An Approach for Logic-based Knowledge Representation and Automated Reasoning over Underspecification and Refinement in Safety-Critical Cyber-Physical Systems

Hendrik Kausch
RWTH Aachen University
Aachen, Germany

Mathias Pfeiffer
RWTH Aachen University
Aachen, Germany

Deni Raco
RWTH Aachen University
Aachen, Germany

Bernhard Rumpe
RWTH Aachen University
Aachen, Germany

***Abstract*—In this paper the extension of an intelligent compositional verification framework for cyber-physical systems is presented and the capabilities of accompanying underspecification-refinement steps by verification are demonstrated on a representative example of a flight guidance system. Formal knowledge representation using higher-order logic and intelligent reasoning is shown to be applied to software engineering problems to perform correctness proofs, execute symbolic tests or find counterexamples. The theorem prover Isabelle is a mature and fundamental tool, which allows to represent knowledge as a collection of definitions and theorems and reason about systems. To increase the usability, an architecture description language (ADL) coupled with a code generator from the ADL to Isabelle is used. These and the rapid increase of computation capabilities suggest that a prominent application for reducing certification costs of critical systems such as intelligent flight control systems or assistance systems for air or road traffic management is not far in the future.**

I. Introduction

The complexity of safety-critical systems is increasing in the avionics and other fields and new technologies like unmanned flying vehicles are rapidly introduced into the market. These bring new challenges to certification processes. While trying to maintain high system reliability, the scalability of traditional quality assurance methods such as testing and reviewing is not ideal and causes considerable amount of costs [1].

If requirements were to be specified in a formal way, one could reason about them and thereby replace or complement many tests (please see also the conclusion for a more detailed discussion). Abstract Interpretation [2], [3] is a static analysis technique which has been applied in avionics. It does acts rather on code-level (in contrast to the approach of this paper, which handles requirements on all abstraction levels.) While having good automation, Abstract Interpretation is usually targeted on very specific artifacts and usually requires some manual expertise to discharge false positives. Model Checking [4] also has been applied to small and middle sized systems. But it does suffer from the well-known state-explosive problem when trying to handle larger systems. While hardware is better handled, the complexity of verifying software increases exponentially with the size of the (state space of the) system (while e.g. in theorem proving this growth is rather linear [5]).

Meanwhile, artificial intelligence fields such as logic and knowledge representation have also been applied successfully in software engineering. When representing knowledge by logic, theorem proving is then reduced to intelligent reasoning. So by creating a knowledge base for safety-critical systems, automatic reasoning becomes possible by reducing verification into applying AI techniques such as metaheuristic (proof-) search techniques. Same holds for error detection; a counterexample-finder takes as input a system model, (the negation of) a property, and error detection is reduced into a search problem (and in case of an error the trace-path of this search is returned as a malicious input). In a time of increasing computational power, these techniques open up possibilities to maintain manageable certification costs.

Common ADLs used for model-based development avionics are AADL [6], SysML [7], Simulink [8] etc. SCADE [9] (based on the dataflow language Lustre [10]) is used by Airbus [4] coupled with a model checker. In comparison, our ADL presented in this paper is extended to be able to handle not only SCADE-like time-sensitive deterministic systems (such as our previous automotive case study in [11]), but also (higher-level predicative-oriented and lower-level state-oriented) underspecification and refinement.

For the incorporation of analysis tools in their life-cycle processes, companies such as Airbus and Dassault Avionics have identified a number of requirements [9], [12]:

- Soundness
- Ability to be integrated into the certification standards conforming process
- Cost Savings
- Deliver correctness by construction
- Scalability: Compositional Verification
- Expressivity of Specification Language
- Timing aspects and underspecification refinement
- Usability by normal software engineers on normal machines

The Chair for Software Engineering at the RWTH Aachen University has been conducting research for many years using its language workbench MontiCore [13] for developing domain-specific languages (DSLs) and also performing model transformations. A model transformation could be for example a refinement [14] of an underspecified model into a more specified one in the same modeling language, or a code generator mapping a model from a domain-specific front end language into an equivalent model in an analysis tool (e.g. a model checker or a theorem prover).

In principle, for any DSL one can create a knowledge base and reason about it in a logic language. In this paper this is demonstrated on the example of an architecture description language (ADL) for cyber-physical systems (called MontiArc [11], [15]) coupled with a code generator from the ADL into the knowledge base (here theorems in Isabelle). The added value demonstrated in this paper is accompanying step-wise refinements during the design-lifecycle of safety-critical systems by verification to reduce certification costs. The knowledge base consists in the encoding of generic dataflow-oriented data structures, functions and theorems for reasoning in the theorem prover Isabelle [16]. The semantical [17] underpinning of MontiArc is FOCUS [18], a dataflow-based methodology for the stepwise development and refinement of systems. If the behavior of a component is described by a MontiArc automata with input/output, it is implementable (also called realizable) per construction. For further discussions about foundations of our methodology, please read also [11]. The semantics of (non-deterministic) components are (sets of) stream processing functions [14]. The unique selling point of FOCUS compared with other concurrents such as π-calculus [19], CSP [20] or Petri nets [21] is that refinement is fully compositional. Thus, one can decompose a system, refine each component separately e.g. until an implementation, and then be sure that after composing back the new system will be a refinement of the old one, thus sparing associated testing and integration costs. It also allows leveraging proof reuse (as will be shown in the running example section II-G or section II-I).

The contribution of this work is the extension of logic-based a knowledge representation for an automatic reasoning approach aiming at reducing certification costs of safety-critical systems. This is demonstrated through accompanying the refinement of different abstraction levels of underspecification (see fig. 1) by verification. The full compositionality of refinement of FOCUS is leveraged for the step-wise refinement of a representative example of a pilot flying system (in this cyber-physical system the bus component is hardware and the flight guidance component is software).

On a more detailed level, this includes:

- Extending an ADL and its knowledge base for enabling higher-level history-oriented requirement specification.
- Extending an ADL and its knowledge base for enabling refinement of history-oriented requirement specifications into lower-level (closer to implementation, yet still not necessarily deterministic) state-based requirement specifications.

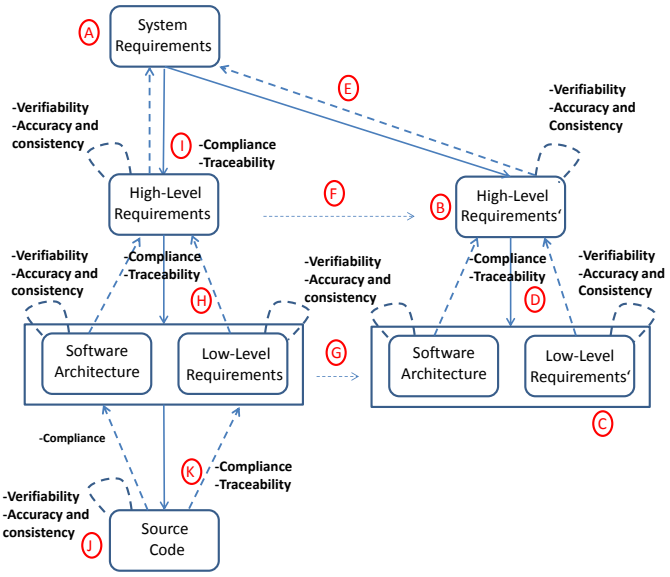

Fig. 1. Certification Activities (Letters = Sections in Chapter 2). The denoted activities are shown to be handled by our methodology and thus save a lot of test and review costs.

- Extending an ADL and its knowledge base for enabling refinement of lower-level non-deterministic state-oriented requirement specifications into another state-oriented specification by means of transition-refinement and state refinement.
- Extending counterexample-finding capability for design error detection.
- Optimizing signatures of key structures in the knowledge base to increase the automation degree and reduce the needed user-expertise (consisted in rewriting the encoding with total functions instead of partial functions (such as in [22]), thus making automatic proof finding much easier)
- Integration of an abstraction mechanism called *locals*, which improved the efficiency of the code generator about 10 times and enables thus an easier-qualifiable code generator from the ALD to the theorem prover.
- Discussing at what extend such a knowledge representation approach can scale to meet industrial requirements, documented in a table in the conclusion after the evaluation and consideration of the lessons learned.
- Providing an Integrated Verification Environment created with Visual Studio Code for the integration of all the artifacts of the framework (see Appendix).

The rest of the papers consists in the following: The next chapter describes a typical avionics software development activity, where step-wise refinements of underspecification accompanied by verification is presented on the example of a flight guidance system (adapted from a collaboration between NASA and Rockwell Collins [23], we handle the more involved asynchronous case).

II. Running Example

General Approach

In this section we present the development of a pilot flying system (PFS) with two flight guidance systems (FGSs) in a refinable, decomposable fashion, adapted from [23]. While in [23] different levels of abstractions are handled with different techniques (theorem proving, model checking, abstract interpretation), we demonstrate to be able to handle all abstraction levels by our framework. The red-colored letters (fig. 1 extracted from certification activities of DO-178C) correspond to the sections of the next chapter. During the formalization of informal requirements, errors in a first version of the higher-level requirements are detected (red-letters B, C), and then repaired (letters I, H). The system to be developed is depicted in fig. 3. The tool chain is described in fig. 2. The components of the system transmit potentially infinite sequences of messages called streams over its channels [18]. Time-synchronous streams are used to model time-sensitive behavior. As described previously in [11], this can be realized through extending the message alphabet by $\sim$ to denote that no data is transmitted in a time slot. Timing granularity can be set at wish depending on the system to be modeled (e.g. in the case of the flight guidance system below, one time unit represents 1 millisecond).

The semantics of components are sets of stream processing functions mapping input channel histories to output channel histories and are hierarchically decomposable [11].

One-directional channels connect components by transmitting input and output messages. In the PFS, messages sent between the FGSs are transmitted by hardware buses. Each FGS outputs a pair of booleans where the first boolean denotes the liveness of the system and the second boolean is used as an acknowledgement bit. Clocks restrict the other components behavior by sometimes forcing them to output the last message and not reacting to its input in any way. The system is visualized in fig. 3.

For each component (hardware and software), HLRs and LLRs were created. When verifying compliance (say between HLR and SysReqs), the HLRs of all components in the software architecture are proven to fulfill all SysReqs. We demonstrate to handle in the following a few representative certification activities. In our knowledge representation approach, certification credits concerning "'verifiability"' of requirements (see fig. 1) are generally claimed by having used a formal language for the specification, and "'accuracy"' is claimed by having formal proofs.

A. System Requirements (SysReqs)

The system requirements (SysReqs) are largely adapted from [23]. By formulating them as Object Constraint Language (OCL) (see [11], [24]) constraints, these can be inputted to the tool chain together with the MontiArc system model. It is then checked whether the system model fulfills the requirements. The first 5 SysReqs are similar to the 5 SysReqs from [23] (see also Appendix). We give the first SysReq in natural language.

SysReq1: At least one side is the active pilot flying side.

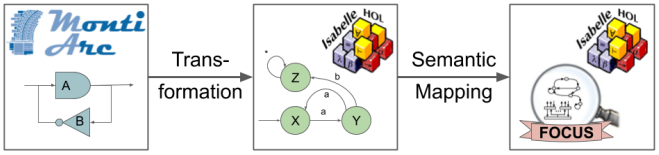

Fig. 2. Verification tool chain comprised of two stages: MontiArc frontend and Isabelle backend. Specifications are modeled in MontiArc, transformed to equivalent Isabelle representations, then semantically mapped to Stream Processing Functions and their properties are checked highly automatically.

Below is the first SysReq also as an OCL expression. It is formulated in the context of the general component structure from fig. 3 and addresses the streams by their port name:

```
fst c1[n] or fst c3[n]
```

The OCL expression consists of elements known from first-order logic. Ports are named after the connected channels from fig. 3. A syntactic extension allows to obtain the nth element (point in time) of the stream flowing in the channel $c1$ as $c1[n]$. The translation as Isabelle theorem looks alike (variables are per default universally quantified). The first item of a tuple is returned by fst.

Furthermore, we define a safety-critical sixth SysReq informally.

SysReq6: Pressing and holding the transfer switch c5 for 10 milliseconds always switches the inactive flying side, if the inactive component received the last transfer switch input and the system is in a stable state (see also section C for the architecturial contex).

B. High-Level Requirements' (HLR')

The engineer now tries to develop high-level requirements for each sub-component, so that the overall system requirements can be fulfilled. First, we introduce the high-level requirements of the clocks (corresponding in fig. 1 to HLR'):

```
component ClockUnfairSpec {
  timing sync;

  port
    out boolean clk1;

  spec {
    // outputs an inf. long bool stream
    clk1.length() = INF
  }
}
```

The bits on the clocks output stream determine whether its associated component is active (executing) or not. The ADL MontiArc is used to define the high-level requirements (HLRs)of the clock component. This is a component with one output and no input channel. First, the timing of the clock is defined. The keyword $sync$ indicates weak-causal and $causalsync$ strong-causal time-synchronous components (causality captures realizability in time sensitive modeling,

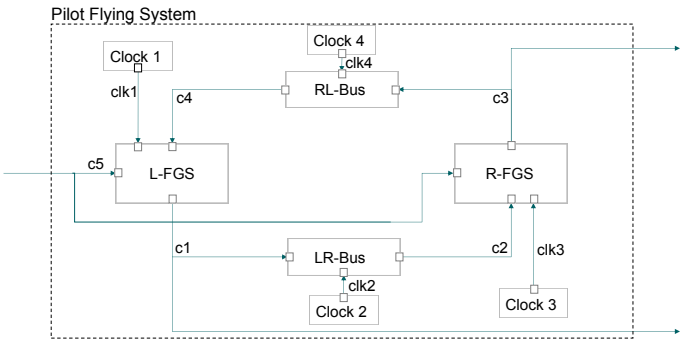

Fig. 3. An architecture of the pilot flying system (PFS), sufficient for correctly synchronizing the pilot flying sides and the other mentioned system requirements. `clk1` etc. denote the channels that have clocks as sources, `c1` etc. refer to the other communication channels, `L-FGS` denotes the left flight guidance system, `RL-Bus` denotes the bus from right to left.

[11]). Weak-causality enforces that the production of an output at a point in time t does not depend from an input arriving after the point in time t. Strong-causal components have to define an initial output and delay interaction by 1 time unit. They are needed for well-defined semantics in feedback loops. The behavior is specified as a high-level predicate (*spec* stands for specification). Inside the *spec* section, requirements to the boolean output stream clk can be formulated as OCL expressions. The $length$ function obtains the length of a stream and INF represents infinity.

The HLRs for the Buses and FGSs are defined similarly. This paper lists only the LLR' as supplementary material in fig. 7.

C. Software Architecture and Lower-Level Requirements' (LLR')

In fig. 3 each component is represented by a named box. The channels describe the internal interaction between the components. The incoming and outgoind channels are the global in- and outputs of the PFS. Small boxes represent the ports of a component. Our scalable vector graphics (SVG) generator draws figures based on MontiArc models. Such visualization is a powerful tool to gain overview and improve the communication, especially in the early stages of the development process.

Here, each of the 4 clocks is connected to one of the non-clock components to model their non-deterministic behavior. On a "'False"' clock output signal, each non-clock component does not react to its input and simply repeats the last output. Input channel $c5$ transmits the transfer switch status, $c1$ is the output of the L-FGS system and $c4$ its input from the RL-Bus etc.

The complete system can be decomposed into sub-components hierarchically [15]. After the translation into Isabelle code, the composition operator $\otimes$ connects the in- and output channels of two components by its names (for some technical details concerning the knowledge base see [25]). The following composition correctly defines the system model because the composition operator is commutative and associative (thus making the composition order irrelevant) [26].

$$FGS_L \otimes FGS_R \otimes Bus_{LR} \otimes Bus_{RL} \otimes clk_1 \otimes clk_2 \otimes clk_3 \otimes clk_4$$

After defining the Pilot Flying System as a composition of its sub-components, we define the low-level requirements of the sub-components. The unfair clock has to output non-deterministicially true or false (input and transition guards can be modeled in MontiArc [15], but are not present in this component). A graphical representation of the automata is given in fig. 4 and the textual version is:

```
component ClockUnfairAutomata {
  timing sync;

  port
    out boolean clk1;

  automaton {
    // one state
    state Single;
    initial Single;

    // outputs true or false in every step
    Single -> Single / {clk1=true};
    Single -> Single / {clk1=false};
  }
}
```

Timing and interface of the component are defined similar to section II-B. The automata then specifies the behavior in a lower-level, yet still non-deterministic fashion. States are defined in the first line of the automata body. The second line sets the initial state of the automata. Following the state specifications, the automata transitions and the output and state-change behavior are defined on a step-by-step basis.

D. Compliance of LLR' to HLR' and HLR' Consistency

The consistency of a specification is defined as the existence of at least one implementation that satisfies the specification. The consistency of our HLR' can be shown as follows: First prove the compliance of LLR' to HLR' (i.e., LLR' refines HLR'). Then show that an implementation of LLR' exists. For the later, consider that LLR' (e.g. of the clocks) are non-determinsitic (total) automata, and can easily be converted to an implementation (deterministic automata) by removing transitions responsible for non-determinism. Next, we focus on compliance of LLR' to HLR'.

For this, the LLR' MontiArc automata is translated to an Isabelle automata and further transformed to (sets of) stream processing functions. The HLR' spec is directly transformed to (a set of) stream processing functions. The transformation from automata to stream processing function (SPF) is defined as a (greatest) fixed point calculation of the corresponding functional [14]. Now, a theorem can be formulated in OCL by the user, stating: LLR' **refines** HLR'. This is translated into

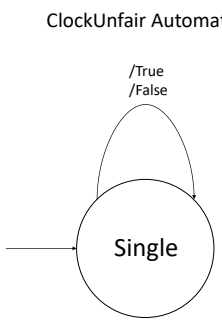

Fig. 4. The unfair clock outputs non-deterministically booleans

the following Isabelle theorem, where the Isabelle-semantics of a MontiArc construct is abbreviated by ⟦...⟧):

theorem
"⟦ClockUnfairAutomata⟧⊆⟦ClockUnfairSpec⟧"

So refinement is reduced to a set-inclusion of SPFs. This theorem is proven by showing that every SPF from ⟦ClockUnfairAutomata⟧ outputs a bool stream that is infinitely long, thus satisfying the OCL requirement in the spec-behavior of ClockUnfairSpec. All other components consistencies are proven similarly and follow the overall tactic to show the fulfillment of all specification properties. Since the other components are deterministic, their semantics contains exactly one SPF. In order to increase automation of reasoning over (sets of) stream processing functions, abstract (case-study-independent) theorems have been proven in the knowledge base [25].

It might be interesting to note that compared to [11], the largest part of the generator has been moved inside Isabelle. This keeps the inter-language transformation from architecture description language (ADL) to Isabelle minimal (about 10 times less compared to [11]). This was achieved by integrating the concept of locales [27]. The whole tool chain becomes thus easier qualifiable due to the axiomatic and conservative nature of Isabelle (please note that the translation from a DSL into a logical language needs in general to be qualified, and is usually performed by testing it for a representative collections of models).

E. HLR' Compliance with SysReq

The next important step is to check the fulfillment of the system requirements. HLR' fulfills the first 5 SysReqs, but not the 6th. The proofs that HLRs comply with the 5 SysReqs is generally history-oriented (reasoning over infinite streams). Meanwhile, tools like quickcheck and nitpick were integrated in our tool chain and can be used to automatically search for counterexamples. Quickcheck originates from Haskell and is a test-based tool that needs an executable implementation from which it generates Haskell code and is fast [28]. Nitpick, on the other hand, is a SAT-solving based tool that is able to find even infinitely long stream-counterexamples, and also reason over not-necessarily-executable specifications [29]. A counterexample for the 6th SysReq would be e.g. the following streams on channels $c5$ and $clk3$:

$$stream_{c5} = False^1 \bullet True^{11} \bullet False^\infty$$
$$stream_{clk3} = True^1 \bullet False^{11} \bullet True^\infty$$

The $\bullet$ infix is used for stream concatenation, whereas i-fold repetition is formalized as $\_^i$. The stream $True^\infty$ for example is an infinitely long stream of messages $True$. Obviously, the assignment does not contradict the HLR' of the clock (random $true$s and $false$s). But the system does not change the flying side after pressing the button for 10 milliseconds (SysReq 6), as the clock blocks any reaction for 11 time-units. Thus, a new and sufficient HLR has to be formulated.

The infrastructure for finding counterexamples in our tool chain has been extended by identifying and inserting (done as described e.g. in [28]) fitting lemmas into the transformed Isabelle encoding. These lemmas are essential not only to counterexample-finders, but also facilitate automatic proofs and general transparency of the encoding.

F. Refinement of HLR' to HLR

As one may notice, one does not have to come up with an entirely new HLR. With only a slight refinement of HLR', the 6th SysReq can be fulfilled. The new clock specification (HLR) is obtained by adding the following OCL-predicate into the spec-body of the ClockUnfairSpec:

```
forall n in nat.
  exists m in nat.
    n < m && m < n+10 && clk[m];
```

The new HLR restricts the clocks to output at least one $true$ every 10 milliseconds. The new specification is called $fair$ clock and also translated to Isabelle. After generating the corresponding set of stream processing functions for HLR, "HLR refines HLR'" is reduced to a set-inclusion proof.

theorem
"⟦ClockFairSpec⟧⊆⟦ClockUnfairSpec⟧"

The signatures of the involved functions are identical, and the predicate of HLR is a slight sharpening of the predicate of HLR' (by the added OCL-requirement in section II-F), and thus the refinement proof is easily found automatically. Since all SPFs from ClockFairSpec output an infinite stream of booleans, the only requirement of the ClockUnfairSpec is already met.

G. Refinement of LLR' to LLR

Similar to above, one does not need to come up with an entirely new LLR (for the clocks) out of their HLR, but can instead refine the almost-successful LLR' just enough to comply with the HLR. A new LLR for the fair clock is defined as a non-deterministic automata (see fig. 5) using a finite timer to force at least one $true$ every 10 milliseconds:

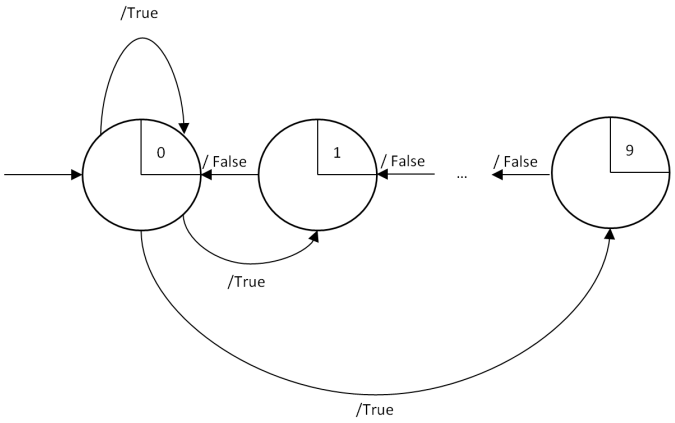

Fig. 5. Fair clock with counter

One can save certification costs (of proving compliance of LLR with HLR and SysReq all over again) by proving that LLR is a refinement of LLR', and then reasoning that the already proven properties (LLR' refines HLR' and HLR' fulfills almost all SysReq) will also hold for LLR. This means one only needs to prove the missing 6th SysReq.

The refinement of LLR' into LLR is a refinement between non-deterministic automata. Similar to the refinement between HLR and HLR', a theorem in Isabelle can be formulated over the automata's semantics:

theorem
"⟦ClockFairAutomata⟧ ⊆ ⟦ClockUnfairAutomata⟧"

This is proven by two simple refinement steps. The first one is a semantics-preserving state-refinement (through splitting the one state of LLR' into 10 by introducing a counter variable. The set of behaviors remains after this step the same (does not become strictly smaller yet) [14] (section 5.3.5 and 6.4.3). The state refinement is usally a prelude for subsequent underspecification-reducing refinement steps, such as the following. The second and final step is a transition-refinement [14] (section 6.3.1) by which transitions (responsible for non-determinism), which lead to "unfair" behavior (infinite consecutive inactivity clock-signals) are removed.

H. Compliance of LLR to HLR and HLR Consistency

As LLR' refines HLR' (section II-D) and LLR refines LLR' (section II-G), it follows that LLR refines HLR'. Now all that is left to prove is that LLR satisfies the additional constraints of HLR (compared to HLR'). This reduces certification costs for the HLR consistency proof significantly. The property that the state with $counter = 0$ is reached after maximal 9 steps (visible in fig. 5) is sufficient to prove the additional HLR property easily automatically.

I. HLR compliance with SysReq and Traceability of HLR

As HLR' is already proven to satisfy all but one SysReq and HLR is a further refinement of HLR', we can directly conclude that HLR also satisfies the first 5 SysReq. We thus focus on proving the remaining 6th SysReq. This reduces certification costs of HLR compliance.

Compliance: The compliance with the last SysReq follows straightforwardly (proof is thus easily found automatically) from the added OCL-predicate in the spec-body of HLR in section II-F).

Traceability: Certification credits for the traceability of HLR is claimed by deleting HLR-predicates (OCL-expressions from the spec body) one-by-one and proving that each deletion in isolation already leads to an incompliant system requirement (one of these cases led to the above mentioned HLR').

J. Source Code

The non-deterministic clocks can be refined to a deterministic behavior by deleting transitions that offer non-deterministic choices (e.g. by deleting one of the transitions such as those occurring in `ClockUnfairAutomata`). Apart from the non-deterministic clocks, any LLR of other components is a deterministic MontiArc model and can be interpreted as an implementation, since its semantic is a single stream processing function [14].

K. Compliance of Source Code to LLR

After translating the deterministic components (representing the source code) to Isabelle, the compliance of source code with LLR can be proven. The proof is reduced to showing that the resulting stream processing function (i.e. the semantics of the deterministic automata) is an element of the LLR semantics (a set of SPFs). Accuracy and consistency are correct per construction in the case of a deterministic component. The compliance of the source code to the software architecture and to the system requirements **holds without further effort** by the unique property of FOCUS, that refinement is fully compositional. Since the other (non-clock) components already were deterministic, their compliance with their LLRs is proven directly.

III. Conclusion

In conclusion, the methodology in this paper demonstrated handling representative scenarios for a verified development and compositional refinement of safety-critical systems. A developer can specify either directly using a logic language (e.g. Isabelle), or using an ADL for distributed systems as frontend to describe interfaces of the components, their behavior and their interaction in a comfortable way. Then the system model and all desired properties can be then translated into an equivalent specification in a knowledge base created in a logic language.

The developed knowledge base for MontiArc is very general and can be largely reused for creating knowledge bases for other modeling languages (such as AADL, SysML, SCADE, Simulink etc.) Keeping FOCUS as semantical underpinning has the already mentioned advantage that refinement is fully compositional.

This kind of approach can replace a lot of tests and reviews. This helps also with requirements involving always/never,

which cannot be exhaustively verified in general by tests. Please note though that a certain group of tests and reviews can only be complemented hereby, but not completely replaced, for instance checking whether:

- requirements formalization is correct (in general compliance between informal and formal models),
- the methodology is justified and appropriate,
- requirements and software architecture are compatible with the target computer (unless the target environment is formally modeled),
- a requirement has not been forgotten,
- there is no unidentified dead or deactivated code.

As can be seen in fig. 1, a lot of development and certification costs can be saved by omitting coming up with Source Code' out of LLR', since it would not be correct anyway, because HLR' are not good enough. One could have even spared the effort of creating LLR' by the following reasoning: HLR' has to fulfill SysReqs and to be consistent. In our case we went first for consistency. Instead, by leveraging the higher-level history-oriented spec-infrastructure of our methodology, before one takes care of consistency, one could have shown first that high-level spec's and the architecture violate SysReqs (and these kind of proofs are generally history-oriented, thus much easier than induction-based proofs of inductively, state-oriented LLR automata proofs [14]). Then one does not need to prove HLR consistency (which involves coming up with LLR automata and showing their compliance to HLR), since they will be not good enough.

Concerning lessons learned, an interesting observation is that there are a couple of reasons for preferring the specification of components by means of an ADL ideally in a state-based fashion (coupled with a generator), rather then giving the user just an encoding of the stream data type and set of theorems over streams in Isabelle:

- The ADL is for a user more comfortable than writing recursive (specified typically as least fixed points [30]) stream processing functions in Isabelle.
- The input/output automata of this work are designed to describe realizable components per construction (and only these).
- The automata of our methodology are general enough to represent every realizable stream processing function (proof in [14]).

Finally, the table in fig. 6 summarizes how the methodology presented in this paper handles the industrial requirements described in the introduction. The biggest challenge encountered is having a proof being found automatically in a large model. The mitigation of this is an ongoing work consisting in increasing the number of general theorems over dataflow-based systems (which are identified intellectually during verifying case studies), and also exploiting the rapid increase of computational capability by using a central web-based high-performance service to perform the proof search.

However the application of reasoning over knowledge bases is still not mature enough and further research and time will be needed to make it a standard in the tool box of software engineering development processes.

Soundness	Established usually by peer reviews, from 1980-now there are over 200 papers about FOCUS.
Ability to be integrated into the DO-178x conforming process	The methodology replaces or complements many tests and supports fulfilling certification requirements. By relocating the majority of the code generator internally in Isabelle (we aim over >99%), the tool chain becomes easier qualifiable due to Isabelle's axiomatic and conservative nature.
Cost savings	Replaces traditional verification methods throughout the development life cycle, as seen in the running example (also see the point below)
Correctness by construction	Automata Language restricts the user into specifying only well-behaved implementable functions (vs expect user to write realizability proofs) Underlying methodology FOCUS: Refinement fully compositional – After decomposing a system, refining the components separately, and composing back, the new system is a refinement of the old one (no new (unwanted) behaviors are added, thus sparing test and integration costs)
Scalability: Compositional verification	Verification: Compatibility of composition with refinement allows modularizing and breaking down the proof complexity of representative industrial-sized models (as in the running example)
Expressivity of specification language	Automata language can represent all implementable Stream Processing Functions (semantical mapping is surjective, proof in [9])
Timing aspects and underspecification refinement	Supported as presented in the running example
Usability by normal software engineers on normal machines	A user-friendly frontend language and a high-level API in the knowledge base helps (for hiding low-level engine concepts) A large amount of encoded lemmas helps aiming for high automation (which implies less user expertise needed) - see demo in: https://www.youtube.com/watch?v=krl4Q7MAAlo

Fig. 6. Industrial requirements

REFERENCES

[1] A. Brahmi, D. Delmas, M. H. Essoussi, F. Randimbivololona, A. Atki, and T. Marie, "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.

[2] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine, "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.

[3] E. Payet and F. Spoto, *Checking Array Bounds by Abstract Interpretation and Symbolic Expressions*, pp. 706–722. 06 2018.

[4] T. Bochot, P. Virelizier, H. Waeselynck, and V. Wiels, "Model checking flight control systems: the airbus experience," pp. 18 – 27, 06 2009.

[5] S. P. Miller, A. C. Tribble, M. W. Whalen, and M. P. E. Heimdahl, "Proving the shalls: Early validation of requirements through formal methods," *Int. J. Softw. Tools Technol. Transf.*, vol. 8, no. 4, pp. 303–319, 2006.

[6] X. Diao, B. Liu, and S. Wang, "A survey of avionics analysis and simulation based on aadl model," *Int. J. Inf. Commun. Techol.*, vol. 9, pp. 282–299, Jan. 2016.

[7] F. Mhenni, J.-Y. Choley, N. Nguyen, and C. Frazza, "Flight control system modeling with sysml to support validation, qualification and certification," *IFAC-PapersOnLine*, vol. 49, pp. 453–458, 12 2016.

[8] B. Albert, H. Usach, J. Vila, and A. Crespo, "Development of integrated modular avionics applications based on simulink and xtratum," 05 2013.

[9] G. Berry, A. Bouali, X. Fornari, E. Ledinot, E. Nassor, and R. de Simone, "Esterel: a formal method applied to avionic software development," *Science of Computer Programming*, vol. 36, no. 1, pp. 5 – 25, 2000.

[10] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice, "Lustre: A declarative language for programming synchronous systems," in *POPL*, 1987.

[11] S. Kriebel, D. Raco, and B. Rumpe, "Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy's Streams Become Feasible?," in *[Software Engineering (SE) und Software Management (SWM), SE SWM, 2019-02-18 - 2019-02-22, Stuttgart, Germany]*, pp. 87–94, BMW Group, Chair of Software Engineering at RWTH Aachen, Feb 2019.

[12] J. Souyris, "Formal methods at airbus: Experience feedback," 2012.

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

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

[15] A. Haber, J. O. Ringert, and B. Rumpe, *MontiArc - Architectural modeling of interactive distributed and cyber-physical systems*, vol. 2012,3 of *Technical report / Department of Computer Science, RWTH Aachen*. Aachen and Hannover and Göttingen: RWTH and Technische Informationsbibliothek u. Universitätsbibliothek and Niedersächsische Staats- und Universitätsbibliothek, 2012.

[16] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A proof assistant for Higher-Order Logic*, vol. 2283 of *Lecture notes in artificial intelligence*. Berlin [etc.]: Springer, 2002.

[17] D. Harel and B. Rumpe, "Meaningful modeling: What's the semantics of "semantics"?," *Computer*, vol. 37, pp. 64 – 72, 11 2004.

[18] M. Broy and K. Stølen, *Specification and development of interactive systems: Focus on streams, interfaces, and Refinement*. New York: Springer, 2001.

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

[20] C. A. R. Hoare, "Communicating sequential processes," in *The origin of concurrent programming*, pp. 413–443, Springer, 1978.

[21] W. Reisig, *Petri nets: an introduction*, vol. 4. Springer Science & Business Media, 2012.

[22] J. O. Ringert and B. Rumpe, "A Little Synopsis on Streams, Stream Processing Functions, and State-Based Stream Processing," *Int. J. Software and Informatics*, vol. 5, no. 1-2, pp. 29–53, 2011.

[23] D. Cofer and S. Miller, "Do-333 certification case studies," in *NASA Formal Methods* (J. M. Badger and K. Y. Rozier, eds.), (Cham), pp. 1–15, Springer International Publishing, 2014.

[24] J. Warmer, A. Kleppe, T. Clark, A. Ivner, J. Högström, M. Gogolla, M. Richters, H. Hussmann, S. Zschaler, S. Johnston, D. Frankel, and C. Bock, *Object Constraint Language 2.0*. 01 2001.

[25] J. C. Bürger, H. Kausch, D. Raco, J. O. Ringert, B. Rumpe, S. Stüber, and M. Wiartalla, "Towards an Isabelle Theory for distributed, interactive systems - the untimed case," Tech. Rep. AIB-2020-02, RWTH Aachen, Jan. 2020.

[26] M. Broy and B. Rumpe, "Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung," *Informatik-Spektrum*, vol. 30, no. 1, pp. 3–18, 2007.

[27] C. Ballarin, "Locales and locale expressions in isabelle/isar," in *International Workshop on Types for Proofs and Programs*, pp. 34–50, Springer, 2003.

[28] L. Bulwahn, "The new quickcheck for isabelle: random, exhaustive and symbolic testing under one roof," pp. 92–108, 12 2012.

[29] J. C. Blanchette and T. Nipkow, "Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder," in *Interactive Theorem Proving* (M. Kaufmann and L. C. Paulson, eds.), (Berlin, Heidelberg), pp. 131–146, Springer Berlin Heidelberg, 2010.

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

APPENDIX

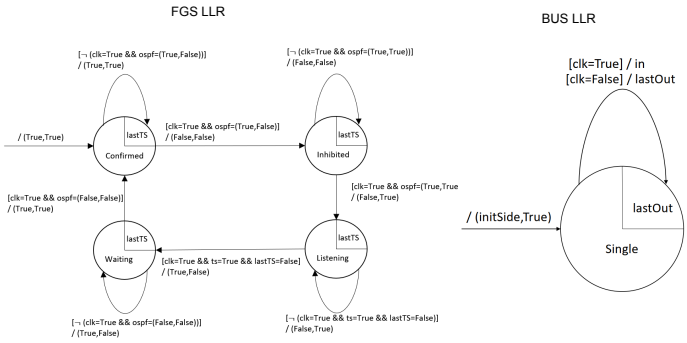

Fig. 7. Supplementary model: Automata for a Flight Guidance System component and a Bus. Both MontiArc explicit states (circles) and implicit states (variables inside the circles) are indifferently translated in Isabelle as elements of the state space. In the Bus $lastOut$ is a variable to save the last output message and in is the input message.

SysReq 2..5

- The System is in a $stable$ situation whenever both FGSs acknowledgements are $true$. If the system is in a stable state, then at most one side is the active pilot flying side.
- Pressing the transfer switch changes the inactive side, if the system is in a stable state and the inactive side receives the input.
- In the beginning one side has to be the active pilot flying side. Without loss of generality we choose the left side to be active. The right side has to be inactive.
- Only switch to inactive side, if transfer switch is pressed.

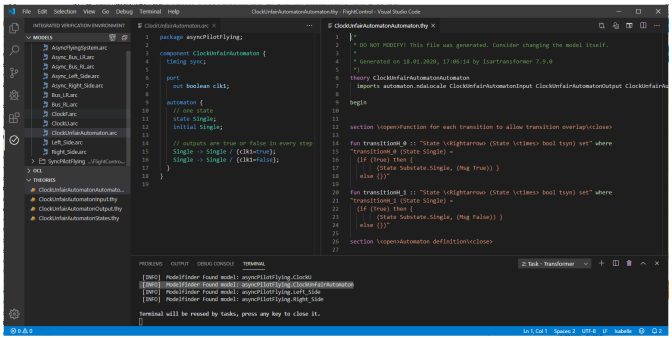

Fig. 8. Supplementary: Integrated Verification Environment