timed port automata. *arXiv preprint arXiv:1411.6027* (2014).

[13] HAREL, D., AND RUMPE, B. Meaningful Modeling: What's the Semantics of "Semantics"? *Computer 37*, 10 (2004), 64–72.

[14] HE, J., AND TURNER, K. J. *Specification and Verification of Synchronous Hardware using LOTOS*. Springer US, Boston, MA, 1999, pp. 295–312.

[15] HUFFMAN, B. C. *HOLCF '11: A definitional domain theory for verifying functional programs*. Portland State University, [Portland, Or.], 2012.

[16] KAUSCH, H., PFEIFFER, M., RACO, D., AND 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* (February 2020), R. Hebig and R. Heinrich, Eds., vol. 2581, CEUR Workshop Proceedings.

[17] KAUSCH, H., PFEIFFER, M., RACO, D., AND 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* (2020), p. 0671.

[18] KAUSCH, H., PFEIFFER, M., RACO, D., AND 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* (February 2021), S. Götz, L. Linsbauer, I. Schaefer, and A. Wortmann, Eds., vol. 2814, CEUR.

[19] KLEIN, G., ANDRONICK, J., ELPHINSTONE, K., MURRAY, T., SEWELL, T., KOLANSKI, R., AND HEISER, G. Comprehensive formal verification of an os microkernel. *ACM Trans. Comput. Syst. 32*, 1 (Feb. 2014).

[20] KRIEBEL, S., RACO, D., RUMPE, B., AND 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)* (February 2019), S. Krusche, K. Schneider, M. Kuhrmann, R. Heinrich, R. Jung, M. Konersmann, E. Schmieders, S. Helke, I. Schaefer, A. Vogelsang, B. Annighöfer, A. Schweiger, M. Reich, and A. van Hoorn, Eds., vol. 2308 of *CEUR Workshop Proceedings*, CEUR Workshop Proceedings, pp. 87–94.

[21] MILNER, R. *A Calculus of Communicating Systems*. Springer-Verlag, Berlin, Heidelberg, 1982.

[22] MOY, Y., LEDINOT, E., DELSENY, H., WIELS, V., AND MONATE, B. Testing or formal verification: Do-178c alternatives and industrial experience. *IEEE Software 30*, 3 (2013), 50–57.

[23] MURRAY, T., AND LOWE, G. On refinement-closed security properties and nondeterministic compositions. *Electr. Notes Theor. Comput. Sci. 250* (09 2009), 49–68.

[24] NIPKOW, T., PAULSON, L. C., AND WENZEL, M. *Isabelle/HOL: A proof assistant for Higher-Order Logic*, vol. 2283 of *Lecture notes in artificial intelligence*. Springer, Berlin [etc.], 2002.

[25] PARROW, J. An introduction to the pi-calculus.

[26] RADIO TECHNICAL COMMISSION FOR AERONAUTICS https://www.rtca.org.

[27] RINGERT, J., AND RUMPE, B. A little synopsis on streams, stream processing functions, and state-based stream processing. *Int. J. Software and Informatics 5* (01 2011), 29–53.

[28] RUMPE, B. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Doktorarbeit, Technische Universität München, 1996.

[29] SAE-INTERNATIONAL. Artificial Intelligence in Aviation G-34, June 2020.

[30] SAE-INTERNATIONAL. Cyber Physical Systems Security Committee G-32, June 2020.

[31] SCHOPP, U., SCHWEIGER, A., REICH, M., CHUPRINA, T., LUCIO, L., AND BRUNING, H. Requirements-

based code model checking. In *2020 IEEE Workshop on Formal Requirements (FORMREQ)* (Los Alamitos, CA, USA, sep 2020), IEEE Computer Society, pp. 21–27.

[32] SysML v2 Submission Team. OMG Systems Modeling Language (SysML). Tech. rep., Aug. 2021.

[33] SysML v2 Submission Team (SST) GitHub `https://github.com/Systems-Modeling`.