Model-Based Design of Correct Safety-Critical Systems using Dataflow Languages on the Example of SysML Architecture and Behavior Diagrams

Hendrik Kausch[1], Mathias Pfeiffer[1], Deni Raco[1], Bernhard Rumpe[1]

Abstract: An ADL-agnostic mapping from dataflow languages into a theorem prover is developed. A stream-based semantics for key modeling concepts based on the mathematical framework FOCUS and the theorem prover Isabelle is proposed. SysMLv2 Part Definitions, Requirement Definitions and State Definitions are automatically mapped to equivalent logical structures in Isabelle for verifying system design against requirements. A Pilot Flying System adapted from NASA and Rockwell Collins is used as running example.

1 Introduction

The goal of this paper is providing SysML [FMS14] users with a verification infrastructure by an automated mapping in a theorem prover. There are a number of reasons motivating this contribution: the increasing importance of model-based development in industry, SysML being currently a well-accepted systems-engineering language family with large tool support, the importance of model analysis to facilitate certification in safety-critical domains, as well as the high reliability provided by theorem provers.

There have been several related contributions on mapping a domain-specific language into a reasoning infrastructure. For example, [KLS18] presents an automated translation from the systems engineering modeling language SysML into the input languages of the NuSMV, Prism and Spin model checkers. Palladio Component Model for model-driven performance prediction [BKR09] offers a language for a component-based architectural modeling, and a model-transformation that translates these models into event based simulation code, which then gets executed. In comparison, the approach of our paper focuses on the model level only, achieving a verified (with respect to system requirements) executable model (represented by a deterministic state machine), but not discussing further the generation of object code from this model. Also, MechatronicUML offers a UML profile to model mechatronic systems using model-based design and providing formal analysis (e.g. by a translation to the model checker UPPALS [Ge15]). Similar to our proposed approach, it is based on well-defined semantics. In comparison to our proposed method, model checkers are known to have in general a higher automation degree. They suffer though from the state-space-explosion problem, where the verification effort increases exponentially with

[1] RWTH Aachen University.

the state space of the system. Hence it affects in general software verification more than hardware verification. Model checkers usually mitigate this by building equivalence classes of the state space of the system at the expense of accuracy. Theorem proving techniques achieve on the other hand a better coverage and suffer less from the state-explosion problem, where the proof size grows rather linearly with the increasing system size. Furthermore, the ADL MontiArc [HRR12] has been connected in our previous work with the theorem prover Isabelle by a code generator [Kr19, Ka20a, Ka20b].

A challenge is represented by the fact that SysML has been subject to fragmentation caused by intentionally vague syntax and semantics. Statecharts for example, one of the SysML sub-languages, counted over 20 different variants and respectively different interpretations at one point [VdB94]. Currently, a new version SysMLv2 [FS20] is in development. It aims to defragment the modeling landscape by defining an unambiguous meaning for model elements and by providing concrete textual syntax, which facilitates model processing. If SysML models shall be used not just as means of communication, but also for analysis and reasoning about system models, then unambiguous semantics [GB11] is required.

This paper associates a general semantical foundation to dataflow-based languages, enabling in particular reasoning over SysMLv2 specifications. The creation of a sound generic specification and verification infrastructure for dataflow-based modeling languages in the theorem prover Isabelle has been initiated in our previous work [Bü20]. The stream-based methodology FOCUS [BS01] is used as mathematical underpinning and is encoded in Isabelle. The semantics of a (non-deterministic) component is a (set of) stream processing functions (SPFs), where streams describe the dataflow on communication channels. The behavior of atomic components is specified by state machines or predicates over the components input and output streams. The encodings in the theorem prover, consisting mainly of type definitions, function definitions, and theorem definitions, enable reasoning over system models.

By generalizing our previous translation from MontiArc to Isabelle, an ADL-agnostic code generator mapping to the Isabelle stream-based encodings is created as depicted in Fig. 1. SysMLv2 is used as another example of a modeling language, where SysML specifications are analogously also semantically understood as FOCUS dataflow networks.

This paper thus extends previous works by the following:

- factorizing the commonalities of ADL-to-Isabelle code generation into a generic framework, and thus minimizing ADL-specific development costs for future extensions,
- reporting on the results of mapping SysMLv2 Part Definitions, Requirement Definitions and State Definitions into equivalent structures in Isabelle using an avionics case study as running example.

The remainder of this paper is structured as follows: First, a running example is presented.

Afterwards, the generic semantical mapping is described. Next, the implementation of the generic ADL-agnostic framework using an intermediary representation is presented and results on mapping SysML models to Isabelle are reported. Finally, a conclusion is presented. Supplementary meta models, formal textual specifications and graphical representations are provided in an Appendix.

2 Example of a Cyber-physical Avionics System

A possible design of a Pilot Flying System (PFS) adapted from a case study of NASA and Rockwell Collins [CM14] is presented. This is a simple example, yet representative for a class of avionic protocols (Fig. 2 shows also a graphical overview). The PFS system consists of 4 components. The two components on the left and right side are redundant flight guidance system (FGS) and later referred to as side component. Each of them communicates through a bus with the other side component. Additionally, a pilot might interact with the system by using a transfer switch which sends a signal to the side components. There are a couple of system requirements (SysReqs) to the whole system, for example that at least one Side component is active at all times or that in the beginning one component is active and the other is inactive.

The systems components can start by being underspecified and shall be step-wise refined in a correct manner (i.e. without losing the ability to fulfill such wanted SysReqs). The side and bus components form a feedback cycle and this increases the complexity significantly compared to sequential or parallel compositions. The complexity is further amplified by having a highly-underspecified context in form of disturbances for each component, which uncontrollably dictate whether a component is able to function correctly. Thus, the PFS systems components might separately fail to act for an arbitrary time or be completely faulty.

While the disturbance context cannot be controlled, the system developer can make assumptions about it and thus, e.g. by means of the prominent assumptions and guarantees specification style [MC81, AL94, AL95, BS01]. Properties of components are hereby described as a guaranteed behavior if its context behaves according to the assumptions.

Refinements of the system components can take the form of implementation-close state-based specification styles (Fig. 3) towards the final phases of the development cycle. While being relatively low-level and not necessarily deterministic yet, this style does guarantee that the specification is consistent (guarantees the existence of an implementation, [Kr19, Ka20a]). The interface of the side component has three input channels and one output channel. Its inputs channels are the one receiving signals from the bus component, the transfer switch forwarding signals from the pilot and the environmental disturbances. Its produced output is then send via another bus to the other side component. Internally, it is defined by a state-machine with input and output [Ru96]. Transitions and their labels have been omitted for simplicity.

3 Verification Tool-Chain

This paper gives a stream-based semantics to a class of dataflow modeling languages by identifying their commonalities and then mapping these into an Isabelle encoding of FOCUS [BS01]. The Appendix A offers a more detailed introduction.

3.1 The Tool-Chain

In Fig. 1 the new design of the updated ADL-agnostic Tool-Chain is depicted. System models and desired properties/requirements are transformed to Isabelle code. This generated code is divided in different theory files for each component and system. These theory files contain generated model specific Isabelle datatypes, functions and theorems with generated proofs over datatypes and functions. The underlying implementations of core constructs of FOCUS in Isabelle (core in Fig. 1) are on the other hand imported by the model specific generated Isabelle files. Thus, the structure and semantics of the input modeling language is based on the FOCUS/Isabelle encoding.

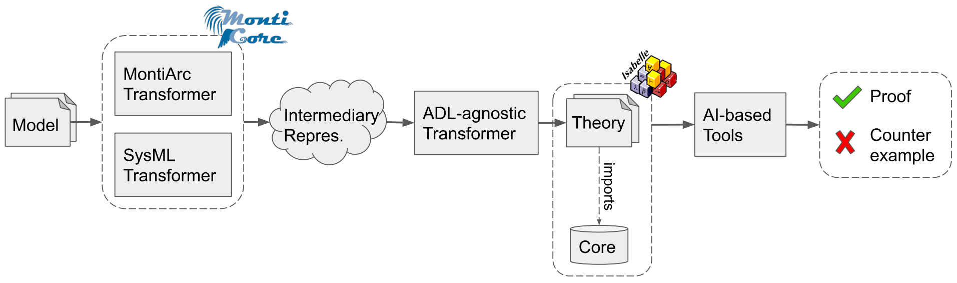

Fig. 1: Overview of our architecture description language (ADL)-agnostic verification framework

Our framework is updated in this work to be flexible in regards to the ADL being used. This is achieved by using an ADL-agnostic *intermediary representation*. For this, the ADL is first transformed into this intermediary representation. From there, a generator transforms the intermediary representation into Isabelle theories. These generated model specific theories integrate then with the core FOCUS/Isabelle implementation. After their generation, the Isabelle theories are processed by AI-based automated tools. These include an integration of the proof-finder Sledgehammer [PB10], enhanced with a Machine Learning filtering mechanism for prioritizing proofs, as well as scripts of heuristic common proof-finding activities which simulate human Isabelle users. Reasoning activities are intended to optimize certification costs and efficiency and consist usually in finding proofs, counterexamples as error detection (e.g. for refinement relations), as well as simulation execution.

Finally, architecture specifications of systems can become quickly overwhelming and difficult to handle with increasing size. The illustration of textual specifications by means of graphical representation of the hierarchical structure is a helpful step towards practicability. For this, an Integrated Verification Environment (IVE) was extended as a graphic interface to support operating the Tool-Chain and its verification process. An example of a modular system development accompanied by compositional verification is demonstrated in a short video [Ka20c].

4 ADL-agnostic Verification Framework

Until now, only MontiArc [HRR12] models were transformable to FOCUS/Isabelle by our tool-chain. This paper extends the tool-chain by generalizing ADL unspecific parts of the Isabelle code generation and reports on ongoing work of adding SysML as another possible user input language.

Based on Sect. 3.1 and well-known methodological requirements mentioned in Appendix A, this chapter starts by introducing the intermediary representation's meta model. These are the constructs and rules for building the intermediary representation of a specific system. This will determine the expressiveness of our framework, as it defines the maximum amount of information that can be formalized and thus verified.

4.1 ADL-agnostic Meta Model

The main concept of the intermediary representation's meta model is a stream processing specification (SPS) (set of SPFs). Every component that shows a certain behavior, be it explicitly defined (e.g. by a state machine) or implicitly inherited via composition, can be modeled as such a set of SPFs. Fig. 4 shows SPSes and directly related concepts in the form of a class diagram (CD). CDs model classes (concepts) as named boxes with optional properties. Relations between those concepts are depicted as connections.

Each SPS defines an input and output datatype. These datatypes in turn define the *channel*s over which the SPS communicates. Each *channel* transports one strongly typed stream of messages. These streams are also referred to as communication histories.

SPSes are divided into two categories: *Definition*s and *Instance*s. A *Definition* describes behavior and interface while an *Instance* is one of potentially many occurrences or usages of a *Definition*. A *Definition* can have *Parameter*s which can be set differently in each *Instance*, as explained in List. 13. A *Definition* can also be a *Refinement* of other *Definition*s.

A *Definition* is either a state based behavior specification *Automaton*, a history based behavior specification (abbreviated from now on as *Specification*), or a *Composition*.

State based behavior specifications enable imperative behavior modeling. Our meta model implements I/O automata [Ru96] adapted to input and output *Channel*s. An I/O automaton in general is defined as a five tuple $(S, I, O, \Delta, Init)$ of state space S, input and output channel sets I respectively O, transition relation Δ, and set of initial configurations $Init$. Fig. 5 overviews the intermediary representation's meta model for *Automaton*s. Modeling infinitely large state spaces is achieved using *Variable*s. The input and output *Channel*s of the respective *Datatype*s are omitted as they were already introduced with Fig. 4.

A transition relation Δ is a subset of $(S \times I^{\Omega} \times S \times O^{\Omega})$. The notation I^{Ω} denotes stream tuples enhanced with channel labels from the set of labels I (also known as stream bundles [Ru96]).

The transition relation's intermediary representation looks as follows: A starting *State* and input determine a next *State* and *Output*s. Our implementation enables infinitely large transition relations via classes of transitions. A *Transition* is a collection of transitions according to their preconditions, their target *State*, and their *Output*. The preconditions (*guard*) determine whether the transition is „active“ and allowed to proceed (fire). *Guard*s can check state variables and inputs using predicates. Those predicates contain Isabelle compatible Boolean expressions and are stored as strings for flexibility.

*State*s (or their *VariableValue*s) and *Output*s can similarly use such predicates to define potentially infinitely large sets of alternative variable and output valuations. We use a technique based on predicate set builder notation, from now on referred to as set comprehension. Both direct valuation and set comprehension use the same intermediary representation. The two modes are distinguished by an additional *isComprehension* flag. If comprehension is on, *value* describes a selector for a set of values instead of a single one. Programmatically speaking, from all possible values (of fitting type), one that matches the selector is selected „at random“ and then assigned to a variable or output. See Fig. 6 for a class based overview. Please find an example of set comprehension in Sect. 4.4.1 and Fig. 12.

The initial configurations *Init* generally are a subset of $(S \times O^{\Omega})$, i.e., a relation between initial state S and output bundle O^{Ω}. Their implementation is fundamentally identical to the *Transition*s.

History based *Specification*s enable declarative behavior modeling. They specify desired behavior via relations between input and output streams, stored as String-based Isabelle predicates. These predicated select the desired behaviors from the set of all behaviors, i.e., the set of all implementable functions. This means they effectively employ assumption-guarantee behavior modeling.

Finally, non-atomic components' behavior implicitly results from the flow based composition of their parts. These parts are called *Instance*s. They represent occurrences of a *Definition* and cannot exist outside of a *Composition*. An *Instance* might assign *ParameterValue*s to its *Definition*'s *Parameter*s. Fig. 7 gives an overview. *Instance*s are important as they allow re-use of existing descriptions in new locations. This greatly reduces verification complexity and cost. A *Definition* acts as a central library for all *Instance*s thereof. The *Definition* stores its architectural and behavioral description, as well as any additional properties and obtained verification results of those descriptions. The *Instance*s can then access this information without having to reproduce it. We achieve this library re-use functionality by connecting each *Instance* to the *Definition*. The *Instance* merely wraps the *Definition*s input and output *Datatype* to adapt it to each usage location. This connects the appropriate *Channel*s via internal mappings.

4.2 Verification of SysML Models

We use SysML to exemplarily show the usage of a different (from MontiArc) modeling language as input for our Isabelle verification infrastructure. The official SysMLv2 specification is, as of writing this paper, not released yet and does not include a concrete semantics definition. We thus took a subset of available language components and gave them a dataflow semantics. This subset was selected to allow us to describe state based behavior, history based behavior, and composition, as well as parameters for and refinement relations between all three.

We now detail this language subset and its syntax as we understand it from current development versions of the SysMLv2.

4.2.1 State Definitions

A State Definition can be used in a Part Definition to describe the behavior. The example in Listing 1 defines the *Bus* subsystem introduced in Sect. 2. At first, a Part Definition (PD) defines its name and parameters. The PD also indicates potential refinement relation via the *specializes* keyword (abbrev. "*:>*"). The PD's interface is defined using the *port* keyword, followed by a unique name and a type. The type of ports is defined separately in so called port definitions. It suffices to say that port definitions define a name and individual communication lines. Each communication line has a direction (*in* or *out*) and a type (e.g., *Boolean*). We restrict port definitions to one communication line called *val* for simplicity.

The PD does not directly specify behavior but rather delegates it to an State Definition (SD). To do so, an occurrence of a state based description is declared using *state* and then given a name. The type of description is selected by preceding it with a colon. All relevant communication lines of the ports, as well as parameter are passed through to the SD in a comma separated list.

Listing 2 shows the partial *BusAutomaton* that is being referred to in the PD. The SD is based on the state based behavior of the bus subsystem introduced in Sect. 2. Variables, states, and transitions are be defined in the body of the SD, as is shown in List. 3. The variable defined there will be used to store previous messages. Transitions specify the behavior. A transition uses current state and guard to determine whether it is active. If it fires, the action is executed, sending values to ports and variables. Lastly, the automaton switches into the next state. The example transition in Listing 4 reacts according to an external disturbance *dist*. If the automaton is in state *Single* (line 2) and no disturbance occurs (line 3), the message is transmitted and the *last* variable stores the transmitted message (line 4).

4.2.2 Requirement Definitions

The example in Listing 5 shows a Requirement Definition (RD) for the *Side* component introduced in Sect. 2. Similar to SDs, a PD definition starts by defining the parameters and interface of the component. The RD specifies which PD is to be restricted via the *subject* keyword, followed by colon-separated handle and type. The subjects behavior is restricted using predicates over input and output history.

The possibility to formulate expressions is of key concern. We used the existing MontiCore (framework for developing domain specific languages (DSLs)) expression library [HR17] here. The predicate uses the previously linked *Side* component via its handle *side*. The predicate models an implication using *implies* and a conjunction using *&&*.

4.2.3 Composition

The example in Listing 6 shows the PFS component introduced in Sect. 2. After defining the interface and the disturbance context (described partially in line 2-5) as described in Sect. 4.2.1, the parts are then defined (line 7-11) in the body. The *part*s are connected to each other and to the compositions interface via communication channels. These channels are created using the *connect* keyword, followed by a source port, the keyword *to*, and a target port.

4.3 Transformation from SysML to Intermediary Representation

Based on well-known concepts for system modeling such as modularity, underspecification, composition, and refinement (Appendix A), we crafted an intermediary representation as abstraction between language specific notation and verification framework. Then, we selected a subset of the SysMLv2 language family that allows us to model all intermediary concepts. Now, we show a mapping of SysMLv2 models to our intermediary representation. This effectively gives SysMLv2 (or a subset thereof) a precise mathematical semantics (please refer to Appendix E for intermediary diagrams).

4.3.1 State Definitions

The accompanying PD is used to create the input and output *Datatype*. Each *port* results in a *Channel* with appropriate *name* and Isabelle *type*. The *Parameter*s are similarly derived. If *specialize*'d components are specified, a *Refinement* (see Fig. 4) relation is created linking the *refined* (the PD with the keyword) and the *original*. An example for the *Bus* SD from Sect. 4.2 is shown in Fig. 8.

The SD is then mapped to an *Automaton*. The *StateSpace* is derived from *State*s defined via *state* keywords, and *Variable*s defined via *value* keywords. An example based on the state space from Sect. 4.2 is shown in Fig. 9. The *Transition*s are essentially one-to-one mappings of the abstract syntax of transitions in SysML. Only the *guard* and *value*s are transformed to valid Isabelle expressions. An example based on the transition from Sect. 4.2 is shown in Fig. 10.

4.3.2 Requirement Definitions

RDs are mapped to *Specification*s. Each predicate is transformed into a valid Isabelle predicate and then stored in a list of strings.

4.3.3 Compositions

The *part*s of a compositions are mapped to *Instances*. The "wiring"denoted by *connect* is transformed into *Channel*s and a mapping of those *Channels*. We then calculate input and output *Datatype* for each *Instance* as the set of *Channel*s that are connected to the *Instance* in question. Finally, we construct the internal mappings (Sect. 4.1) from *Instance* datatype to *Definition* datatype. Fig. 11 shows a partial object diagram (OD) based on the PFS model from Sect. 4.2.

4.4 Mapping the Intermediary Representation to FOCUS/Isabelle and Evaluation

As next, we develop a mapping from the intermediary representation to our existing FOCUS/Isabelle implementation [Kr19, Ka20a]. This mapping enables tool-supported automated verification and reasoning. Please see Appendix F for details on the generated Isabelle specifications of the running example. We then evaluate our results by successfully verifying a System Requirement of the PFS.

4.4.1 Automata

We first encode the *Automaton* in Isabelle straight forwardly and then convert to an SPS in Isabelle internally. For this, we use a translation to a set of functions according to [Ru96]. This reduces the complexity of encoding, i.e., the gap between meta model and Isabelle concepts, increasing the encoding's confidence. This also allows us to implement the translation in a verifiable environment (Isabelle), further improving confidence in its correctness.

The transition function is the aggregation of all distinct transitions. We use the term transition loosely here, denoting not just a single transition in the theoretical sense but rather the modeled entity *Transition*. This entity could, as explained in Sect. 4.1, define multiple or even infinitely many transitions. Each transition is printed to a separate precondition (*guard*) and matching result. This separation enables better counter example finding (via simulation execution).

The *guard* predicate in Listing 7 is generated from the first transition of the *Bus* automaton in Sect. 4.2.1. The *guard* determines the activeness of the transition using a Boolean, a state, and a Boolean tuple. The state *s* is checked to be *Single* and *dist* to be *False* at the same time via conjunction ($\wedge$).

An exemplary *result* of a transition based on the *Bus* from Sect. 4.2.1 is listed in List. 8. Non-determinism or underspecification is encoded as a set of alternatives. The right hand side of the equality specifies a singleton set of tuples, consisting of a target state *Single* with variable value *i*, i.e., storing the current input. The output is set to be equal to the input *i*.

The actual transition shown in Listing 9 then joins *guard* and *result*. If the *guard* holds, the *result* is produced. Otherwise an empty set is returned, denoting an inactive transition, i.e., preconditions were not met.

The use of a guard expression and set comprehension for both outputs and state variables enables the modeling and encoding of potentially infinitely many transitions. As an example, please consider the set comprehension for *Output*s in Fig. 12. The (simplified) *result* is shown in List. 10, encoded as a set constructed using set comprehension.

Furthermore, in case the transition function is known to be finite, we print a second version of the transition function using finite lists instead of potentially infinitely large sets. This enables us to randomly select a transition, making the automaton executable. The execution can be handled by Isabelle directly or externally via an integrated code export mechanism. The execution of automata enables counter-example finding, reducing the costly verification process by finding mistakes early. Line based error reporting allows developers to continually and purposefully refine their model.

4.4.2 Specifications

The printing of *Specification*s is straight forward. The set of predicates acts as restrictions on the set of all possible SPSes. The predicates reason about the entire input and output history, i.e., *Stream*s.

The example in Listing 11 shows a shortened predicate *pred* for the *Side* RD introduced in Sect. 4.2.2. Parts of input stream types are omitted for readability. The function takes parameters and all input and output streams as input and returns a Boolean.

The actual behavior, i.e., the SPS or set of SPFs is then defined by employing predicate based set building (see Sect. 4.4.1). All possible SPF *spf* are filtered by all predicates. This makes *Specification*s potentially not consistent, i.e., it is possible to model contradicting predicates that leave an empty set of possible implementations.

A simplified version of the *Side* SPS based on the above predicate *pred* is shown List. 12. The input of (omitted) Boolean parameter *initial* results in a set of function from bundles of streams over the channel set I to bundles of streams over O. Using a set of functions allows non-determinism or underspecification.

4.4.3 Compositions

Compositions merely set or pass-trough parameters to the *Instance*s and then apply the general composition operator ([Kr19, Ka20a]) $\bigotimes$ to join the parts. Composition is both associative and commutative, meaning that the order of composition does not change the resulting SPS.

The PFS composition is shown in List. 13. As it does not have parameters, the empty type *unit* is used for its parameters. Parameter values (*True* and *False*) are set according to the modeled parts in SysML as defined in List. 6.

4.4.4 Refinement and Evaluation

Refinement relations are printed into theorems. The theorems state that the behavior (as a set of functions) of a refined component is a subset of the behavior of the original. For the *Refinement* between *UnreliableBus* and *Bus* as introduced in Sect. 4.2.1, is shown in Listing 14.

Contrary to the architecture and behavior encodings, the refinement Isabelle theories are subject to further optimization. The MontiBelle framework uses proof and counterexample finding tools such as Sledgehammer [PB10], PSL [NK17], Quickcheck [CH11, Bu12], and Nitpick [BN10] in later stages of the verification pipeline. These tools are supported by a library of proof templates, applied automatically during transformation. Counterexamples can be shown to the user while proofs are injected and saved in the appropriate theory.

Finally, we evaluated our tool-chain by verifying a System Requirement of the PFS. It was shown that, at the beginning, the left pilot side is active and the right one is inactive. This property holds for the Isabelle specification of the overall PFS. The behavior of the Isabelle representation of the PFS is derived as follows. State Definitions in SysML are transformed into Isabelle automata. These automata are transformed by a semantical mapping [Ru96] into stream processing functions. The stream processing functions of all atomic components are then composed using the encoded composition operator of FOCUS in Isabelle [Bü20] to form the specification representing the behavior of the PFS. It is then shown that the property is fulfilled by the systems' specification. The core encodings of composition over functions [Bü20] enabled a highly automatic, easy-to-find proof (see theorem „SysReq“ in Appendix F).

5 Conclusion

In this paper we presented a general concept for verifying properties of systems developed using dataflow languages. First, we highlighted concepts such as modularity, underspecification, composition, and refinement as important for a coherent systems engineering method. We then developed an intermediary representation based on these concepts with FOCUS as foundation in mind. We used this representation to factor out the ADL-agnostic part from an the existing ADL-specific code generator. This allowed us to isolate our framework from specific ADLs.

Finally, we reported about trying out the generic infrastructure by enabling the use of yet another modeling language, SysMLv2, as user input language, by transforming SysML models to our intermediary representation's meta model and then mapping these into equivalent Isabelle constructs. Deriving the correctness of a System Requirement of the Pilot Flying System, as shown in the supplementary artifacts consisting in specifications and theorems, provides high confidence on the correctness of the model transformations. Furthermore, we believe that a wider-spread accepted semantics for (at least a subset of) SysML would be highly desirable for verification and reasoning in general.

In the future, it is planned to extend the support for additional modeling languages, as well as revising our current understanding of the already supported structures. We think that our intermediary representation proved to be sufficiently generic during this work. We expect it to be well extendable for future modeling languages, and it should reduce development costs for ADL-specific code generators. However, the intermediary representation's meta model is still subject to constant improvements.

Bibliography

[AL94] Abadi, Martín; Lamport, Leslie: Open systems in TLA. In (Anderson, James; Peleg, David; Borowsky, Elizabeth, eds): Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing - PODC '94. ACM Press, New York, New York, USA, pp. 81–90, 1994.

[AL95] Abadi, Martín; Lamport, Leslie: Conjoining specifications. ACM Transactions on Programming Languages and Systems (TOPLAS), 17(3):507–535, 1995.

[BKR09] Becker, Steffen; Koziolek, Heiko; Reussner, Ralf: The Palladio component model for model-driven performance prediction. Journal of Systems and Software, 82:3–22, 01 2009.

[BN10] Blanchette, Jasmin Christian; Nipkow, Tobias: Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder. In (Kaufmann, Matt; Paulson, Lawrence C., eds): Interactive Theorem Proving. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 131–146, 2010.

[BR07] Broy, Manfred; Rumpe, Bernhard: Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung. Informatik-Spektrum, 30(1):3–18, 2007.

[BS01] Broy, Manfred; Stølen, Ketil: Specification and development of interactive systems: Focus on streams, interfaces, and Refinement. Springer, New York, 2001.

[Bu12] Bulwahn, Lukas: The New Quickcheck for Isabelle. In: Certified Programs and Proofs, volume 7679 of Lecture Notes in Computer Science, pp. 92–108. Springer Berlin Heidelberg, 2012.

[Bü20] Bürger, Jens Christoph; Kausch, Hendrik; Raco, Deni; Ringert, Jan Oliver; Rumpe, Bernhard; Stüber, Sebastian; Wiartalla, Marc: Towards an Isabelle Theory for distributed, interactive systems - the untimed case. Technical Report AIB-2020-02, RWTH Aachen, January 2020.

[CH11] Claessen, Koen; Hughes, John: QuickCheck: a lightweight tool for random testing of Haskell programs. Acm sigplan notices, 46(4):53–64, 2011.

[CM14] Cofer, Darren; Miller, Steven: DO-333 Certification Case Studies. In (Badger, Julia M.; Rozier, Kristin Yvonne, eds): NASA Formal Methods. Springer International Publishing, Cham, pp. 1–15, 2014.

[FMS14] Friedenthal, Sanford; Moore, Alan; Steiner, Rick: A practical guide to SysML: the systems modeling language. Morgan Kaufmann, 2014.

[FS20] Friedenthal, Sanford; Seidewitz, Ed: A Preview of the Next Generation System Modeling Language (SysML v2). 09 2020. `https://www.ppi-int.com/ppisyen95/`.

[GB11] Graves, Henson; Bijan, Yvonne: Using formal methods with SysML in aerospace design and engineering. Annals of Mathematics and Artificial Intelligence, 63(1):53–102, 2011.

[Ge15] Gerking, Christopher; Dziwok, Stefan; Heinzemann, Christian; Schäfer, Wilhelm: Domain-Specific Model Checking for Cyber-Physical Systems. MoDeVVA, Models, 09 2015.

[HR17] Hölldobler, Katrin; Rumpe, Bernhard: MontiCore 5 Language Workbench Edition 2017. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag, December 2017.

[HRR12] Haber, Arne; Ringert, Jan Oliver; Rumpe, Bernhard: 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.

[Ka20a] Kausch, Hendrik; Pfeiffer, Mathias; Raco, Deni; Rumpe, Bernhard: An Approach for Logic-based Knowledge Representation and Automated Reasoning over Underspecification and Refinement in Safety-Critical Cyber-Physical Systems. In: SE-WS 2020: Software Engineering workshops 2020 : combined proceedings of the workshops at Software Engineering 2020, co-located with the German Software Engineering Conference 2020 (SE 2020) : Innsbruck, Österreich, March 05, 2020. volume 2581 of CEUR workshop proceedings, 17. Workshop Automotive Software Engineering, Innsbruck (Austria), 24 Feb 2020 - 25 Feb 2020, [RWTH Aachen], [Aachen, Germany], Feb 2020.

[Ka20b] Kausch, Hendrik; Pfeiffer, Mathias; Raco, Deni; Rumpe, Bernhard: MontiBelle-Toolbox for a Model-Based Development and Verification of Distributed Critical Systems for Compliance with Functional Safety. In: AIAA Scitech 2020 Forum. p. 0671, 2020.

[Ka20c] Kausch, Hendrik; Pfeiffer, Mathias; Raco, Deni; Rumpe, Bernhard: , Verified Design of Safety-Critical Cyber-Physical Avionics Systems with the MontiBelle Framework, 2020. `https://youtu.be/cl4O3KXZrrc`.

[KLS18] Kölbl, Martin; Leue, Stefan; Singh, Hargurbir: From SysML to Model Checkers via Model Transformation. SPIN, pp. 255–274, 06 2018.

[Kr19] Kriebel, Stefan; Raco, Deni; Rumpe, Bernhard; Stüber, Sebastian: 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]. BMW Group, Chair of Software Engineering at RWTH Aachen, pp. 87–94, Feb 2019.

[MC81] Misra, Jayadev; Chandy, K. Mani: Proofs of networks of processes. IEEE transactions on software engineering, (4):417–426, 1981.

[NK17] Nagashima, Yutaka; Kumar, Ramana: A proof strategy language and proof script generation for Isabelle/HOL. In: International Conference on Automated Deduction. Springer, pp. 528–545, 2017.

[PB10] Paulson, Lawrence C.; Blanchette, Jasmin Christian, eds. Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers, 2010.

[Ru96] Rumpe, Bernhard: Formale Methodik des Entwurfs verteilter objektorientierter Systeme. PhD thesis, Zugl.: München, Techn. Univ, Zugl. München, 1996.

[VdB94] Von der Beeck, Michael: A comparison of statecharts variants. In: Formal techniques in real-time and fault-tolerant systems. Springer, pp. 128–148, 1994.

Appendix

A Dataflow-based Specification with the FOCUS Framework

We recall some well-known methodological concepts. A system modeling methodology should be able to support:

- Modularity
- Underspecification and non-determinism
- Composition
- Refinement
- Abstraction from concrete implementation
- Encapsulation
- Possibility to specify real-time properties
- Reasoning

Modularity implies separation of concerns. Supporting different abstraction levels by underspecification is important for reducing the complexity of development. Composition is key for a top-down development. One wants to decompose the system, refine the components separately from higher-level requirements, to lower-level ones, until an implementation, and then compose back. For this, it is important that composition is compatible with refinement. Encapsulation in a hierarchical decomposition ensures information hiding. Furthermore, the capability of modeling time, particularly in safety-critical applications (e.g. airbags), is crucial and can be considered as a functional requirement. Finally, one wants to be able to reason about system models.

FOCUS is a mathematical methodology for capturing these aspects. It provides higher-level history-oriented specifications (equations connecting complete input and output stream histories), as well as implementation-close state-oriented specifications by automata (Fig. 3). Underspecification is possible by either defining sets of stream processing functions by predicates over streams, or by using non-deterministic state machines with input and output. The key property of FOCUS is that **refinement is fully compositional** [BR07]. The key concept of FOCUS is the stream representing an observation of a channel history. Components communicate by message passing through unidirectional channels. The semantics of a component is a set of stream processing functions (also called stream processing specification or SPS). The behavior of atomic components can be specified by state machines, a style close to implementation or by using a high-level history-based specification approach [BS01].

B Overview Graphics for the Pilot Flying System

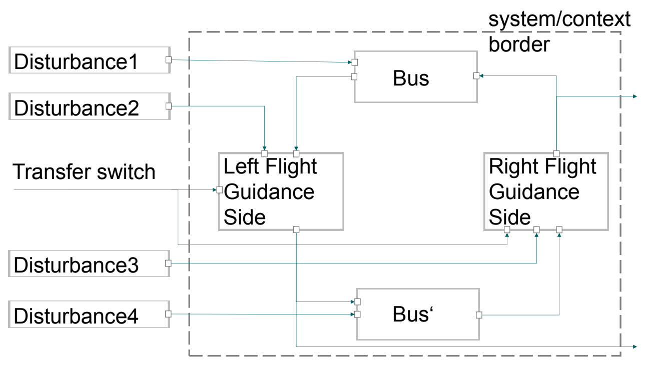

Fig. 2: Overview of the Pilot-Flying-System (PFS) [CM14]

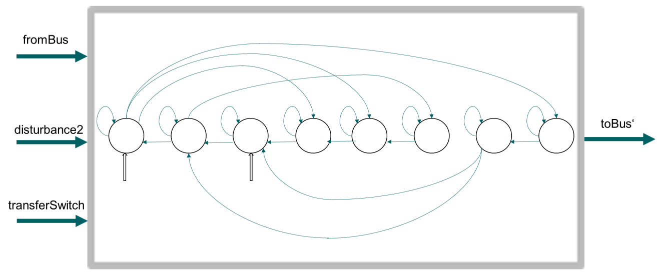

Fig. 3: State-based specification of the left Side component of the Flight Guidance System (transition labels omitted for simplicity)

C Meta Model of Intermediary Representation - A class based overview

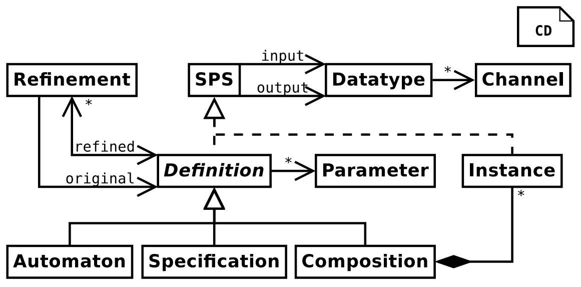

Fig. 4: Core components of the ADL-agnostic intermediary representation's meta model. Main concept is an SPS. Each SPS defines an input and output datatype. SPSes are divided into two categories: *Definition*s and *Instance*s. A *Definition* is either a *Automaton*, a *Specification*, or a *Composition*. *Refinement*s relate refined *SPSes* to the original.

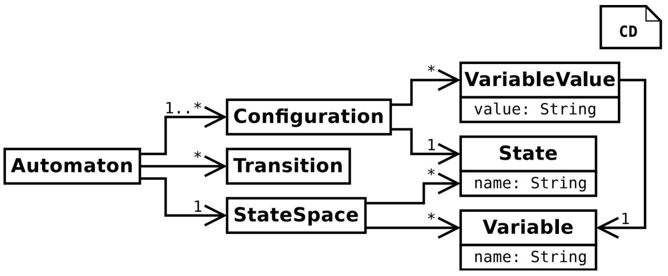

Fig. 5: An *Automaton* consists of an initial *Configuration*, a *StateSpace*, and *Transition*s. The *Datatype*s from Fig. 4 are omitted for brevity.

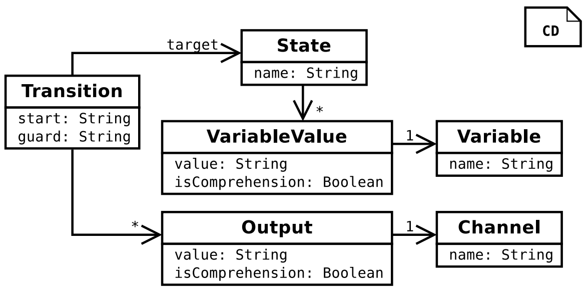

Fig. 6: A *Transition* has a *start*, a *guard*, a target *State*, and *Output*s. Both the *State*'s *VariableValue*s and *Output*s define an *isComprehension* flag to handle infinitely large transitions.

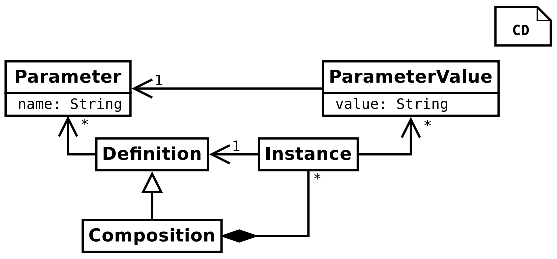

Fig. 7: A *Composition* is a *Definition* and consists of *Instance*s. An *Instance* sets the *Parameter*s of a *Definition* via *ParameterValue*s.

D SysML Specifications for the PFS

```
part def Bus(initial:Boolean) :> UnreliableBus {
  // Define the interface
  port dist: InBool;
  port i: InPair;
  port o: ~InPair;

  // Pass through
  state behavior: BusAutomaton(
    dist::val, i::val, o::val, initial);
}
```

List. 1: Part Definition of the Bus. First few lines define name, refinements, interface, and parameters. *Bus* takes a Boolean parameter named *initial*. For the sake of this example, we assume the *Bus* refines a not further detailed non-deterministic *UnreliableBus*, which randomly loses and alters messages. *initial* parameter can be later used (not shown here) to define the starting output of the *Bus*. Ports *dist* (disturbance) and *i* are typed to *InBool* and *InPair* respectively. Both contain exactly one input communication line named *val*. Types are *Boolean* and *Pair* (an implementation of a 2-tuple). Output port *o* makes use of the already defined *InPair* port definition by simply inverting its direction via the tilde symbol. Last few lines define the named occurrence *behavior* of the *BusAutomaton* and pass the communication lines and parameter.

```
state def BusAutomaton(
  in dist: Boolean,
  in i: Pair<Boolean, Boolean>,
  out o: Pair<Boolean, Boolean>
  initial: Boolean)
{
  ...
}
```

List. 2: State Definition of the Bus. Keywords *state def* denote an SD. Communication channels are defined using directional keywords *in* and *out*. Variables do not have a direction. Each communication channel and variable define a name and colon-separated type. *Pair* type allows for type parameters between angle brackets to specify the type of each element in the tuple.

```
value last: Pair<Boolean, Boolean>;
state Single;
```

List. 3: Variable definition using the keyword $value$ and name $last$ of type $Pair\langle Boolean, Boolean\rangle$, followed by the declaration of the only state named $Single$.

```
transition
  first Single
  if dist == false
  do action {send i to o; send i to last;}
  then Single;
```

List. 4: Transition definition in SysML SD. Transition reacts according to an external disturbance *dist*. If automaton is in state *Single* and no disturbance occurs, the message is transmitted and stored in *last*.

```
requirement {
  // Link to definition
  subject side: Side;

  // Uses MontiCore expressions
  require constraint {
    side::initial implies
```

```
        (side::ospf::nth(0)::getFirst()
         && ...);
    }
    ...
}
```

List. 5: Requirement Definition inside the Part Definition of the Side Component. Each predicate starts with *require constraint*. Parameter *initial* and output history on port *ospf* is accessed using double colon (*::*) notation, similar to Java's or C#'s dot notation. A function *nth* was defined to allow access to a specific time slice, *0* here being the first one. Since *ospf* transports *Pair*, we access the first element of the tuple via a getter function *getFirst*.

```
part def PFS {
  // Define the interface
  port dist1: InBool;
  port dist2: InBool;
  ...

  // Define the subcomponents
  part lside: Side(true);
  part lrbus: Bus(true);
  part rside: Side(false);
  part rlbus: Bus(false);

  // Define the communication channels
  connect dist1 to lside::dist;
  connect dist2 to lrbus::dist;
  ...
}
```

List. 6: Part Definition of the composed Pilot Flying System. Each *part* specifies a handle (e.g., *lside*, abbreviation for left side) and its type. The type (e.g., *Side*) also takes parameter values in round brackets. Parameter values can be literals (e.g., *true*), parameters of the composition being passed through, or expressions of unlimited complexity based on the previous two options. When defining communication channels, ports of the subparts are referenced using double colon notation introduced in Sect. 4.2.2.

E Object Diagrams (ODs) - intermediary representations of the PFS

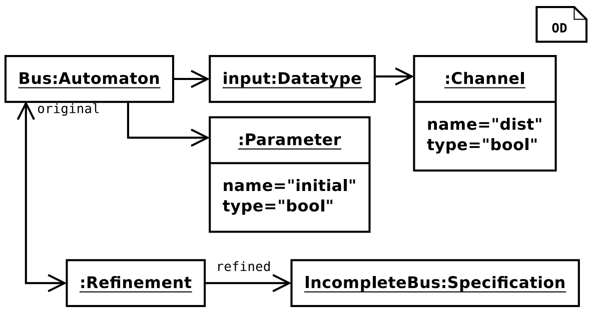

Fig. 8: Object Diagram (OD) of a concrete *Automaton* adhering to the CD in Fig. 4. The example shows a partial representation of *Datatype*s, *Parameter*s, and *Refinement*s from the SD in Sect. 4.2.1.

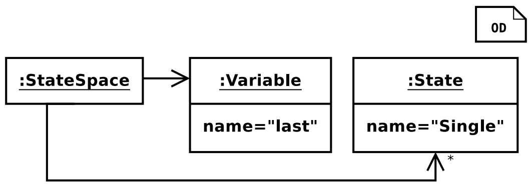

Fig. 9: Example *StateSpace* of a single *State* „Single“ and *Variable* called „last“. The OD is based on CD in Fig. 5 and SD in Sect. 4.2.1.

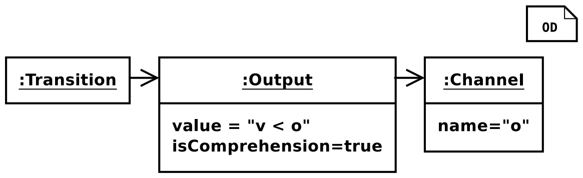

Fig. 12: Example of an *Output* of a *Transition* with set comprehension enabled. A *StateSpace* with one natural number variable v and an equally typed output o is assumed. A *Transition* that can output any natural number larger than v's current value on port o is then modeled. Intermediary representation of such an *Output* is depicted in Fig. 12. *isComprehension* flag is activated. *value* then specifies a set of values for o via the constraint "$v < o$". Such a technique allows modeling of infinitely many transitions.

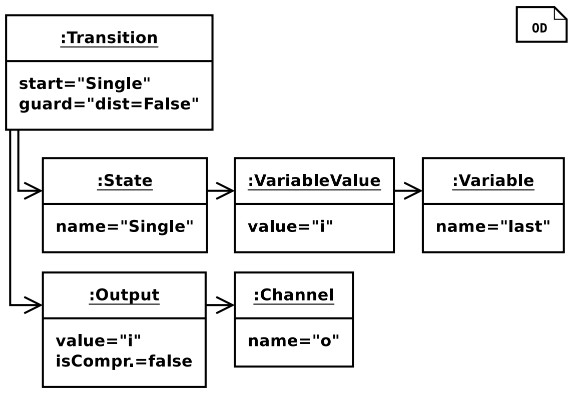

Fig. 10: Example *Transition* based on the CD from Fig. 6 and SD from Sect. 4.2.1. The start and target state are „Single“. The *guard* makes sure no disturbance *dist* occurred. The *Variable* „last“ is set to „i“, as is the *Channel* „o“.

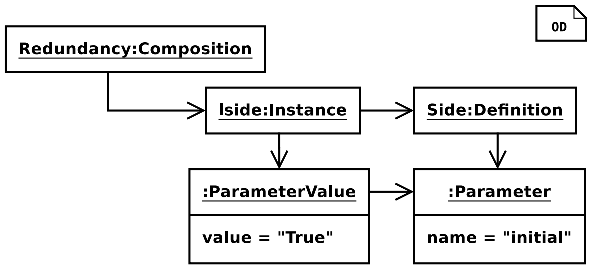

Fig. 11: Partial composition OD. Shows only the first *Instance* of PFS (ref. Sect. 4.2) and omits datatype mappings.

F Generated Isabelle Functions and Theorems

```
(* parameters ⇒ state ⇒ input ⇒ bool *)
fun guard::"bool ⇒ State ⇒ (bool×bool) ⇒ bool" where
"guard initial (s last) (dist, i) = (s=Single ∧ dist=False)"
```

List. 7: Guard in Isabelle. Comments are enclosed in *(*...*)*. *fun* keyword starts a function declaration, followed by an optional name (*guard*) and colon-separated signature declaration. Keyword *where* starts the left hand side of the function definition. The name *guard* is repeated, followed by positionally matched handles for all parameters. The right hand side starts after the equals sign.

```
(* parameters ⇒ state ⇒ input ⇒ (state × output) set *)
fun result::"... ⇒ (State×...) set" where
"result initial (s last) (dist, i) = { ((Single i), i) }"
```

List. 8: Result of a transition in Isabelle. The input types are equal to the *guard* in List. 7. The output type is a set of tuples instead. Each of those tuples holds a target *State* and values for each output port.

```
fun trans::"bool ⇒ State ⇒ bool ⇒ (State×...) set" where
"trans initial (s last) (dist, i) = if (guard initial (s last) (dist, i))
 then (result initial (s last) (dist, i)) else {}"
```

List. 9: The transition aggregates guard and result. The *if...then ...else...* directive is used for case distinction.

```
fun result::"... ⇒ (State×nat) set" where
"result _ (Single v) _ = { ((Single v), o) | o. v < o }"
```

List. 10: Set-based builder notation (set comprehension) for potentially infinitely large Transitions in Isabelle. Redundant inputs (parameters, inputs from ports) are omitted (_). We use a placeholder output o and quantify it (after the |) using a dot to separate to-be-quantified variables (o) and the predicate ($v < o$).

```
(* parameters ⇒ input ⇒ output ⇒ bool *)
fun pred::"bool ⇒ (...×(bool×bool stream) ⇒ (bool×bool stream) ⇒ bool" where
"pred initial (dist, ts, ospf) pf = initial →(fst (snth 0 ospf)) ∧ ..."
```

List. 11: Predicates in history-based specifications in Isabelle. Predicate uses implication (arrow to the right). Whenever *initial* is set to true, the right hand side follows. Right hand side states that first (*fst*) element of the tuple at the initial time slice *0* (*snth* is the Isabelle function for accessing the nth element of a stream) in the *ospf* stream is true.

```
fun sps::"... ⇒ (I^Ω →O^Ω) set" where
"sps initial = {spf | spf. ∀input.
                    pred initial (getInputs input) (getOutputs (spf input)) ∧...}"
```

List. 12: Set of SPFs in Isabelle. Ω denotes the stream bundles as mentioned. Input streams are quantified inside a bundle *input* and then extracted using appropriate getter functions (*getInputs*). Outputs are created by applying the *spf* to the *input* bundle and retrieving the outputs (*getOutputs*) from the resulting bundle.

```
(* parameters ⇒ (SPS:Input → Output) *)
fun sps::"unit ⇒ (I^Ω →O^Ω) set" where
"sps _ = lside.sps True ⊗ lrbus.sps True ⊗ rside.sps False ⊗ rlbus.sps False"
```

List. 13: Composing the components into the overall Pilot Flying System in Isabelle. The *Instances*' SPSes are accessed via dot notation, i.e. *lside.sps* for the *sps* of *lside*.

```
theorem refinement: "Bus.sps initial ⊆ UnreliableBus.sps initial"
```

List. 14: Refinement in Isabelle. The theorem states that the behavior of a refined component is a subset of the behavior of the original.

```
theorem SysReq:
  assumes "f ∈ PFS.sps"
    shows "lside_initially_active f ∧ ¬rside_initially_active f"
  <proof>
```

List. 15: Verifying the system specification against a system requirement in Isabelle. The first assumption (*assumes*) extracts a single behavior *f* from the SPS. The statement (*shows*) then states that exactly one side has to be active initially.