Leveraging Highly Automated Theorem Proving for Certification

Deni Raco, Bernhard Rumpe, and Sebastian Stüber

Software Engineering, RWTH Aachen University, Germany www.se-rwth.de

Abstract. This work demonstrates how highly automated theorem proving can be leveraged for sparing testing costs during certification of safety-critical software such as in avionics. A verification framework for distributed interactive systems is presented. Components are modeled as stream processing functions. The functional methodology is modular with respect to serial, parallel and feedback composition. To specify and formally verify properties of distributed systems, a stream-based verification infrastructure is encoded in the theorem prover Isabelle. Composition operators for components are provided, thus allowing to scale proofs from individual components to complex networks. The underlying mathematical theory FOCUS stands out among competitors by the fact that refinement is fully compositional. The associativity and commutativity of the provided general composition operator enables a compositional verification. In this paper the stream theory encoded in Isabelle is demonstrated. This represents a small part of our encoding of the methodology focusing only on channel histories specifications, yet sufficient to demonstrate dealing successfully with underspecification, supporting automatic refinement checking during design time, time-sensitive specification, as well as verifying safety and liveness properties. The theory is evaluated in a case study where the occurring refinement steps from system requirements, to high level requirements, to low level requirements, until an implementation is reached, are demonstrated to be proven correct at the push of a button.

Keywords: Formal Verification · Distributed Systems · Certification

1 Industrial Background

In a software development process, such as the one depicted in Fig.1 for avionics, checking the correctness of the refinement steps from system requirements, to high level requirements, to low level requirements, until the source code, can get very costly to execute by reviewing and testing[2]. Formal methods can provide help for this. Well-known classes of formal methods are[40]:

- Theorem Proving:
  - Most powerful, most expressive formal methods tools.
  - Require expertise and continuous interaction to successfully use.

- Model Checking:
  - Less expressive than theorem provers.
  - Mostly automated, but still require expertise to use successfully.
- Abstract Intepretation and Static Analysis [29, 7, 38, 22, 42]:
  - Least expressive, targeted to very specific artifacts.
  - Require some expertise to discharge false positives.

Theorem proving [5] can assure correctness, a confidence level in general not reachable by testing. Furthermore, it can result in sparing costs on testing (partially replaces it, partially complements it), which is known as the most costly part of the development of safety-critical systems [10, 2].

Using theorem proving, one can show e.g. that the high-level requirements are consistent (i.e. do not contradict each other) by proving that there is at least one implementation satisfying the high-level requirements. One can show that the system architecture and the high-level requirements of the components comply with the system requirements by proving that the system requirements are satisfied by the design instantiated with any components that satisfy the high-level requirements [5].

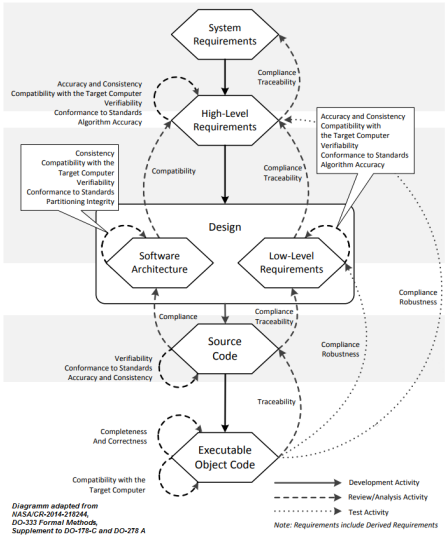

Fig. 1. Software Development Process as in DO-178C [5]

The usual challenges when using theorem proving are:

- Sound semantics of the underlying formal method?
- Degree of automation acceptable?
  - Significant expertise and user training necessary?
- Is the theorem proving tool qualified itself?

A set of 5 criteria as defined by Airbus for the use of formal methods is [41]:

- Soundness
- Cost Savings
- Analysis of unaltered programs
- Usability by normal software engineers on normal machines
- Ability to be integrated into the DO-178B conforming process

To deal with this requirements, the underlying methodology of this paper offers:

- Sound semantics from a well-estabilished stream-theory [19, 13, 34, 4, 8, 9, 3, 33]
  - Time-sensitivity, stateful modeling, nondeterminism and refinement supported
- User-friendly, yet sufficiently expressive modelling language [11]
  - Practicable for industrial application without need for theorem-proving skills
- Can be integrated into the conformance process of functional safety standards
  - Tool qualification of Isabelle [26] practicable due its axiomatic, conservative nature
- Scalability through compositional verification (allowed by commutativity and associativity of the general composition operator [19, 3])
  - Saves costs on testing (can partly replace it, and partly complement it)

The unique selling point of the stream-based formalism FOCUS [4] is:

- Refinement is fully compositional. Refinement of a subcomponent implies refinement of the overall composition.
- Properties of the old system hold directly for the new refined one aswell, thus sparing tests and integration costs after each refinement step.
- Verifying refinement of underspecification accompanies the design phase

The focus in this paper is in showing how the developed methodology handles typical development challenges in a software life cycle. The framework is thus demonstrated to deal successfully with important aspects of development of distributed systems such as underspecification, refinement and time-sensitive specifications, as well as safety and liveness properties. For this, refinement of specifications concerning streams allowed in a communication bus of the Flight Guidance System are presented. For the purpose of this paper, this is sufficient to demonstrate how theorem proving can be leveraged to spare costs in testing, achieved by enriching an encoded infrastructure sufficiently with theorems to ensure the proof for a refinement of a specification is found at the push of the button. For scaling the methodology over networks of components and a compositional verification of properties of the overall system at the push of a button, see [19] and https://www.youtube.com/watch?v=krl4Q7MAAlo for a short video demo.

In particular, the contributions of this paper consist of:

- The encoding of streams with domain-theoretical concepts in the theorem prover Isabelle
- an infrastructure for time-sensitive specifications
- a sufficient collection of functions and theorems over these structures to ensure that refinement proofs of the case study are performed at the push of a button without further user interaction
- refinement of requirements
- liveness property specification
- a case study consisting in checking refining requirements evaluate the encoded abstract theories

This paper does not go in depth in demonstrating details of our formalization of components as stream processing functions (see the previous work for a demonstration [19]) due to the limited space. This paper rather focuses in depth on one specific part of our contribution, namely in this case modeling channel histories of a distributed system, underspecification, and full automated refinement checking. This is a simple, yet sufficiently expressive example to demonstrate a run through all the development process phases when refining requirements such as those occurring in the guidance certification standards.

The rest of the paper consists in the following: The next section presents a short overview of the underlying theory. Subsequently, a formalization of a communication bus eg. in the Flight Guidance System is demonstrated, as well as the automatic refinement checking process from system requirements, to high-level requirements, low-level requirements, until the code.

2 Modular Hierarchical Methodology Providing Correctness by Design

Distributed systems in particular have proven to be more error-prone than sequential software [4, 27]. Their correct development has been a challenge in the past decades. To reduce ambiguities, formal methods have been proposed. Formal methods are known to be challenging in becoming practical in the industry due to their high costs [18]. Methodologies like CSP [15, 14], FOCUS [4], CCS [24], Petri Nets [31], or the π-calculus [25] are used to detect potential sources of errors earlier [12, 23, 1]. They support the correctness of the typical steps of development, which are usually structural decomposition and refinement. In order to spare integration costs, these two need to be compatible.

For the stepwise development of systems, the mathematical methodology FOCUS [4] is used in this work. The communication between components of a distributed system is modeled here by streams of messages flowing in unidirectional communication channels. The advantage that FOCUS has compared to the others is that refinement is indeed fully compositional. This means that one can decompose a system into components, refine those separately, and have the composed system be correct by construction. An example is shown in Figure 2.

To specify and formally verify properties of distributed systems, a part (the stream encoding) of a verification tool chain (Figure 3) is demonstrated in this

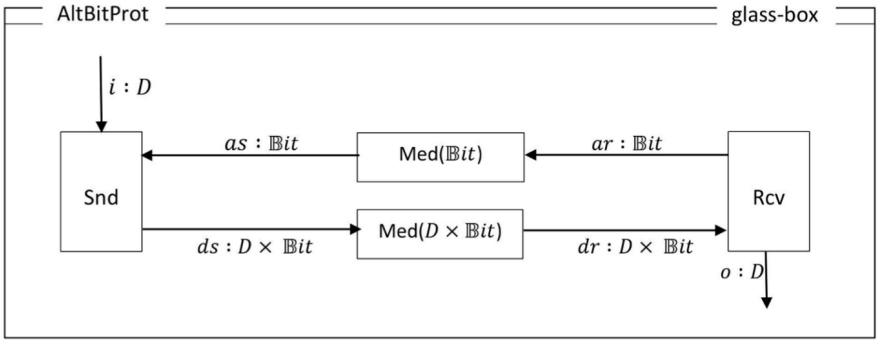

Fig. 2. Example of a Distributed System: The Alternating Bit Protocol [4]. For example the sender has an input as with messages from the alphabet $Bit := \{0, 1\}$.

paper. A developer of a distributed system is provided with an architecture description language (ADL) [11],[19] to specify in a user-friendly way the components and their interaction. This language also allows a developer to specify a desired property of the system. By the push of a button, the created system model is then transformed into an equivalent specification in the theorem prover Isabelle [26]. The property is transformed in a theorem and exterior solvers are called to prove or disprove the property. A more detailed overview is given in [19].

The ADL MontiArc [11] was created with MontiCore [16]. This language is used to describe component-and-connector architectures. The component behavior is described here by an automaton with input/output [34]. The user can specify the component interfaces, their behavior, their interaction, as well as the desired property of the system in the language OCL [32, 6, 35], which is embedded in the ADL. The ADL components are then translated into equivalent automata encoded in the theorem prover Isabelle. These automata are transformed within Isabelle into (sets of) stream processing functions, which constitute the semantics of the automata [34]. The desired property of the system written from the user in the ADL is translated into a theorem in Isabelle. Finally, general theorems over stream processing functions written in Isabelle support highly-automated property verification.

There are a couple of reasons for enforcing the specification of components by means of an ADL featuring automata, rather then giving the user just an encoding of the stream data type and set of theorems over streams in Isabelle and total freedom of specifying components over (tuples of) streams [8, 9].

First, the ADL is for a user more comfortable than writing recursive (specified typically as least fixed points [37]) stream processing functions in Isabelle or their composition [3].

Second, the automata with input and output of this methodology are designed to describe the *realizable* components (and only these) [34]. Realizability is reflected by the following properties called *monotonicity* and *continuity*.

Since streams model history, not every function is a model of a real life interactive component. After emitting a message, a component cannot take it back. Thus an extension of the input can only lead to the extension of the output. This property is called *monotonicity*. This property is also needed to guarantee the existence of least fixed points [34] to give meaning to (streams flowing in) feedback loops [33].

Also, a component cannot react to infinity, thus one cannot look at the infinite input stream to produce an output. This property is known as *continuity* [34]. This property is also needed not only to guarantee the existence, but also to calculate the least fixed point by approximation from the finite prefixes of a stream.

Allowing a user to specify any kind of function from streams to streams would also mean to expect the user to write a proof that the function is *realizable*. Since this would be not practical, a better way would be to provide the user a language (the ADL with automata of this paper) by which only realizable functions can be specified. The proof that the (sets of) stream processing functions, which are obtained from the semantics of (non-deterministic) automata, are realizable [34], is one of the important general theorems, which is encoded in the theorem prover Isabelle.

Third, the automata of this methodology are general enough to represent every realizable stream processing function [34].

Proper composition operators such as serial, parallel, feedback, and a general operator combining all these, are encoded to support a modular modeling. The composition of realizable components by these operators results in a realizable composed component [34].

Furthermore, a proper specification methodology should be able to describe underspecification. This may arise due to lack of information during the life cycle of a development process or due to non-deterministic behavior during run time. In this methodology this is reflected by non-deterministic automata, which have as semantics a set of stream processing functions.

Behavioral refinement is an important development step performed by decreasing underspecification through elimination of alternatives and thus making a specification more precise.

Refinement of the behavior of an automata is reflected by set-inclusion of the (sets of) stream processing functions representing their semantics [34]. Thus, the properties of the refined system are derived in this methodology per construction from the proven properties of the previous one, thus sparing testing and integration costs on the new system.

Refinement being fully compositional guarantees that after decomposing the system, refining the components separately, and then composing back, the system does not gain new behaviors.

A specification should also able to handle timing information. An airbag controller for instance is highly time-dependent. The specification framework presented in this work is able to represent timing. One way to model time is to extend the alphabet of messages of a stream with a dummy element $\sim$ (read: eps, see next section for a formal description in Isabelle). By assuming a discrete global clock, each time unit in a stream consists either in a message, or an $\sim$ used to model the absence of messages. Using timed automata with input and output, one can represent in this methodology the desired timed stream processing functions. The realizability properties in the timed case (such as e.g. it is invalid to look in the future of the input when producing the actual output message) are equivalently represented by the concepts of *weak causality* [34], as well as the stronger version by adding delay and thus dealing with feedback loops called *strong causality* [34].

In this methodology, a code generator from the ADL MontiArc to Isabelle generates specifications of the components, their composition, and helper theorems to support verification. A library of theorems encoded in the theorem prover Isabelle leads to high automation of the proofs.

Some of the mathematical fundaments of the concepts above have been formalized in theorem provers, such as HOLCF in [17]. The works [36, 8, 9] formalized streams in the theorem prover Isabelle with domain-theoretical concepts, and constitute a foundation of this paper. Furthermore tools such as AutoFOCUS [39] use stream-based semantics to model distributed systems. Also, the Ptolemy Project [20, 21] tackles the challenge to formalize component networks. In the Isabelle-formalization in this paper a high-level API is provided to hide fixed-point theoretical concepts from the developer.

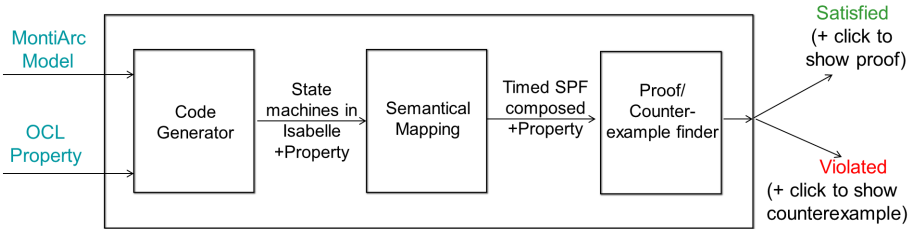

Fig. 3. Tool Chain

3 Verification of Distributed Software Development Steps

3.1 Algebra of Stream Processing Functions

FOCUS [4] and its main construct *stream* consitute the mathematical underpinning of the methodology. An (untimed) *stream* is a (potentially infinite) sequence

of messages over a set M. M^ω is the set of all streams and constitutes the union of finite streams M^* and infinite ones M^∞. Similar to lists, a constructor " : " with signature $M \Rightarrow M^\omega \Rightarrow M^\omega$ is used to create streams by appending an element to an existing stream.

To specify time-sensitive behavior, a variant of timed streams (so-called *time-synchronous streams*) is used. The message set is extended by an element $\sim$ (read *eps*). As mentioned, it is interpreted as "no messages arrived in that timeslot". A discrete global clock is assumed. An element of the stream is then a message or an $\sim$ (which has the length of one time frame as well). This way one can react to the absence of messages.

The concatenation of two streams is denoted by $\frown$. A prefix ordering $\sqsubseteq$ is defined on the set of streams to approximate infinite streams:

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

M^ω forms a complete partial order [34].

3.2 Abstract Theories in Isabelle

An important goal of the paper is to demonstrate the verification of requirements refinement. The theorem prover Isabelle [26] is a generic system for implementing formalisms in higher-order logic. Informally HOL can be described with the equation

$$\text{HOL} = \text{Functional Programming} + \text{Logic}.$$

An Isabelle-Theory consists of data type definitions, functions and proofs [19, 28]:

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

Please note that the proofs of the abstract lemmas below are omitted for space reasons, whereas the proof of the case study lemmas is fully demonstrated (and consists in applying the abstract lemmas of the theory). The logic of computable functions (HOLCF) has been encoded by Regensburger [30] and Huffman [17]. Based on this, the data type of natural numbers extended with the infinite element is encoded using the `domain` command [17]:

```
domain lnat = lnsuc (lazy lnpred::lnat)
```

lnsuc is a constructor and its inverse is lnpred. The domain command makes sure the type `lnat` is an element of the classes zero and ord by generating a bottom element and an order relation.

```
instantiation lnat :: "{ord, zero}"
begin
  definition lnzero_def: "(0::lnat) ≡ ⊥"
  definition lnless_def: "(m::lnat) < n ≡ m ⊑ n ∧ m ≠ n"
  definition lnle_def:   "(m::lnat) ≤ n ≡ m ⊑ n"
instance ..
end
```

A finite natural number `n` is represented by `Fin n`, where lntake is a function which retrieves a specified number of elements of the recursively constructed data type lnat.

```
definition Fin::"nat ⇒ lnat" where
"Fin k ≡ lntake k·∞"
```

The dot after the letter k above is an abbreviation used when a continuous function is applied [17].

Then, the infinite element is encoded:

∞ is the maximum of all `lnat`s

```
definition Inf'::"lnat" ("∞") where
"Inf' ≡ fix·lnsuc"
```

`Fix` denotes the least fixed point operator calculated as Kleene-Approximation. A couple of theorems for increasing automation follows:

Bottom element of lnat is 0:

```
lemma bot_is_0: "(⊥::lnat) = 0"
```

0 is not equal ∞:

```
lemma Inf'_neq_0[simp]: "0 ≠ ∞"
```

∞ is a fixed point of `lnsuc`:

```
lemma fold_inf[simp]: "lnsuc·∞ = ∞"
```

The brackets with content [simp] ensure that this rewriting rule is integrated in the core of Isabelle and the lemma does not need to be explicitly called by name during a proof. The preparations are done to be ready to encode the stream data type. The implementation is similar to that of lazy lists in Haskell, and apart from little technical details, looks as following, where 'a is a type parameter abstracting from the sort of messages:

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

`domain` [17] creates a new (potentially recursive) data type enhanced with an order (in this case the prefix order) and a bottom element (here the empty stream). `lsconc` is the name of the constructor that appends an element to the rest of the stream.

The data type can be extended into time-synchronous streams by using the constructor Msg with arity 1 and eps for "no data" (abbreviated as $\sim$).

```
datatype 'a tsyn = Msg 'a | ∼
```

For instance the following time-synchronous stream over natural numbers $<[Msg\ 3, \sim, Msg\ 5, \sim, \sim, ...]>$ is interpreted as: message 3 arrives in the first time slot, no message arrives in the second time slot, message 5 arrives in the third time slot, and then no message arrives anymore. Depending on the application to be modeled, the granularity (duration of a time slot) can be interpreted at will (a time slot can correspond to 1 millisecond, or perhaps to 1 minute).

Now a couple of functions, abbreviations, as well as lemmas over streams follow.

The empty stream is denoted as ϵ.

```
abbreviation sbot :: "'a stream"  ("ϵ")
where "sbot ≡ ⊥"
```

`sup'` is used to construct a stream by a single element.

```
definition sup':: "'a ⇒ 'a stream" ("↑_" [1000] 999) where
"sup' a ≡ updis a && ϵ"
```

The `updis` command above lifts an arbitrary type to a discrete pointed partial order [17].

`sdom` retrieves the set of all values in a stream and `snth` is used to get the n-th element of a stream.

```
definition sdom:: "'a stream → 'a set" where
"sdom ≡ Λ s. {snth n s | n. Fin n < #s}"
```

`slen` retrieves the length of a stream. It is defined as the number of its elements or ∞ for inifinite streams.

```
definition slen:: "'a stream → lnat" where
"slen ≡ fix·(Λ h. strictify·(Λ s. lnsuc·(h·(srt·s))))"
```

The command `strictify` above turns a function into a strict one [17].

UNIV denotes the set of all elements in a data type.

The function `slookahd` applies a function to the head of stream. If the stream is empty, $\bot$ (the polymorphic parameter 'a of streams has also an order defined in it and has also a least element) is returned. This function is especially useful for defining own stream-processing functions.

```
definition slookahd:: "'a stream → ('a ⇒ 'b) →
    ('b::pcpo)" where
"slookahd ≡ Λ s f. if s = ϵ then ⊥ else f (shd s)"
```

Hereby `shd` returns the head of a stream.

`sfilter` removes all elements from the stream which are not included in the given set.

```
definition sfilter:: "'a set ⇒'a stream →'a stream" where
"sfilter M ≡ fix·(Λ h s. slookahd·s·(λ a.
```

```
    (if (a ∈ M) then ↑a • (h·(srt·s)) else h·(srt·s))))"
```

Type classes, such as those in Haskell, are supported: `class 'a::countable` means that the type 'a is restricted to belong in the class of countable types, where there exists an injective mapping from the type 'a into the natural numbers.

```
class countable =
  assumes ex_inj: "∃to_nat :: 'a ⇒ nat. inj to_nat"
```

`sinftimes` concatenate a stream infinitely often to itself (an abbreviation in form of a suffix is introduced in brackets).

```
definition sinftimes  :: "'a stream ⇒ 'a stream" ("_∞")
    where
"sinftimes ≡ fix·(Λ h. (λs.
  if s = ϵ then ϵ else (s • (h s))))"
```

Finally, a collection of lemmas over streams to improve high automation:

Only the empty stream has length zero:

```
lemma only_empty_has_length_0 : "#s ≠ 0⟹s ≠ ϵ"
```

Filtering with a superset of the stream's domain does not change the stream:

```
lemma sfilter_sdoml3:
  "sdom·s ⊆ X⟶sfilter X·s = s"
```

A couple of connections between sfilter and sdom (notice the infix abbreviation of the `sfilter` function):

```
lemma sfilter_bot_dom: "(A ⊖ s) = ⊥⟹sdom·s ⊆ UNIV - A"

lemma sdom_sfilter1: assumes "x∈sdom·(A⊖s)"
shows "x∈A"

lemma sfilterEq2sdom_h: "sfilter A·s = s⟶sdom·s ⊆ A"

lemma sfilterEq2sdom_h: "sfilter A·s = s⟶sdom·s ⊆ A"
```

If the head of a stream is in M, then `sfilter` will not drop the head of the stream:

```
lemma sfilter_in[simp]:
"a ∈ M⟹sfilter M·(↑a • s) = ↑a • sfilter M·s"
```

If the stream is not empty, then the following holds for the length:

```
lemma srt_decrements_length : "s ≠ ϵ⟹#s =
    lnsuc·(#(srt·s))"
```

If x isn't empty then concatenating head and rest leaves the stream unchanged:

```
lemma surj_scons: "x≠ϵ⟹↑(shd x) • (srt·x) = x"
```

If filtering everything except z from the stream x doesn't produce the empty stream, then z must be an element of the domain of x:

```
lemma sfilter2dom:
  "sfilter {z}·x ≠ ϵ⟹z ∈ sdom·x"
```

Mapping a stream to head and rest is injective:

```
lemma inject_scons: "↑a • s1 = ↑b • s2⟹a = b ∧ s1 = s2"
```

If the head of a stream is in M, then `sfilter` will keep the head of the stream:

```
lemma sfilter_in[simp]:
"a ∈ M⟹sfilter M·(↑a • s) = ↑a • sfilter M·s"
```

After filtering by filter `T`, the head of the result is in `T`:

```
lemma sfilter_ne_resup: "sfilter T·s ≠ ϵ⟹shd (sfilter
   T·s) ∈ T"
```

A relevant connection between `sfilter` and `sinftimes`:

```
lemma sfilter_sinftimes_in[simp]:
"sfilter {a}·(sinftimes (↑a)) = sinftimes (↑a)"
```

Repeating a stream infinitely often is equivalent to repeating it once and then again infinitely often:

```
lemma sinftimes_unfold: "sinftimes s = s • sinftimes s"
```

Prepending a singleton stream increases the length by 1:

```
lemma slen_scons[simp]: "#(↑a•as) = lnsuc·(#as)"
```

For nonempty `s`, `sinftimes s` is infinite:

```
lemma slen_sinftimes: "s ≠ ϵ⟹#(sinftimes s) = ∞"
```

Finally, infinitely cycling the empty stream produces the empty stream again:

```
lemma strict_icycle[simp]: "sinftimes ϵ = ϵ"
```

The neccessary structures were introduced, so that the case study can now be specified and verified.

3.3 Case Study in Isabelle: Automatic Checking of Requirements Refinement

A bus in the Flight Guidance System is modeled and a specification indicates which streams are allowed to flow in it. The system requirement in this case is a set of streams fulfilling a fairness property, which is required to guarantee that a communication protocol using this bus works correctly (the protocol is not further specified, but such fairness requirements are not unusual; notice for example that the Alternating Bit Protocol [4] is only correct under the assumption of a fair medium).

By the following system requirement (SysReq), the only stream histories of messages allowed to flow in the bus are only those such that, after waiting an infinite amount of time, the number of actual proper messages occurring in the bus is infinite (thus the "no-data" symbol eps can occur only a finite amount of time).

```
definition SysReq :: "(nat tsyn stream) set" where
"SysReq ≡ {s. #((UNIV-{∼})⊖s) = ∞}"
```

To guarantee the SysReq, the developed high level requirements can look as follows: The length of the stream should be infinite and every element of the stream is an actual message, starting from the second one. The data type can be abstracted to any countable one (typical for high level requirements; fixed only after reaching low level requirements), in the sense that, if a property holds for any arbitrary countable type, then it also holds on natural numbers of (as wished in SysReq). The decision, whether the first element is a proper message, or an eps, is postponed for a point in time where the overall architecture of the system is known. If this bus is eg. to be embedded in connection with a feedback loop, then the first element being an "no-data" can act as a delay of 1 unit and thus make sure that the semantics of the stream flowing in the feedback loop is uniquely defined. If not, then this "delay" shall not be needed.

```
definition HLR :: "(('a::countable) tsyn stream) set"
    where
"HLR ≡ {s. #s=∞ ∧ sdom·(srt·s) ⊆ UNIV-{∼}}"
```

The refinement checking is formulated as a subset relation between HLR and SysReq and the encoding of sufficient abstract theorems ensures that the proof is found at the push of a button without the need of user interaction:

```
lemma "HLR ⊆ SysReq"
  by (smt HLR_def SysReq_def Collect_mono mem_Collect_eq
      Inf'_neq_0 fold_inf lnat.sel_rews(2)
       sfilter_sdoml3 sfilter_bot_dom sfilter_in
           sfilter_nin slen_empty_eq srt_decrements_length
       surj_scons sfilter2dom sdom_def)
```

Next, low level requirements shall be specified, namely the underspecification is reduced eg. after determining that there is no feedback loops in the architec-

ture, and thus no necessity for introducing a delay. So the first message is fixed, leading to LLR (yet a deterministic implementation is not reached yet):

```
definition LLR :: "(nat tsyn stream) set" where
"LLR = {s. #s=∞ ∧ sdom·s ⊆ UNIV-{∼}}"
```

The correctness of the refinement is checked again fully automatically:

```
lemma "LLR ⊆ HLR"
  by (smt HLR_def LLR_def DiffD2 sdom_sfilter1
      sfilter_sdoml3 sfilter_srtdwl3 singletonI surj_scons
  Inf'_neq_0 inject_scons sfilterEq2sdom_h sfilter_in
      sfilter_ne_resup slen_empty_eq Collect_mono)
```

An interesting question concerning requirements is usually whether they are consistent, i.e. is there at least an implementation fulfilling these?

Finally, an implementation is chosen, namely the infinite stream consisting of only the number one:

```
definition Code :: "nat tsyn stream" where
"Code = ↑(Msg 1)∞"
```

The consistency of LLR is shown fully automatically by proving that the Code-implementation above is an element of the LLR set.

```
lemma "Code ∈ LLR"
by (smt LLR_def Code_def Diff_UNIV Diff_empty
    Diff_eq_empty_iff bot_is_0 insert_iff tsyn.distinct(1)
    mem_Collect_eq only_empty_has_length_0 sfilterEq2sdom
        sfilter_sinftimes_in sinftimes_unfold
    slen_scons slen_sinftimes strict_icycle
        subset_Diff_insert subset_singleton_iff
        lnat.con_rews)
```

In conclusion, theorem proving can be leveraged to spare costs for certification activities by enriching the corresponding encodings with a large number of general abstract theorems. For scaling up to networks of components, a code generator can help by generating helpful theorems about the model and their proof. The code generator mapping a modeling language into a formal language generally needs to be qualified. On the other hand, the qualification of the generated proofs does not pose a threat though, since the proofs will finally be checked by the theorem prover Isabelle and the qualification of Isabelle is not hard due to its axiomatic and conservative nature.

References

1. Akroun, L., Salaün, G.: Automated verification of automata communicating via fifo and bag buffers. Formal Methods in System Design **52**(3), 260–276 (2018)
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 (Jan 2018), https://hal.archives-ouvertes.fr/hal-01708332
3. Broy, M., Rumpe, B.: Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. Informatik Spektrum **30**(1), 3–18 (2007)
4. Broy, M., Stølen, K.: Specification and Development of Interactive Systems. Focus on Streams, Interfaces and Refinement. Springer Verlag Heidelberg (2001)
5. Cofer, D.D., Miller, S.P.: Do-333 certification case studies. In: NASA Formal Methods (2014)
6. Cook, S., Kleppe, A., Mitchell, R., Rumpe, B., Warmer, J., Wills, A.C.: The Amsterdam Manifesto on OCL. In: Object Modeling with the OCL, pp. 115–149, LNCS 2263. Springer Verlag, Berlin (2002)
7. Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., Védrine, F.: Towards an industrial use of fluctuat on safety-critical avionics software. In: International Workshop on Formal Methods for Industrial Critical Systems. pp. 53–69. Springer (2009)
8. Gajanovic, B., Rumpe, B.: Isabelle/HOL-Umsetzung strombasierter Definitionen zur Verifikation von verteilten, asynchron kommunizierenden Systemen. Informatik-Bericht 2006-03, Technische Universität Braunschweig, Carl-Friedrich-Gauss-Fakultät für Mathematik und Informatik (2006)
9. Gajanovic, B., Rumpe, B.: Alice: An advanced logic for interactive component engineering. In: 4th International Verification Workshop (Verify'07). Bremen (2007)
10. Gigante, G., Pascarella, D.: Formal methods in avionic software certification: The do-178c perspective. In: Margaria, T., Steffen, B. (eds.) Leveraging Applications of Formal Methods, Verification and Validation. Applications and Case Studies. pp. 205–215. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
11. Haber, A., Ringert, J.O., Rumpe, B.: MontiArc - Architectural modeling of interactive distributed and cyber-physical systems, Technical report / Department of Computer Science, RWTH Aachen, vol. 2012,3. RWTH and Technische Informationsbibliothek u. Universitätsbibliothek and Niedersächische Staats- und Universitätsbibliothek, Aachen and Hannover and Göttingen (2012)
12. Hall, A.: Seven myths of formal methods. IEEE Software **7**(5), 11–19 (Sep 1990). https://doi.org/10.1109/52.57887
13. Harel, D., Rumpe, B.: Meaningful modeling: what's the semantics of" semantics"? Computer **37**(10), 64–72 (2004)
14. Heim, R., Nazari, P.M.S., Ringert, J.O., Rumpe, B., Wortmann, A.: Modeling robot and world interfaces for reusable tasks. In: Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on. pp. 1793–1798. IEEE (2015)
15. Hoare, C.A.R.: Communicating sequential processes. In: The origin of concurrent programming, pp. 413–443. Springer (1978)
16. Hölldobler, K., Rumpe, B.: MontiCore 5 Language Workbench Edition 2017. Aachener Informatik-Berichte, Software Engineering, Band 32, Shaker Verlag (December 2017)
17. Huffman, B.C.: HOLCF '11: A definitional domain theory for verifying functional programs. Portland State University, [Portland, Or.] (2012)
18. Kasauli, R., Knauss, E., Kanagwa, B., Nilsson, A., Calikli, G.: Safety-critical systems and agile development: A mapping study. pp. 470–477 (08 2018)
19. 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-WS.org (February 2019)
20. Lee, E.A.: Computing needs time. Communications of the ACM **52**(5), 70–79 (May 2009)
21. Lee, E.A.: Fundamental limits of cyber-physical systems modeling. ACM Transactions on Cyber-Physical Systems **1**(1) (11 2016), http://chess.eecs.berkeley.edu/pubs/1183.html
22. Li, Y., Tan, T., Xue, J.: Effective soundness-guided reflection analysis. In: Blazy, S., Jensen, T. (eds.) Static Analysis. pp. 162–180. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
23. Maoz, S., Pomerantz, N., Ringert, J.O., Shalom, R.: Why is my component and connector views specification unsatisfiable? pp. 134–144 (2017)
24. Milner, R.: Communication and concurrency, vol. 84. Prentice hall New York etc. (1989)
25. Milner, R.: Communicating and mobile systems: the pi calculus. Cambridge university press (1999)
26. 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 [etc.] (2002)
27. Olveczky, P.C.: Designing Reliable Distributed Systems. Springer-Verlag London (2017)
28. Paulson, T.N.L.C., Wenzel, M.: A proof assistant for higher-order logic (2013)
29. Payet, E., Spoto, F.: Checking Array Bounds by Abstract Interpretation and Symbolic Expressions, pp. 706–722 (06 2018)
30. Regensburger, F.: HOLCF: Eine konservative Erweiterung von HOL um LCF. na (1994)
31. Reisig, W.: Petri nets: an introduction, vol. 4. Springer Science & Business Media (2012)
32. Richters, M., Gogolla, M.: On formalizing the uml object constraint language ocl. In: International Conference on Conceptual Modeling. pp. 449–464. Springer (1998)
33. Ringert, J.O., Rumpe, B.: A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing. International Journal of Software and Informatics **5**(1-2), 29–53 (July 2011)
34. Rumpe, B.: Formale Methodik des Entwurfs verteilter objektorientierter Systeme. Doktorarbeit, Technische Universität München (1996)
35. Rumpe, B.: Modellierung mit UML, vol. 2nd Edition. Springer (2011)
36. Spichkova, M.: Specification and Seamless Verification of Embedded Real-Time Systems: FOCUS on Isabelle. VDM Verlag Dr. Müller Aktiengesellschaft & Co. KG, Saarbrücken (2008)
37. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific journal of Mathematics **5**(2), 285–309 (1955)
38. Urban, C., Ueltschi, S., Müller, P.: Abstract interpretation of ctl properties. In: Podelski, A. (ed.) Static Analysis. pp. 402–422. Springer International Publishing, Cham (2018)
39. Voss, S., Zverlov, S.: Design space exploration in autofocus3 - an overview. In: Mařík, V., Lastra, J.M., Skobelev, P. (eds.) IFIP First International Workshop on Design Space Exploration of Cyber-Physical Systems. Springer (2014)
40. Wagner, L.: Formal methods for certification: Why and how? Safe & Secure Systems and Software Symposium (S5) URL: http://www.mys5.org/Proceedings/2016/Day_3/2016-S5-Day3_1505_Wagner.pdf (2016), accessed on 19.07.2019

41. Wiels, V.: Formal methods in aerospace: Constraints, assets and challenges. URL: https://richmodels.epfl.ch/_media/madrid13-slides-virginie-wiels.pdf, accessed on 19.07.2019
42. Yuki, T., Feautrier, P., Rajopadhye, S., Saraswat, V.: Array dataflow analysis for polyhedral x10 programs. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 23–34. PPoPP '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2442516.2442520, http://doi.acm.org/10.1145/2442516.2442520