Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy's Streams Become Feasible?

Stefan Kriebel
BMW Group
Munich, Germany

Deni Raco
Chair of Software Engineering
RWTH Aachen University
Aachen, Germany

Bernhard Rumpe
Chair of Software Engineering
RWTH Aachen University
Aachen, Germany

Sebastian Stüber
Chair of Software Engineering
RWTH Aachen University
Aachen, Germany

Abstract—**Avionics is definitely a safety-critical application domain. Software complexity is ever increasing together with more autonomy as well as increased real-time based interaction between airplanes, drones and potentially future air taxis.**

This again raises the question, whether developing software the same way as we did the last 30 years is still appropriate, or in the times of much better formal methods and cheap and powerful computational capabilities, it would be feasible to use clear and model-based specification techniques for an integrated systems engineering approach and formally verify any physical and logical implementation of functionality, including the software against that specification. This could be another important step towards quicker development of highly safety-critical systems.

I. Introduction

In this paper, we demonstrate early results on a small part of this systems engineering approach, namely using a user-friendly architecture description language [1], [2] and proving safety-critical properties in the theorem prover Isabelle [3]. Because the theory and its underlying methodological concepts are too much to be explained in this short paper, we highlight a small example only, focusing on the demonstration of the feasibility of formal verification of complex software.

Therefore, the paper demonstrates the use of the FOCUS framework [4]–[8] for specifying distributed interactive systems and verifying safety-critical properties of real-time critical software such as common in avionics systems. The methodology is modular with respect to composition. A user-friendly architecture description language (ADL) like MontiArc [1] serves as a developer frontend, allowing a state-based specification of components [8] and enables the definition of the desired safety-critical properties. An appropriate tool maps the ADL model and its behavior specifications into specifications and theorems in the theorem prover Isabelle [3]. There, general composition operators exist that compose specifications of the components into a specification of the overall system. Properties can be specified on a global scale and decomposed as well and proven on atomic components. Composition of properties leads to globally correct software. Furthermore, refinement of a component in a decomposed structure automatically leads to refinement of the composition, which allows to individually develop each component, but also to replace variants of implementations along the life cycle of a system. High-automation of the proofs, as well as the handling of feedback loops, unbounded non-determinism, time-sensitive specifications, safety and liveness properties, and refinement checking are key for an efficient development process.

The rest of this paper is structured as follows: First, a motivation for the methodology is given. Then, the underlying theory is presented. Next, the tool chain consisting of the frontend DSL, the mathematical backend, and the generator is illustrated. Finally, the verification of a property of the running example is demonstrated.

II. Background

Designing distributed systems [6], [9] which react in real-time, are dependable and fulfill safety and liveness requirements, is a challenging problem [10]. Formulating requirements only in natural language is still common in the embedded industry and the challenging part is to early detect and avoid ambiguity [11] and inconsistencies [6]. In safety-critical systems errors might lead to injury or high costs as consequences [12]. For this reason, formal methods, like CSP [13], [14], FOCUS [6], CCS [15], Petri Nets [16], or the π-calculus [17] are used to detect potential sources of errors earlier [18], [19]. However, an important property which makes the methodology of this paper stand out among the competitors above is that refinement of a component in a decomposed structure automatically leads to refinement of the composition [7] (thus refinement is fully compositional).

Avionics have long life cycles and a lot of maintenance is needed [20]. Further growth in flying vehicles, passenger numbers and cargo is expected. Since the spatial expansion possibilities of the system infrastructure are limited, the use of correct software systems becomes vital [21]. It is thus not without reason that the list of considerations for the production of software for airborne systems named ED-12C/DO-178C has as its motto: *"the greater the software development rigor, the fewer errors occur in software design"* [22].

Safety-critical functionalities in avionics are usually treated by a complex system management, which deals with fault detection and redundancy management [23]. The increasing complexity in avionics is heavily due to [24]:

- the increasing number of features,
- the internal complexity of each feature,
- interaction among features,
- better scientific instruments,
- processing large amount of data in real time,
- thousands of sensors and measurement devices,
- redundant components, ready to automatically configure themselves in case of failures.

For a stepwise correct design of avionics systems, a model-based methodology [25], [26] would be beneficial. A modeling ADL offers the architect a way to express the knowledge about a process, and the created models can be translated into other models or executable code [27]. Most importantly, the models of the software can be tested or verified, independent of the hardware. By having testing or verification being performed parallel to the software model design, before committing to hardware, the likelihood that design errors are recognized late and potentially bring high costs is reduced.

Some further reasons for the usefulness of testing or verifying the software models are [24]:

- by verifying early in the life cycle before hardware is available, one reduces the number of needed tests on the physical system,
- software models can enable one to have access to sub-modules and test those separately, whereas in hardware this might be in some cases physically not possible,
- even if sometimes accessing hardware sub-modules might be possible, it might mean causing damage to it,
- testing fault protection behaviors might be highly costly and damaging.

Testing components, however, on all combinations of inputs and states (represented by the variable values at a certain point in time) might be not feasible. In addition, when comparing testing strategies from the literature, it is usually hard to show that one testing strategy would always detect more faults than another one [28].

In this paper a methodology for specifying and verifying distributed interactive systems is demonstrated and applied on an example.

III. Underlying Theory

The methodology presented in this paper builds on the mathematical underpinning FOCUS [6] and is implemented as a tool chain depicted in Fig. 1, being particularly suited for usage in safety-critical systems such as avionics. The frontend of the tool chain MontiArc [1] (created with MontiCore [29], [30]) is a developer-friendly domain-specific language for the specification of component interfaces, their behavior and their composition. A MontiArc model is transformed by a code generator into an Isabelle automaton specification, and then this (potentially non-deterministic) automaton is mapped to its semantics, namely a (set of) stream processing functions.

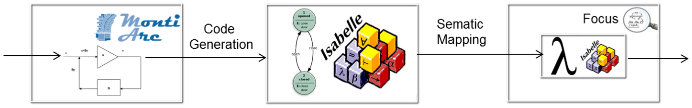

Fig. 1. Verification tool chain

Components communicate through sending and receiving messages through directed channels. A stream (a potentially infinite sequence of messages from an alphabet) [6] represents the communication history of a channel. The semantics [31] of a (potentially non-deterministic) component is a (set of) stream processing functions [8].

Since streams model history, not all functions over streams model real-life interactions. Only a subset of these functions fulfilling certain properties are suited to model real-life components. A component cannot take an already emitted message back. This means that an extension of the input can only lead to an extension of an output. This is known as *monotonicity* [8]. Second, to define liveness properties, one needs infinite streams to describe full histories. It is however not implementable to look at the complete (infinite) input stream to emit an output message (which of course occurs after a finite time). Enforcing this means restricting the set of functions to the so-called *continuous* ones [4]. In addition to restraining from reacting to infinity, continuous functions also guarantee the existence and the inductive computation of least fixed points, which is necessary to give meaning to feedback loops [6], [8], [32].

All above mentioned domain-theoretical concepts has been formalized in the theorem prover Isabelle/HOLCF (HOL stands for higher-order logic, CF stands for computable functions) in [33], [34] and [35] independently formalized parts of FOCUS in Isabelle/HOL (without domain-theory). [36]–[41] were some first results in formalizing streams in the theorem prover Isabelle with domain-theoretical concepts, and constitute the foundation of this paper. On the other hand, the tool-chain named AutoFOCUS [42] has had early results in using the HOL-formalization of FOCUS. A further related work to formalize model component networks is the Ptolemy Project [9], [43] where the authors create a framework for actor-oriented design. Also, an approach to verify AADL-models is presented in [44].

For simple event-based communication, an untimed stream approach is sufficiently expressive to model an interactive system. In real-time systems, however, a timed model is much more appropriate. This allows software to react on absence of messages and signals. A formal specification of such a software component therefore does not only incorporate reactive behavior, but also timing specification on both sides: along with a component's reaction, it is also considered how long the neighbor systems, sensors etc. have time to respond. One mathematically simple and powerful way to model time is to extend the set of sent messages with a *virtual element* called *tick* and enforce infinitely many ticks in each infinite behavior observation. Two consecutive ticks describe an equidistant time interval with only finitely many occurring messages. In certain circumstances this observation may be reduced to at most one or even exactly one message per time interval. Focus is capable of handling all these variants even in combination. Focus can also look at different time scales [45], [46] allowing the developer to select the right time abstraction for each component.

As said, specifications of component behavior are mathematically formalized as a set of functions mapping timed input streams to timed output streams. To model correct behavior (i.e. not looking in to the future), some restrictions, such as *weak-causality* or even embodying some delay in form of *strong-causality* e.g. to avoid the Brock-Ackermann anomaly [6], [47] in feedback compositions. The concept of monotonicity, continuity and causality are often generally referred to as *realizability* or *interactive computability* [48], the equivalent concept to computability of partial functions carried over to interactive systems.

To support system decomposition, operators for sequential, parallel and feedback composition are encoded in Isabelle, as well as a general composition operator [7], [49]. A high-level API is provided in this work to hide fixed-point theory from the user.

One of the strengths of the methodology of this work is that by using a variant of non-deterministic and thus underspecified state machines with input/output [8] to define the components behavior, the system is by construction realizable [6], [50]. Those state machines receive their semantics as sets of stream processing functions [8] and are thus fully integrated into the refinement and composition framework that FOCUS provides within Isabelle. These realizable stream processing functions are an abstraction to the automata with input/output from [7], [8] in the same way that partial functions are an abstraction to the Turing machines.

One important particularity of this methodology is that the composition of realizable functions is realizable as well [7], [49]. So a realizable composed system can be hierarchically decomposed into a collection of realizable components, each represented by a (set of) stream processing functions. This way a complex architecture can be built in a fully compositional way.

As already mentioned, sets of functions are used to give specifications a meaning allowing multiple behaviors, either due to potential non-determinism of the components, or due to insufficient information during development.

Refinement is used to make underspecified components specifications more precise along the development process. Underspecified behavior can be refined towards an implementation. Refinement of non-deterministic automata is semantically represented by set inclusion of sets of stream processing functions [7], [8]. A number of important refinement techniques, such as *transition refinement* and *state decomposition* are formalized in order to automate the checking of refinement correctness. Transition refinement means that by removing one of a set of alternative transitions, the set of behaviors becomes more precise. State refinement allows e.g. splitting states by inserting substates.

The most important property which makes this methodology stand out among competitors (like CSP [13] [14], CCS [15], Petri Nets [16], or the π-calculus [17]) is that refinement of component specifications is semantically reflected by the concept of set inclusion between function sets and that refinement of a component in a decomposed structure automatically leads to refinement of the composition [7]. That means, *refinement is fully compositional*. This property is a major reason why this proposed methodology is well-suited to specify larger and complex distributed systems, because it allows to scale a specification from atomic components to large systems, such as airplanes. One can specify a system, decompose the specification, refine the individual sub-systems until an implementation is reached, and then have the guarantee that the composition of the implementations is correct by construction.

Furthermore, a component library with popular components (NOT, AND, OR, NOR, XOR, ADD, Delay, variants of a transport medium, counters, timers, etc.) was created to support specifications. For example the transport medium-components have an abstract non-deterministic specification (the set of implementations describing their possible behaviors is even uncountable) and also a number of refined behaviors, resembling special characteristics such as liveness properties. In essence, without going deeper in details in this short paper, the formalization of refinement checking of (potentially unbounded) non-deterministic specifications consisted in the encoding in Isabelle of the corresponding refinement calculus from [8]. The correctness of the refinement steps is also used to show that the system doesn't gain new, potentially undesired, behaviors and the properties of the initial system are automatically derived for the refined system as well (the user is thus relieved from the necessity to prove these again for the new system). Along with the specification, a set of useful theorems and their proofs are generated for each component, which increases the automation of the proof of the desired system property.

Finally, abstract theorems are provided to enable an automatic checking of safety and liveness properties. We understand informally by

- *safety*: properties that can be falsified by finite streams.
- *liveness*: properties that can be falsified only by infinite streams.

The methodology is demonstrated in this paper on a feature interaction scenario of a light control, which is actually taken from the automotive domain, but we expect that the avionics domain could be very similar. The (small part of the) system and a desired property are defined through an explicit specification and the desired safety-critical property is proven at the push of a button. The full-automation is the result of:

- the encoding of thousands of general, case study independent theorems for interactive components and their composition in Isabelle,
- a library of common generic reusable components together with proven properties over these,

- the development of the corresponding code generator for all mathematical structures introduced above.

This case study demonstrates the correct encoding and usage of a general composition operator, dealing in particular successfully with feedback cycles by generating appropriately a delay, as well as handling time-sensitive specifications.

IV. The developer-friendly ADL MontiArc

A user friendly domain specific ADL is presented, to enforce the specification of realizable-per-construction components [1]. It enables the high-level specification of composed systems, as in Fig. 2. A running example dealing with communicating controllers in vehicles is used to ilustrate this.

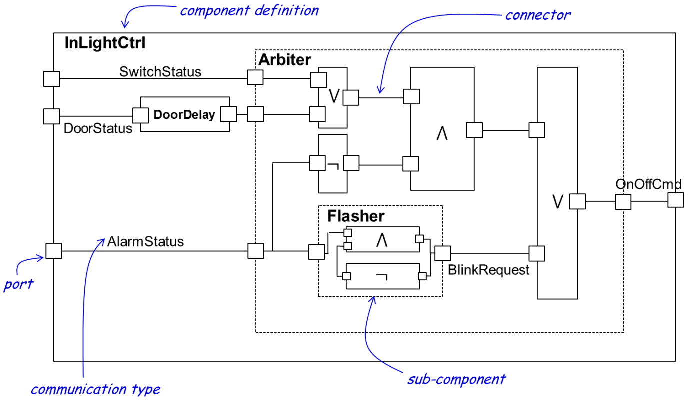

Fig. 2. Communicating controllers in vehicles

In a black box view a component is defined by its input and output channels. Each channel is identified by a name and the data type of the messages allowed to flow in it. Generic data types are supported in order to facilitate reuse. Composed systems are defined by importing the subcomponents and connecting the channels. Output channels of a component can be connected to input channels of the same or of other components. It is allowed to use the same subcomponents multiple times. Generic data types can be instantiated multiple times. Additional information can also be added to the model. For example invariants of components, or whether the component is deterministic or not.

The depicted controller *InteriorLightArbiter* controls the interior light of a car. The description of the formal specification of interfaces, behavior and composition will be introduced later. As can be seen, this system produces its output based on the light switch, the car's doors and the alarm system. Depending on these inputs, it emits a command to turn the interior light on or off. For example it evaluates the light switch status by simply forwarding its status. The interior light is also switched on if the door is opened, and goes out a short while after the door has been closed again. A closed door only changes the light status if the door is closed since 5 seconds. If the alarm system is active, the interior light blinks. As long as its incoming signal value is *on*, the Flasher outputs alternatingly *on* and *off* values.

Chosen safety-critical property informally: *The alarm system always has highest priority. So if the alarm is activated, the interior light will turn on, no matter how both the door status and the light switch behave.*

V. MontiArc in Isabelle

An *untimed stream* is a (potentially infinite) sequence of messages over a carrier alphabet M. M^ω denotes all streams and is the union of M^* the finite ones and M^∞ the infinite ones. To construct streams, the constructor " : " with signature $M \Rightarrow M^\omega \Rightarrow M^\omega$ is defined. The operator $\frown$ denotes the concatenation of two streams [6]. Based on the construction operator on streams, we can define an ordering $\sqsubseteq$ on the set of streams. The prefix ordering [8] on streams $\sqsubseteq$ is defined such that the following holds:

$$\forall x, y \in M^\omega.\ x \sqsubseteq y \Leftrightarrow \exists s \in M^\omega.\ \ x \frown s = y$$

Please note that the prefix order defines a partial order on streams [8] and that M^ω completes M^* to a complete partial order [34] with respect to prefixing.

To specify time-sensitive behavior, timed streams are used (in our case study the so-called *time-synchronous streams* variant). For this, the message alphabet is extended by a dummy element $\sim$ (read *eps*). In this interpretation we assume a discrete global clock and each element of the stream is either a message arriving during each (equidistant) time frame, or an $\sim$ (interpreted here as "no message has arrived", having also the length of one time frame).

Streams have been encoded in the interactive proof assistant Isabelle [51]. The proofs, data structures as well as functions are there formalized in so-called theory files, which have the following structure:

```
theory ExampleTheory
imports Main
begin
(* definitions and lemmas *)
end
```

The implementation in the theorem prover Isabelle of the data type `stream` is, apart from some technical domain-theoretical details [37], similar to lazy lists of (say) Haskell.

```
domain 'a stream =
  lsconc "'a" (lazy "'a stream")
```

The keyword `domain` [34] generates the prefix ordering and a bottom element (the empty stream). The constructor `lsconc` appends an element to the rest of the stream.

Fig. 3 gives an overview of our theories, such as Stream Bundles (SB), stream processing functions (SPF), and sets of functions denoted as stream processing specifications (SPS). Foundamental theories such as LNat (lazy natural numbers, where the set of naturals is extended with an element denoting infinity), or SetPcpo (sets are enhanced with a subset-order) are needed as well. Theory imports are represented as arrows in the Figure. These mathematical structures will be briefly described later.

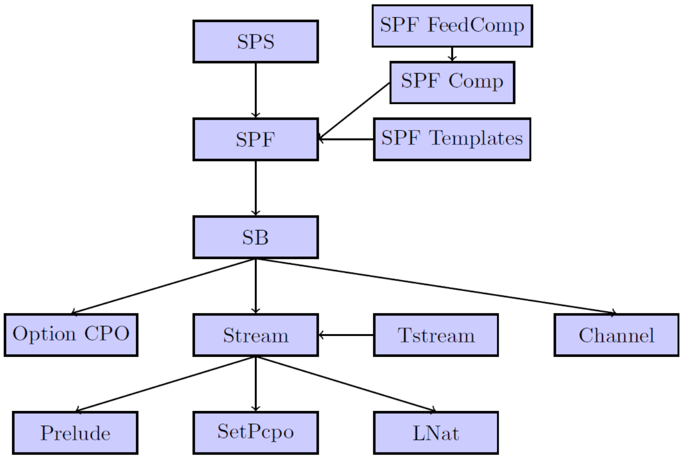

Fig. 3. Structure of the theories

The extension to time-synchronous streams is also done similarly as one would extend a data type in Haskell: given the type stream over a parametric type-variable $'a$, and a constructor for messages Msg, one can then define:

```
datatype 'a tsyn = Msg 'a | ~
```

An instance of a stream of natural numbers over tsyn would then be e.g: Msg 1, ∼, Msg 2, ∼, ∼,.... This is read as: the first time slot sees the message 1 arriving; in the second time slot no message arrives; in the third one the message 2 arrives, and after that no more messages arrive. The granularity of each time interval can be set depending on the case study (say ”minutes”, if we are modeling bus arrival times, or say ”milliseconds”, if we are modeling communication inside a computer processor.)

In our light controller the elements of the streams flowing in the channels are from the carrier set $\mathbb{B} \cup \{\sim\}$.

VI. Stream bundles and Stream Processing Functions

To facilitate composition, we enhance our modeling of component networks by naming channels and defining composition operators which connect channels of the same name and type. The user can then define the type of a channel via a function which for each channel returns a set of allowed messages, i.e., the domain of the channel type. To model the input (or output) streams of a component, we work with an isomorphic transformation of the tuples of streams (instead of just working on tuples): namely with mappings from channel names to streams. Such a mapping is then called Stream Bundle [8] if the messages of the streams mapped to the channels are allowed to flow on it. Thus, we can compose components and define generalized composition operators [7] connecting same-named/same-typed channels without worrying about setting preconditions for the interface compatibility. In the case of (say) an addition component, the encoding in Isabelle of the interface of the input would be the structure of the form $[channel_1 \mapsto stream_1, channel_2 \mapsto stream_2]$ (instead of the intuitive tuple $(stream_1, stream_2)$, which does not offer the flexibility in defining general composition over arbitrary number of channels).

A deterministic component is modeled by a stream(bundle) processing function (the type denoted as SPF), which is then a continuous function mapping stream bundles to stream bundles. The semantics of a non-deterministic automaton is a set of stream processing functions (the type denoted as SPS).

VII. State-based modeling

As mentioned, state-based modeling is used to enforce the specification of realizable-per-construction components. We demonstrate an example by specifying the behavior of the DoorDelay component in Fig. 4:

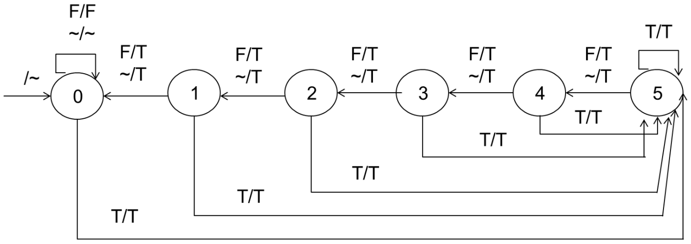

Fig. 4. Behavior of DoorDelay as Automata

This figure is a graphical representation of the behavior represented by the following MontiArc textual description. The time model in the textual description is set to time-synchronous (sync). Then the interface of the component (ports and input/output channels) is defined. A variable `delay` is used to represent the states. Finally the behavior description is given by listing the transitions. Transitions are enhanced by input/output.

```
component DoorDelay {

 timing sync;

 port
  in boolean i,
  out boolean o;

 int delay;

 automaton DoorDelay {
  state S;
  initial S / {delay=0};

  S [!i && delay>0] /
                {delay=delay-1, o=true};
  S [!i && delay==0] / {o=false};
  S [i==null && delay>0] /
                {delay=delay-1, o=true};
  S [i==null && delay==0] / {o=false};
  S [i] / {delay=5, o=true};
 }
}
```

As a contrast to state-based specifications, a specification such as $DoorDelay[a : b : c : d : e : xs] = f(a, b, c, d, e, xs)$ for some function f, messages a, b, c, d, e, and sequence of messages xs might lead to non-realizable behavior, since one can make use of (say) b in the first output time slot before it has arrived as input.

The behavior of the AND, OR, and NOT components is a straightforward lifting from booleans to streams of booleans and the boolean values have priority over $\sim$.

The behavior of a MontiArc component is specified as automata with input/output [8]. Automata with input/output consist of states and transitions. The states of the automaton are used to save information about the current state of the computation. In addition to states java-variables can be used in MontiArc.

Transitions help define the behavior of a component. A transition describes the output and new state of a component. Non-determinism can be modeled by multiple transitions or by a single transition with a set as result.

Each automata is translated in a final step into (sets of) stream processing functions, which constitute their semantics [8]. The Isabelle theory of stream processing functions is rich with theorems, which increase the automation of the proofs.

VIII. Composition and feedback loops

To decompose and then re-compose components in a development life cycle, special composition operators of Fig. 5 were encoded using the notations as described in [7]

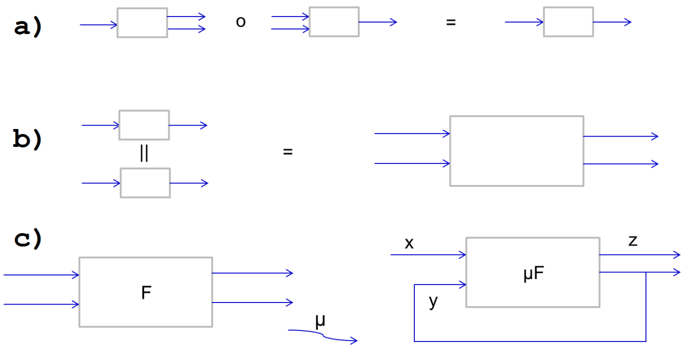

Fig. 5. Special Composition Operators

Serial composition in a) is quite straightforwardly overtaken from function composition in mathematics, and parallel composition in b) creates a new component with an extended interface.

The μ-operator for feedback in c), such as the one occurring in the Flasher-Component, is defined as follows: for streams x, y, z we have $(z, y) = (\mu f).x$, if (z, y) is the least fixed point of the equation $(z, y) = f(x, y)$.

Streams flowing on feedback loops are defined as least fixed points of the corresponding equations [5]. Monotonicity is neccesary for least fixed points to be unique.

A general composition operator $f \otimes g$ was encoded [5], [7], [39] as well, covering all possible combinations of the above mentioned special operators as shown in Fig. 6, where $c_1...c_8$ denote here the channel names.The general operator is equally powerfull to the combination of all special operators. The generated stream processing functions are automatically connected by this operator. The proof of commutativity and associativity [6] of this operator is also encoded in Isabelle. This means that the order of composition of a list of components does not matter.

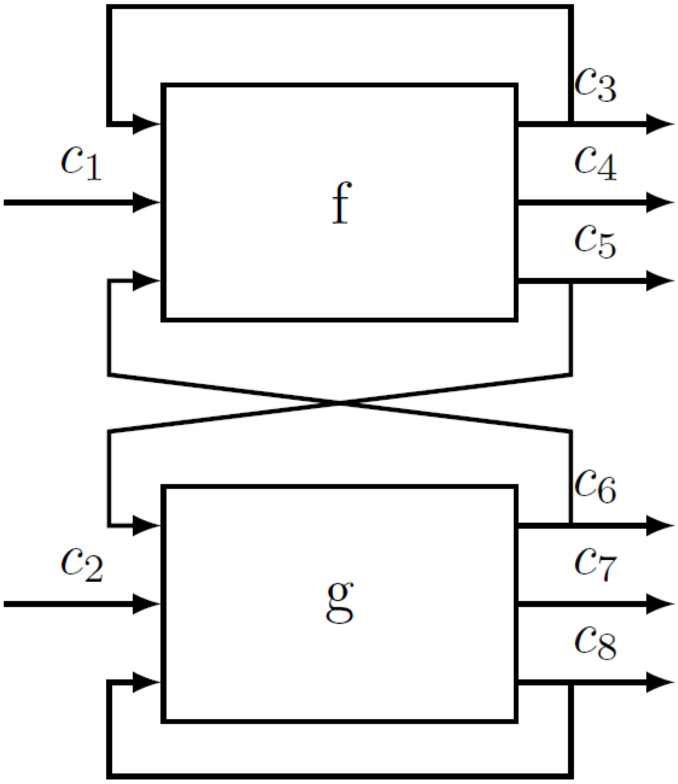

Fig. 6. General Composition Operator

IX. Non-deterministic specifications

Our mathematical model is expressive enough to model interesting aspects of software development such as underspecification and refinement as well. From a user's point of view, it is not distinguishable, whether a system is underspecified (further refinement steps during the development process can make specifications more precise), or it makes non-deterministic decisions on runtime. In the introduction of this paper, we already mentioned that components may be under-specified or non-deterministic. Thus, a single deterministic SPF is not sufficient to describe all possible component behaviors, and instead a set of stream processing function must be used to model the component behavior properly [49]. Nevertheless, the input and output channels of components are fixed, thus all stream processing functions in such a set must have the same input and output channels.

X. Code generator from MontiArc to Isabelle

To verify the properties of user-defined component systems, they are automatically transformed to specifications and theorems in Isabelle.

The equivalence of the user-specification and the generated Isabelle-specification is imperative for any logical reasoning. A complex generation process could lead to different semantics. To reduce complexity of the transformation, MontiArc-ADL and Isabelle-Specification are using similar concepts.

An automata [8] is first transformed from the MontiArc model to an automata in Isabelle. The abstract syntax of automata is encoded in Isabelle as well. In a second step the automata is mapped to its semantics, a set of stream-processing functions [8]. The second step is entirely within Isabelle and its correctness is proven.

A composed specification is realized through the general composition operator [39]. Since the general composition operator can only connect channels with the same name, internal channels are used. These internal channels are not visible in the public interface of the component.

XI. Verification of Properties

MontiArc invariants are mapped to Isabelle lemmata. Simple properties can be proven automatically, whereas more complex properties might require user interaction. To simplify any manual proof, additional lemmata are generated. One can specify his desired (potentially safety-critical) property in MontiArc. The framework is extended sufficiently with abstract theorems, such that most simple properties will be checked on the push of the button. This is also the case for the chosen property of this work, thus showing promising results about the feasibility of the approach. It is shown that the alarm status has priority over other inputs and it guarantees the turning on of the light, no matter how both the door status and the light switch behave.

Let $snth\ n$ denote the time slot of a message in a stream for some natural n, $AlarmStatus$ the input stream of the light controller, and $OnOffCmd$ the output stream of the light controller. The theorem is then formulated as follows:

theorem: $\forall\ n \in \mathbb{N}$: snth n AlarmStatus = True $\Rightarrow$ ((snth n OnOffCmd = True) $\vee$ (snth $(n-1)$ OnOffCmd = True))

$< proof >$

The proof is automatically generated.

Here is a proof sketch. First one checks the correctness of the single components. The Flasher is trickier, since it contains a feedback loop. It was first shown, that the output stream of the Flasher corresponds to the certain desired least fixed point. The proof of correctness for components with feedback loops and for those with a state(such as DoorDelay) takes usually the majority of effort to automatize. The correct behavior of the OR-component is also proven. The relation between the AND-Component and the Flasher is mapped to a general composition AND $\otimes$ Flasher. Then this composition is proven to be reducible to a parallel composition between these. As a next step, the composition Flasher $\otimes$ OR is shown to be reducible to a serial composition. These reductions are recognized automatically by investigating the shared channels between components. The large amount of encoded theorems about the special operators, as well as the extension of the code generator with common component-specific properties, take care of the rest, making this property proven at the push of a button.

XII. Conclusion

In this paper, we have seen that there is an appropriate theory, namely Broy's streams [6], which is able to describe behavior of real-time capable distributed and complex software in a hierarchically decomposable form. Furthermore, along the development process refinement of underspecified component behavior is possible and is fully compatible with the composition operators. That means decomposed subcomponents can be individually implemented and desired properties proven on local components thus that the overall composed system is then correct by construction.

Correct by construction means that there is no complicated integration phase with lots of errors only identified late in the development process. Instead a rather agile development process could become possible: It has an always integrated composed system with individual subsystems hierarchically decomposed and individually refined along the development.

The encoding of Broy's Stream Theory in Isabelle and the available comfortable modeling techniques, such as an ADL as well as state machines are an important step towards a rigorous development process based on model-based specifications and formal verification.

However, there are still a lot of steps to do. (1) The integration of modeling techniques as developers frontend needs to be tighter and potentially also handle other forms of modern specification languages. (2) The library of available predefined components and their specifications must be intensively extended. (3) The proof assistant system needs to be robust and as automatic as possible in any kind of potential situations. Ideally the proof assistant is so highly automated, that ordinary software developers do not explicitly have to cope with proving activities at all, but can concentrate on specifying behaviors on different levels of abstraction, while the underlying proof assistant tells the developers automatically, whether their development steps have been correct. This would lead to a *continuous verification system* quite like the currently already existing continuous integration environments [52] for compilation, generation and testing.

Of course, in practice a combination of all these techniques is desired. Based on our experiences, we believe that formal verification should actually be a strong tool in the toolbox of a mature Software Engineering discipline. However, Software Engineering is still not mature and only future research and industrial applications can show whether and how formal verification tools will be part of our future toolbox.

References

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

[2] Aerospace SAE. Architecture analysis and design language. 2004.

[3] Tobias Nipkow Lawrence C Paulson and Markus Wenzel. A proof assistant for higher-order logic, 2013.

[4] Manfred Broy, Frank Dederichs, Claus Dendorfer, Max Fuchs, Thomas F. Gritzner, and Rainer Weber. *The design of distributed systems: An Introduction to FOCUS*. Citeseer, 1992.

[5] Manfred Broy. Functional specification of time-sensitive communicating systems. *ACM Transactions on Software Engineering and Methodology*, 2(1):1–46, 1993.

[6] Manfred Broy and Ketil Stølen. *Specification and development of interactive systems: Focus on streams, interfaces, and Refinement*. Springer, New York, 2001.

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

[8] Bernhard Rumpe. *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*. Doktorarbeit, Technische Universität München, 1996.

[9] Edward A. Lee. Fundamental limits of cyber-physical systems modeling. *ACM Transactions on Cyber-Physical Systems*, 1(1), 11 2016.

[10] Chih-Hong Cheng, Michael Geisinger, Harald Ruess, Christian Buckl, and Alois Knoll. Mgsyn: Automatic synthesis for industrial automation. volume 7358, pages 658–664, 07 2012.

[11] Grischa Liebel, Anthony Anjorin, Eric Knauss, Florian Lorber, and Matthias Tichy. Modelling behavioural requirements and alignment with verification in the embedded industry. pages 427–434, 01 2017.

[12] Rashidah Kasauli, Eric Knauss, Benjamin Kanagwa, Agneta Nilsson, and Gul Calikli. Safety-critical systems and agile development: A mapping study. pages 470–477, 08 2018.

[13] C.A.R Hoare. Communicating sequential processes.

[14] Robert Heim, Pedram Mir Seyed Nazari, Jan Oliver Ringert, Bernhard Rumpe, and Andreas Wortmann. Modeling robot and world interfaces for reusable tasks. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 1793–1798. IEEE, 2015.

[15] Robin Milner. *Communication and concurrency*, volume 84. Prentice hall New York etc., 1989.

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

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

[18] Anthony Hall. Seven myths of formal methods, 1990.

[19] Shahar Maoz, Nitzan Pomerantz, Jan Oliver Ringert, and Rafi Shalom. Why is my component and connector views specification unsatisfiable? 2017.

[20] Andreas H Schweiger. Applying software patterns to requirements engineering for avionics systems. *2013 IEEE International Systems Conference (SysCon)*, pages 25–30, 2013.

[21] Ralf God and Ulrike Wittke. Cyber-physical aviation. pages 4–6, 2016.

[22] Grischa Liebel, Anthony Anjorin, Eric Knauss, Florian Lorber, and Matthias Tichy. Modelling behavioural requirements and alignment with verification in the embedded industry. 01 2018.

[23] Darbaz Darwesh, Bjoern Annighoefer, and Reinhard Reichel. A demonstrator for the verification of the selective integration of the flexible platform approach into integrated modular avionics. 09 2018.

[24] Mohammed Khan, Michael Sievers, and Shaun Standley. Model-based verification and validation of spacecraft avionics. 06 2012.

[25] Bernhard Rumpe. *Modellierung mit UML*. 01 2004.

[26] Bernhard Rumpe. *Agile Modellierung mit UML: Codegenerierung, Testfälle, Refactoring*. 01 2012.

[27] Stephan Rudolph. *Know-How Reuse in the Conceptual Design Phase of Complex Engineering Products*, pages 23–39. 01 2007.

[28] Ilinca Ciupa, Alexander Pretschner, Manuel Oriol, Andreas Leitner, and Bertrand Meyer. On the number and nature of faults found by random testing. *Softw. Test., Verif. Reliab.*, 21:3–28, 2011.

[29] Holger Krahn, Bernhard Rumpe, and Steven Völkel. Monticore: a framework for compositional development of domain specific languages. *CoRR*, abs/1409.2367, 2014.

[30] Katrin Hölldobler and Bernhard Rumpe. *MontiCore 5 Language Workbench Edition 2017*. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

[31] David Harel and Bernhard Rumpe. Meaningful modeling: what's the semantics of" semantics"? *Computer*, 37(10):64–72, 2004.

[32] Stephen Cole Kleene. *Introduction to metamathematics*, volume v. 1 of *Bibliotheca mathematica. A series of monographs on pure and applied mathematics*. Wolters-Noordhoff Pub, Groningen, 1952.

[33] Franz Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF*. na, 1994.

[34] Brian Charles Huffman. *HOLCF '11: A definitional domain theory for verifying functional programs*. Portland State University, [Portland, Or.], 2012.

[35] Maria Spichkova. *Specification and Seamless Verification of Embedded Real-Time Systems: FOCUS on Isabelle*. VDM Verlag Dr. Müller Aktiengesellschaft & Co. KG, Saarbrücken, 2008.

[36] Borislav Gajanovic and Bernhard Rumpe. Isabelle/hol-umsetzung strombasierter definitionen zur verifikation von verteilten, asynchron kommunizierenden systemen. Technical report, Technical Report Informatik-Bericht 2006-03, Braunschweig University of Technology, 2006.

[37] Borislav Gajanovic and Bernhard Rumpe. Alice: An advanced logic for interactive component engineering. *CoRR*, abs/1410.4381, 2007.

[38] Sebastian Stüber. *Eine domain-theoretische Formalisierung von strombündel-verarbeitenden Funktionen in Isabelle*. Bachelorarbeit, RWTH Aachen, 2016.

[39] Jens Christoph Bürger. *Modular Hierarchical Modelling and Verification of Systems using General Composition Operators*. Bachelorarbeit, RWTH Aachen, 2017.

[40] Marc Wiartalla. *Compositional Modelling and Verification of Distributed Systems with special Composition Operators*. 2017.

[41] Mathias Pfeiffer. *A Code Generator for Modeling Underspecification of State-based Network Components*. Masterarbeit, RWTH Aachen, 2018.

[42] Sebastian Voss and Sergey Zverlov. Design space exploration in autofocus3 - an overview. In Vladimír Mařík, JoseL. Martinez Lastra, and Petr Skobelev, editors, *IFIP First International Workshop on Design Space Exploration of Cyber-Physical Systems*. Springer, 2014.

[43] Edward A. Lee. Computing needs time. *Communications of the ACM*, 52(5):70–79, May 2009.

[44] Bernard Berthomieu, J.-P. Bodeveix, Silvano Dal Zilio, Pierre Dissaux, Mamoun Filali, Pierre Gaufillet, Steve Heim, and François Vernadat. Formal verification of aadl models with fiacre and tina. 2010.

[45] Manfred Broy. (inter-)action refinement: The easy way, 1993.

[46] Manfred Broy. Refinement of time. In *ARTS*, 1997.

[47] J Dean Brock and William B. Ackerman. Scenarios: A model of non-determinate computation. pages 252–259, 01 1981.

[48] Manfred Broy. Computability and realizability for interactive computations. *Inf. Comput.*, 241(C):277–301, April 2015.

[49] Jan Oliver Ringert and Bernhard Rumpe. A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. *Int. J. Software and Informatics*, 5(1-2):29–53, 2011.

[50] Manfred Broy. Computability and realizability for interactive computations. *Information and Computation*, 241, 01 2015.

[51] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A proof assistant for Higher-Order Logic*, volume 2283 of *Lecture notes in artificial intelligence*. Springer, Berlin [etc.], 2002.

[52] John Ferguson Smart. *Jenkins: The Definitive Guide*. O'Reilly Media, Inc., 2011.