[KPRR20] H. Kausch, M. Pfeiffer, D. Raco, B. Rumpe:
MontiBelle - Toolbox for a Model-Based Development and Verification of Distributed Critical Systems for Compliance with Functional Safety.
In: AIAA Scitech 2020 Forum, American Institute of Aeronautics and Astronautics, Orlando, Jan. 2020.
www.se-rwth.de/publications/

MontiBelle - Toolbox for a Model-Based Development and Verification of Distributed Critical Systems for Compliance with Functional Safety

Hendrik Kausch, Mathias Pfeiffer, Deni Raco, and Bernhard Rumpe
RWTH Aachen University, Aachen, Germany

The methodology MontiBelle represents a specification formalism designed for the development and verification of cyber-physical safety-critical systems. It allows abstraction and underspecification and provides an extensive set of verification techniques to actually prove specification correctness. An architecture description language for modeling Cyber-Physical Systems is provided and combined with behavioral specifications as well as an environment to formulate desired properties. ADL Models and specifications are translated into an equivalent specification in the theorem prover Isabelle. Counterexamples can also be found by using a simulation execution within the Isabelle prover and letting the counterexample-finder run in an intelligent way through execution paths which are considered erroneous candidates. The most important property making this methodology stand out among alternate approaches is that refinement is fully compositional. The approach is evaluated on a synchronous flight guidance system.

I. Introduction

Requirements Engineering is an important activity in software development. Many errors are identified in that phase [1]. For this, DO-178C [2] emphasizes that requirements are to be developed in a hierarchically decomposable way, communications and interrelationships among components should be unambiguously described, test coverage should be sufficient [3], etc.

When specifying requirements, usually a compromise needs to be found between completeness, detailedness and formality on one side and abstraction, focus on essential structure and behavior on the other side. However, rigor and level of detail need not be excluding each other. Explicit notions of the specification, refinement and hierarchical decomposition as well as various techniques for merging functional specifications from individual features allow to combine both.

The methodology MontiBelle presents such a specification formalism, designed for the development and simulation-based verification of critical systems in avionics, automotive, etc. to provide compliance with functional safety considerations (DO-178C or ISO 26262). It allows abstraction and underspecification, while retaining full formality and therefore provides an extensive set of verification techniques to actually prove specification correctness. An architecture description language [4] for modeling Cyber-Physical Systems (which have been considered as important for aeronautics in the future [5]) is provided and combined with behavioral specifications as well as an environment to formulate desired properties. All can be defined directly in the specification framework or can be mapped from the ADL, state machines and a logic language like OCL [6]. All (formal) requirement models and specifications are translated into an equivalent specification in the Isabelle theorem prover [7]. Many properties can be proven correct at the push of a button using the provided extensive proof framework as presented in [8] and shown in the video in [9] demonstrating an early version of the methodology.

However, to find inconsistencies, counterexamples can be searched and found. This is done by using a simulation execution within the Isabelle prover and letting the counterexample-finder run in an intelligent way through execution paths which are considered erroneous candidates. Through simulation, the fault behavior caused by the corresponding input can be reliably identified and debugging information is delivered. This technique is a mixture between traditional (exemplaric) simulation and a full blown verification and usually allows to detect errors rather easily, especially when evolving requirements specifications. However, due to its symbolic nature, it does not scale to traditional fully exemplaric simulation, which on the other hand also is not necessary.

The behavioral semantics [10] are defined using sets of stream processing functions from the FOCUS methodology [11, 12]. Refinement of component specifications is semantically well reflected by the concept of set inclusion between

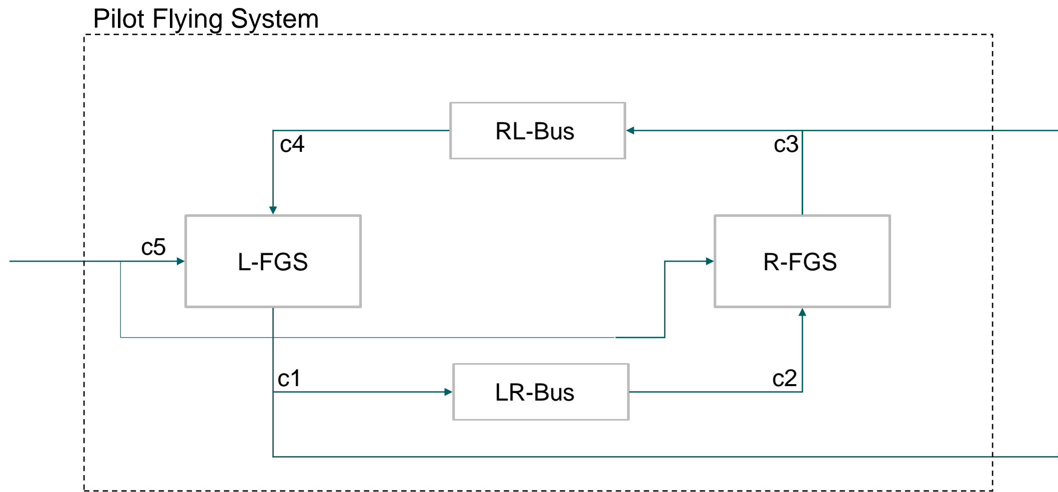

Fig. 1 Synchronous Pilot Flying System white box

function sets. The most important property making this methodology stand out among alternate approaches is that **refinement of a component in a decomposed structure leads automatically to refinement of the composition** [10]. This property is important, because the developers only need to deal with refinement in the small and are sure that the overall system is refined by construction as well. This is the main reason why the methodology scales to larger and complex distributed systems, such as airplanes.

To summarize, MontiBelle offers:

1) Formal, yet abstraction-scalable specification.
2) Symbolic simulation of behavior embedded in their decomposed architectures
3) Formal Verification
4) Simulation-based fault detection and debugging
5) Timing, structural decomposition and behavioral underspecification
6) Interactions with environment (such as pilots, foreign airplanes or ground control)

The rest of the paper is organized as follows: the next section presents a running example from the avionics and the verification challenges. Then the tool chain presents the way it deals with the specification and verification activities. A conclusion summarizes the benefits of the methodology. For the interested reader, details of the models and the core structured formalized in the theorem prover can be found in the appendix.

II. Running example

We describe the verification of development steps through a representative case study adapted from an openly available NASA report [13]. A synchronous electronic pilot flight system PFS with flight guidance components for each pilot can be modeled as a composed feedback system visualized in fig. 1, where each channel transmits boolean messages.

An input transmitted over channel $c5$ represents the transfer switch status. If pressed, a switch to the active flying side is requested. Each of the two flight guidance systems FGS has an activeness state that denotes the active flying side. The Bus components transmit messages between the FGS components. The complete system should fulfill the formalized system requirements. System requirements could be:

- At least one flight guidance system is active

L-FGS

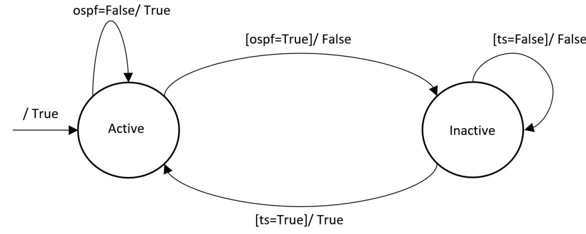

Fig. 2 Synchronous Left Flight Guidance System automaton

- Flipping the transfer switch, switches the active flying side

A possible component design to fulfill the requirements is given in textual form in appendix A. Informally, the Bus components perform a delayed (used as known for handling feedback loops in hardware modeling) transmission by sending out an initial output message and then behaving as identity components. The Left-FGS for instance can be designed as an automaton in fig. 2, and one can show that it complies with the requirements. The Right-FGS is defined in an analogous matter, being initially inactive and therefore outputting $False$ in its initial step.

Asynchronous PFS The synchronous PFS can be extended by non-deterministic clocks for each component to model sensor failure or component malfunctioning by restricting the FGS and Buses to only behave correctly in any step, iff the corresponding clock outputs $True$ in that step. This asynchronous PFS is also adapted from the asynchronous PFS in [13].
Without any restrictions to the clocks behavior, the complete system might never react to anything and never react to any input. By refinement, the clocks can be restricted to a $fair$ behavior such that once in a while they emit a $True$. The MontiArc models of the fair and unfair clock are in appendix A.

A. Simulation-based verification

Simulation-based verification can be performed by writing OCL properties in form of test-cases where all variables are instantiated. From such an OCL expression an equivalent theorem in Isabelle is generated. Automatic provers are applied to each of the test-cases to check these for correctness. This method is applied to all defined OCL properties, thus an automatic verification is possible in many cases.
An example would be to test that either the left or the right side is active in the second step. The OCL property can be formulated like this:

$$c1[2] = true\ OR\ c3[2] = true$$

An equivalent theorem is generated and the prover tools are automatically applied. The main structure of the verification framework is described in appendix A.
One could extend a test-case by quantified variables and perform full formal verification:
SysReq: FORALL n IN nat. c1[n] OR c3[n];
Syntax: The inner OCL expression consists of two clauses and the logical *or* operator. By $c1[n]$, the nth element of the stream flowing on channel $c1$ is donated.
Semantics: For every step in the system execution, either the left or the right FGS has to be active. Thus, it checks if at least one side is pilot flying side.
An equivalent theorem is generated in Isabelle and automatic solvers are called.

Counterexamples In case of an unfulfilled requirement, counterexample tools like quickcheck and nitpick [8] check the property and search for a counterexample assignment of undefined variables/symbols. This leads to an early error-detection and thus fewer costs.

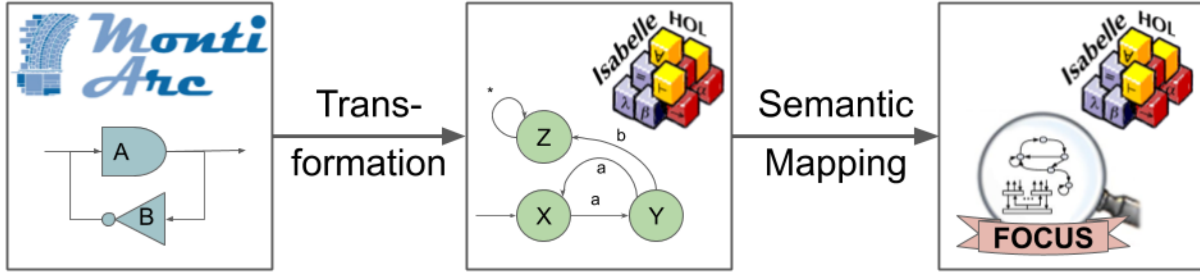

Fig. 3 Verification Tool chain with MontiArc frontend and Isabelle backend.

```
// List of states
state Start, Next, Stop;

// Initial state including initial output
initial Start / {o = false};

// Transition from "Start" to "Next" if guard satisfied, outputs "true"
Start -> Next [counter > 0] / {o = true};
```

Fig. 4 Exemplary states and transitions of a MontiArc automaton.

III. Tool Chain

The tool chain is separated in frontend and backend stages. Figure 3 shows an overview. The frontend allows to model architecture and behavior using MontiArc [14] as well as desired properties using the Object Constraint Language (OCL) [15]. The backend can be further separated into two parts. Firstly, MontiArc and OCL specifications are transformed into Isabelle theories. For architecture and behavior, this is achieved by a mapping to equal concepts encoded in Isabelle. The OCL invariants are mapped to lemmas. Secondly, the encoding is mapped to its semantic [16]. This semantic mapping is completely implemented in Isabelle and as such mathematically correct. The final result is (a set of) FOCUS stream processing functions (SPFs). The complete tool chain is supported by Integrated Verification Environment (IVE), combining modeling, transformation, and verification in one single tool. The tool is designed to be easily qualifiable itself by having the majority of code generation being performed in Isabelle (this is achieved by using genericity concepts such as locals [17]). Isabelle code is easily certified due to its axiomatic and conservative nature [18]. The part of the generator from the ADL into the prover is thus kept at a minimum.

A. MontiArc

MontiArc is a textual, high-level domain specific language (DSL) developed at RWTH Aachen [14]. It allows specifying realizable-by-construction components by mixing a component-connector architecture description language (ADL) with formal behavior specification. The language also supports non-determinism by underspecification. Each component defines an interface using typed ports. Components can be instantiated and reused. Atomic components have their behavior defined either using automata or by (potentially recursive) equations relating infinite input/output streams (history-oriented) [8, 19]. Automata mainly consist of states and transitions as shown in fig. 4. Recursive stream specifications allow to declaratively specify the desired relations between inputs and outputs in a higher-level style closer to informal natural language and shall-statements, without explicitly modeling the implementation.

B. OCL

OCL is part of the Unified Modeling Language (UML) [20]. It is designed to define constraints on objects modeled by other UML languages such as class diagrams (CDs) and object diagrams (ODs). The OCL implementation developed at RWTH Aachen [15] was extended to work with MontiArc models. This extension enables the specification of desired properties of MontiArc instances.

C. IVE

IVE is a purpose built VSCode [21] plugin to support the complete toolchain. The software allows seamless combination of all stages through syntax highlighting, syntax checks, reference checks, sensible overview of files, Isabelle integration, and graphical representations. An example view of the interface is given in appendix A.

IV. Conclusion

This paper presented MontiBelle, a framework for reasoning over cyber-physical systems. It addresses the challenge that due to increasing complexity of avionics software, traditional verification methods on informal requirements such as reviews, while keeping reasonable certification costs, scale no longer. So by presenting a formal textual ADL for specifying requirements coupled with a reasoning infrastructure, a number of traditional verification activities can be replaced or complemented.

References

[1] Feiler, P. H., "Model-based validation of safety-critical embedded systems," *2010 IEEE Aerospace Conference*, IEEE, 2010, pp. 1–10.

[2] DO, R., "178C, Software Considerations in Airborne Systems and Equipment Certification. RTCA," *Inc., Washington, DC, USA (December 1992)*, 2011.

[3] Paz, A., and Boussaidi, G. E., "A requirements modelling language to facilitate avionics software verification and certification," *Proceedings of the 6th International Workshop on Requirements Engineering and Testing*, IEEE Press, 2019, pp. 1–8.

[4] Haber, A., Ringert, J. O., and Rumpe, B., "Montiarc-architectural modeling of interactive distributed and cyber-physical systems," *arXiv preprint arXiv:1409.6578*, 2014.

[5] Ulbig, P., Müller, D., Torens, C., Insaurralde, C. C., Stripf, T., and Durak, U., "Flight Simulator-Based Verification for Model-Based Avionics Applications on Multi-Core Targets," *AIAA Scitech 2019 Forum*, 2019, p. 1976.

[6] Rumpe, B., *Agile Modellierung mit UML: Codegenerierung, Testfńlle, Refactoring*, Springer-Verlag, 2012.

[7] Nipkow, T., Paulson, L. C., and Wenzel, M., *Isabelle/HOL: a proof assistant for higher-order logic*, Vol. 2283, Springer Science & Business Media, 2002.

[8] Kriebel, S., Raco, D., and Rumpe, B., "Model-Based Engineering for Avionics: Will Specification and Formal Verification e.g. Based on Broy's Streams Become Feasible?" *[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, 2019, pp. 87–94. URL `https://publications.rwth-aachen.de/record/758226`.

[9] Niu, S. C., Pfeiffer, M., Raco, D., and Rumpe, B., "Model-based Verification of Distributed Systems at the Push of a Button," 2019. URL `https://www.youtube.com/watch?v=krl4Q7MAAlo`.

[10] Harel, D., and Rumpe, B., "Meaningful modeling: what's the semantics of" semantics"?" *Computer*, Vol. 37, No. 10, 2004, pp. 64–72.

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

[12] Rumpe, B., *Formale Methodik des Entwurfs verteilter objektorientierter Systeme*, Herbert Utz Verlag, 1996.

[13] Cofer, D., and Miller, S. P., "Formal methods case studies for DO-333," 2014.

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

[15] Maoz, S., Mehlan, F., Ringert, J. O., Rumpe, B., and von Wenckstern, M., "OCL Framework to Verify Extra-Functional Properties in Component and Connector Models," 3rd International Workshop on Executable Modeling, Austin (USA), 18 Sep 2017 - 18 Sep 2017, 2017, p. 7 Seiten. URL `http://publications.rwth-aachen.de/record/713152`.

[16] Rumpe, B., "Formale Methodik des Entwurfs verteilter objektorientierter Systeme," Ph.D. thesis, Zugl.: München, Techn. Univ, Zugl. München, 1996.

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

[18] Cofer, D., Klein, G., Slind, K., and Wiels, V., "Qualification of Formal Methods Tools (Dagstuhl Seminar 15182)," *Dagstuhl Reports*, Vol. 5, No. 4, 2015, pp. 142–159. https://doi.org/10.4230/DagRep.5.4.142, URL http://drops.dagstuhl.de/opus/volltexte/2015/5354.

[19] Broy, M., Dederichs, F., Dendorfer, C., Fuchs, M., Gritzner, T. F., and Weber, R., *The design of distributed systems: An Introduction to FOCUS*, Citeseer, 1992.

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

[21] Microsoft Corporation, *VSCode*, Redmond, WA, 2019. URL https://code.visualstudio.com/.

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

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

[24] Paulson, T. N. L. C., and Wenzel, M., "A Proof Assistant for Higher-Order Logic," , 2013.

A. MontiBelle

A. MontiArc Models

In this section, different textual MontiArc models describing the components mentioned in this paper are given and explained.

Textual Parametric Bus MontiArc Model Here, a history-oriented specification (spec) and a state-oriented specifcation (automaton) is presented. The spec describes the behavioral connection between complete output and input streams. The parameter initialSide makes it possible to reuse the same model for both Buses.

```
component Bus(boolean initialSide) {
timing causalsync;

  port
    in boolean in,
    out boolean out,
    //The Bus component has an input and output port

  spec Bus {
    out[0]=initialSide;
    //The initial output is defined by the initialSide parameter

    out[n+1]=in[n];
    // It then behaves like a delayed identity component
   }
}
```

The automaton describes the element-wise output calculation and constitutes an implementation per construction.

```
component Bus(boolean initialSide) {
timing causalsync;

  port
    in boolean in,
    out boolean out,
```

```
    //The Bus component has an input and output port

  automaton Bus {
    state Single;
    initial Single / {out=initialSide};
    //The initial Output is defined by the initialSide parameter

    Single -> Single / {out=in};
    // The transition outputs its input element
  }
}
```

Textual Parametric FGS MontiArc Model Two specification variants are provided. Again, the parameter initialSide is used to define both FGS within one MontiArc model.

```
component Side(boolean initialSide) {
timing sync;

 port
    in boolean ts,
    //port receiving the state of the transfer switch
    in boolean ospf,
    //port receiving the (delayed) output of the other FGS via a Bus
    out boolean pf;
    //output port representing the activeness of the FGS

  spec Side {
    let active[0] = initialSide AND
        active[n+1] = pf[n];
    // helper definition to check whether the side is active or inactive

    let pre_ts[0]=True AND
        pre_ts[n+1]=ts[n];
    // helper definition to access the transfer switch state in the last step

    ((NOT active[n]) AND ts[n] AND (NOT pre_ts[n]))IMPLIES pf[n];
    //Switches to active, if the transfer switch is pressed
    ((NOT active[n]) AND (NOT ts[n]) OR pre_ts[n]) IMPLIES (NOT pf[n]);
    //Does not switch active, if the transfer switch is not pressed or still pressed
    (active[n] AND ospf[n]) IMPLIES (NOT pf[n]);
    //Switches inactive, if the other side switched active
    (active[n] ABD (NOT ospf[n])) IMPLIES pf[n])
    //Stays active as long as the other side is inactive
  }
}
```

The automaton defined in the model behaves exactly like the specification.

```
component Side(boolean initialSide) {
timing sync;

 port
    in boolean ts,
        //port receiving the state of the transfer switch
    in boolean ospf,
```

```
    //port receiving the (delayed) output of the other FGS via a Bus
    out boolean pf;
        //output port representing the activeness of the FGS

  boolean pre_ts;

  automaton Side {
    state Active, Inactive;
    initial (if initialSide then Active else Inactive) / {pre_ts=true};

    Active -> Inactive [ospf] / {pre_ts = ts, pf=false};
    Active -> Active [NOT ospf] / {pre_ts = ts, pf=true};

    Inactive -> Active [ts AND NOT pre_ts] / {pre_ts = ts, pf=true};
    Inactive -> Inactive [(NOT ts) OR pre_ts] / {pre_ts = ts, pf=false};

  }
}
```

Having modeled all sub-components of the PFS, the complete system can be modeled by connecting the sub-components.

```
<<deterministic>>component PilotFlyingSystem {
timing sync;

  port
    in boolean ts,
    //transfer switch input port
    out boolean lpf,
    //left side is active output port
    out boolean rpf;
    //right side active output port

  component Side(true) lside;
  //left side FGS component
  component Bus(true) lrbus;
  //Bus connecting left to right side FGS
  component Side(false) rside;
  //right side FGS component
  component Bus(false) rlbus;
  //Bus connecting right to left side FGS

  connect ts -> lside.ts;
  //connecting transfer switch input to L-FGS input (channel c5)
  connect ts -> rside.ts;
  //same to the right side (channel c5)
  connect lside.pf -> lrbus.ls;
  //connecting L-FGS output to LR-Bus (channel c1)
  connect lrbus.rs -> rside.ospf;
  //connection LR-Bus output to R-FGS input (channel c2)
  connect rside.pf -> rlbus.rs;
  //connecting R-FGS output to RL-Bus (channel c3)
  connect rlbus.ls -> lside.ospf;
  //connecting RL-Bus output to L-FGS input (channel c4)
  connect lside.pf -> lpf;
```

```
  //connecting L-FGS output to a output port (output represents activeness)
  connect rside.pf -> rpf;
  //connecting LRFGS output to a output port (output represents activeness)

}
```

The complete system was visualized in fig. 1.

Textual Unfair Clock MontiArc Model Two specification variants are provided.

```
component ClockUnfair {
timing causalsync;

  port
    out boolean clk;

  spec ClockUnfair {
    LEN clk= INF;
  }

  automaton ClockUnfair {
    state Single;
    initial Single;

    Single->Single / {clk=true};
    Single->Single / {clk=false};

  }
}
```

Textual Fair Clock MontiArc Model Two specification variants are provided.

```
<<transRefines="ClockUnfair">>component ClockFair {
timing sync;

  port
    out boolean clk;

  spec ClockFair {
    LEN clk = INF;
        FORALL n IN nat. EXISTS m IN nat. m<n+10 AND out[m]=true;
  }

  int counter;

  automaton ClockFair {
    state Single;
    initial Single / {counter=rand{j. j<=10}};

    Single -> Single [counter>0] / {clk=false};
    Single -> Single [counter==0] / {clk=true, counter=rand{j. j<=10}};

  }
}
```

B. IVE

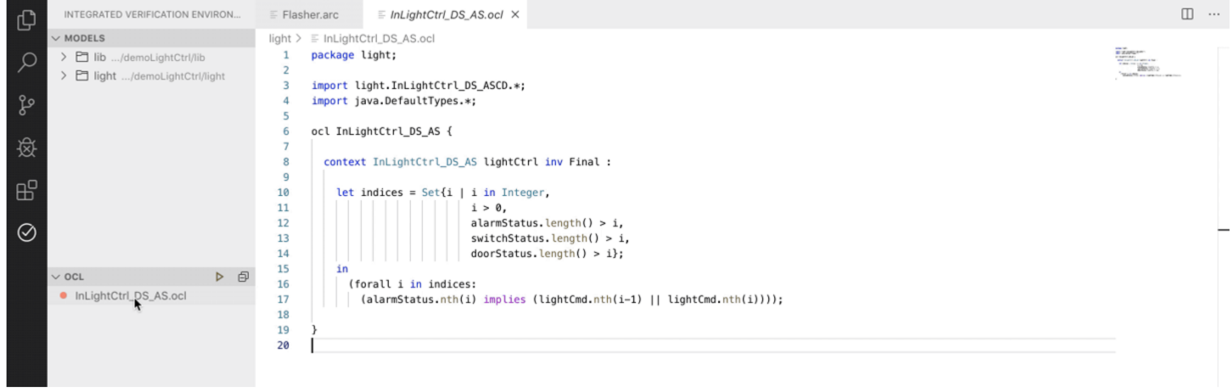

Fig. 5 Integrated Verification Environment

C. Fundamentals of the encoded stream-based framework

Based on the encoding of streams in [8, 22],

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

the following datatypes lead to a user-friendly and efficient framework regarding, length, proof-length, and tool support.

For composing stream processing functions, we add labeled channels [23] and code the structure "stream bundles" (SB) from [16].
Every channel of the stream bundle has a set of allowed messages it can transmit. The predicate sb_well is true, if all channels only transmit allowed messages. The domain is defined by a parametric type variable c and is visible in the signature of every SB.

```
typedef 'c SB("(_Ω)")
          = "{f::('c ⇒M stream). sb_well f}"
```

A fixpoint operator can be defined over the SB type. It composes two continuous functions over SBs. The operator is defined over a recursive fixpoint and → in the signature denotes that a function is continuous [22]. The Ω abbreviation for bundles was introduced to the framework while defining SBs.

```
definition spfComp::"('I1Ω →'O1Ω) →('I2Ω →'O2Ω)
 → ((('I1 ∪ 'I2) − ('O1 ∪ 'O2))Ω →('O1 ∪ 'O2)Ω)"
```

A stream processing function SPF is a continuous function that maps input SBs to output SBs. Since the domains of SBs are visible in the signature, the same holds for the domain and range of SPFs. Since specified components can be modeled as SPFs, a composed system of components can be defined with the composition operator $spfComp$. Again, the used SPF type is taken from [16].

```
type_synonym ('I,'O) SPF = "{f::('IΩ →'OΩ) }"
```

A specification (SPS) is defined as a set of SPFs. By defining it as a type synonym, all predefined and proven properties over general sets automatically hold over the SPS type. SPS represents the semantics of non-deterministic components. The composition operator can be lifted straightforwardly to sets of functions.

```
type_synonym ('I,'O) SPS = "('I,'O) SPF set"
```

Automatons can be used to model (under-)specified components. Here we mention deterministic automatons, which can be used for describing the behavior of components. By the abbreviation $\surd$ we denote a SB element (a *vector* of messages). The data type definition uses *record* explained in [24].

```
record ('state, 'in, 'out) dAutomaton =
  daTransition :: "'state ⇒ 'in√ ⇒('state × 'outΩ)"
  daInitState  :: "'state"
  daInitOut    :: "'outΩ"
```

The semantics of a deterministic automaton is a SPF. The semantics of a non-deterministic automaton is a SPS. Semantical mappings are implemented according to [16].

```
definition daSem::"('S, 'I,'O) dAutomaton ⇒ ('IΩ →'OΩ)"
```

Locales are introduced in [17] and were used in MontiBelle to relocate most of the generator inside the theorem prover (instead of largely generating from the ADL), thus the code generator is easier qualifiable.