F-IDE 2019 Pre-prints

September 24, 2019

Preface

This volume contains the pre-prints of the papers presented at F-IDE 2019: 5th Workshop on Formal Integrated Development Environment held on October 7, 2019 in Porto as part of FM week. It has been generated with the easychair tools, which supported the overall review process of F-IDE. Revised versions of these papers will be published in the EPTCS series after the workshop.

The aim of the F-IDE workshop is to provide a forum for presenting and discussing research efforts as well as experience returns on design, development and usage of formal IDE aiming at making formal methods ”easier” for both specialists and non-specialists. There were 8 submissions. Each submission was reviewed by at least 3 program committee members. The committee decided to accept all 8 papers. The program also includes 1 invited talk.

September 23, 2019
Gif Sur Yvette Cedex

Virgile Prevosto
Rosemary Monahan
Jose Proenca

Program Committee

Cinzia Bernardeschi	University of Pisa
José Creissac Campos	University of Minho & HASLab/INESC TEC
Paul Curzon	Queen Mary University of London
Damien Doligez	INRIA
Andrea Domenici	Università di Pisa
Carlo A. Furia	USI - Università della Svizzera Italiana
Kenneth Lausdahl	Aarhus University
Stephan Merz	Inria Nancy
Stefan Mitsch	Carnegie Mellon University
Rosemary Monahan	Maynooth University
Yannick Moy	AdaCore
Andrei Paskevich	Université Paris-Sud, LRI
François Pessaux	ENSTA ParisTech
James Power	Maynooth University
Virgile Prevosto	CEA Tech List
Jose Proenca	CISTER-ISEP and HASLab-INESC TEC
Steve Reeves	University of Waikato
Bernhard Rumpe	RWTH Aachen University
Claudio Sacerdoti Coen	University of Bologna
Silvia Lizeth Tapia Tarifa	University of Oslo
Mattias Ulbrich	Karlsruhe Institute of Technology
Laurent Voisin	Systerel
Makarius Wenzel	sketis.net
Yi Zhang	US FDA

Additional Reviewers

Pessaux, François
Wachtmeister, Louis
Weigl, Alexander

Table of Contents

What is KeY's key to software verification? 9
Wolfgang Ahrendt . . . .
Experience Report: Towards Moving Things with Types - Helping Logistics Domain Experts to Control Cyber-Physical Systems with Type-Based Synthesis 11
Jan Bessai, Moritz Roidl and Anna Vasileva . . . .
Automated deductive verification for Ladder programming 17
Denis Cousineau, David Mentré and Hiroaki Inoue . . . .
Deeply Integrating C11 Code Support into Isabelle/PIDE 23
Frédéric Tuong and Burkhart Wolff . . . .
A component-based formal language workbench 39
Peter Mosses . . . .
An Integrated Development Environment for the Prototype Verification System 45
Paolo Masci and Cesar Munoz . . . .
The TLA+ Toolbox 60
Markus Alexander Kuppe . . . .
Simulation under arbitrary temporal logic constraints 75
Julien Brunel, David Chemouil, Alcino Cunha and Nuno Macedo
Tool Support for Validation of Formal System Models: Interactive Visualization and Requirements Traceability 81
Eduard Kamburjan and Jonas Stromberg . . . .

Keynote Speaker

Speaker: Wolfgang Ahrendt, Chalmers University of Technology, Sweden

Title: What is KeY's key to software verification?

Abstract: KeY is a deductive software verification approach and system, whose most elaborate version targets Java programs. In a recent KeY case study, which attracted attention also outside formal method circles, verification with KeY could reveal a bug in the main sorting routine of OpenJDK. While this talk will also cover the user interface of KeY, the focus of the discussion is more fundamental. KeY follows to a signifiant extent principles which are different from other deductive verification systems, on the level of the program logic, the proof calculus, the interaction with the prover, the transparency of proofs, and the usage of back-end solvers. In this talk, I will discuss the impact of these aspects, with a special focus on usability. In addition, we will look at how the design of the logic and calculus influenced the integration with other validation techniques, like test generation and runtime verification.

Experience Report: Towards Moving Things with Types – Helping Logistics Domain Experts to Control Cyber-Physical Systems with Type-Based Synthesis

Jan Bessai Moritz Roidl Anna Vasileva
Technical University of Dortmund
Dortmund, Germany
jan.bessai, moritz.roidl, anna.vasileva@tu-dortmund.de

One of the ultimate goals of software engineering is to leave virtual spaces and move real things. We take one step toward supporting users with this goal by connecting a type-based synthesis algorithm, Combinatory Logic Synthesizer ((CL)S), and its IDE to a logistics lab environment. The environment is built and used by domain experts, who have little or no training in formal methods, and need to cope with large spaces of software, hardware and problem specific solution variability. It consists of a number of Cyber-Physical Systems (CPS), including wheel-driven robots as well as flying drones, and it has laser-based support to visualize their possible movements. Our work describes results on an experiment integrating the latter with (CL)S. Possibilities and challenges of working in the domain of logistics and in cooperation with its experts are outlined. Future research plans are presented and an invitation is made to join the effort of building better, formally understood, development tools for CPS-enabled industrial environments.

1 Introduction

Logistics is the science (and sometimes art) of moving things. Connections to computer science are immediate: paths must be planned, resources scheduled, object positions tracked, and automatization demands software systems to control physical actors. Traditionally, logistics is an engineering discipline. As such, it has an unspoken predisposition toward algorithms that use numeric methods for optimizing highly context-specific parameters. Examples include Dijkstra's algorithm [7] for finding optimal paths and Kalman Filters [13] for estimating the position of moving objects. However, symbolic formal methods, which are the subject of large parts of computer science, are rarely transferred to the field. Yet, it has all the potential for their application. Systems scale from small packaging stations to country spanning interdependent logistical networks, while fine-tuned optimized numbers rarely scale at all. Hard- and software require consistent reasoning across daily evolving product lines of logistical equipment. Pareto-optimal solution spaces can be vast or even infinite, in which case they need symbolic representations. Finally, just to name a few of the potentials, formal guarantees can prevent catastrophic failures and thereby not only safe money but also lives, if systems are operated by humans. This paper is part of a collaborative effort at the Technical University of Dortmund to identify and overcome reasons for the lack of transfer. The authors are a mixed team of researchers in computer science (Bessai, Vasileva) and logistics (Roidl). Starting point of the investigation is an experiment, which uses type-based synthesis to find ways through a labyrinth. The scenario has been previously studied to illustrate progress on an IDE for the Combinatory Logic Synthesis, (CL)S, Framework [6]. It is described in Sec. 2. The problem of finding ways is sufficiently close to logistics to provide a starting point and to be practically evaluated in a logistics test lab, which is described in Sec. 3. Conducting the experiment required extensions to existing software systems and it quickly became clear that new ways of visualizing solutions would be required to cater domain expert needs. Extensions and results obtained this way are described in Sec. 4.

Submitted to:
F-IDE 2019

© J.Bessai & M. Roidl & A. Vasileva
This work is licensed under the
Creative Commons Attribution License.

They go beyond what was considered in the domain-independent design of the IDE previously developed for (CL)S. Finally, some future plans and lessons learned from the ongoing cooperation are discussed in Sec. 5. We identify some key technical principles to make collaborations easier, and also focus on the social aspects, which can help to transfer knowledge. The discussion of future plans includes an invitation to join our practical efforts, potentially by testing and improving other tools with us.

2 Synthesizing Robot Paths with Types

The implementation of (CL)S provides a framework for the automatic composition of software components from domain-specific repositories [5, 10, 16]. Automatic composition is performed by answering the type inhabitation problem of relativized Combinatory Logic with intersection types [14] $\Gamma \vdash ? : \tau$. That is, given a repository of typed combinators Γ, find all combinatory terms M (inhabitants) of the goal type τ. The (CL)S Framework is implemented in Scala. It is meant to be a tool for programmers as well as engineers with knowledge in programming who are not necessarily familiar with type theory.

In order to improve usability of the framework, we developed a web-based IDE [2, 6]. It is specifically focused to improve comprehensibility and traceability of the inhabitant search process. The IDE provides a graphical overview of the inhabitants generated by the algorithm in form of hypergraphs [8, 12]. We developed a step-wise build of the hypergraphs in order to explain the generation of solutions. The web IDE also emits warnings if there are unused combinators or uninhabited types. This exposes problematic specifications, which can be further analyzed in a perspective for presenting domain-specific repositories.

Additionally to the perspectives outlined above, which were previously discussed in [6], we developed features that focus on presenting local rather than global properties of solutions. A perspective of the web IDE provides a list of inhabitants, s.t. a user can inspect each inhabitant and the corresponding hypergraph separately. Intersection types allow combinators to have more than one type. For instance, if we have a combinator:

$$down : (MovementPlan \to MovementPlan) \cap (Pos(0,0) \to Pos(1,0)) \cap (Pos(1,2) \to Pos(2,2))$$

it can be used to go from position $(0,0)$ to $(1,0)$ and from position $(1,2)$ to $(2,2)$, while transforming a plan of movements to a plan of movements. Inhabiting a type such as $MovementPlan \cap Pos(2,2)$ requires selecting components form the type in a process called covering. The IDE provides help to understand and debug this process for a given combinator and target, which is especially useful if combinator types are underspecified and lack required components.

We also developed a filtering function based on satisfiability modulo theories (SMT) [12]. In this way, additional constraints can be used to restrict the set of inhabitants and avoid trivial solutions such as $reverse(reverse(s))$ for some sequence s. The IDE supports adding and removing some domain-independent structural constraints, which are helpful in many situations.

	0	1	2
0			
1		■	•
2			
3	■	★	■

Figure 1: Labyrinth example

The synthesis of robot paths is based on the labyrinth example [6], an instance of which is shown in Fig. 1. The size of labyrinths, blocked paths (indicated by black boxes), and start (indicated by a black dot) as well as end positions (indicated by a star) are user-defined. In Fig. 1, we have 3×4 labyrinth with start position $Pos(1,2)$ and goal position $Pos(3,1)$. The repository Γ includes combinators corresponding to the allowed move directions ($up, down, left$ and $right$), a combinator that provides the starting position as well as their type descriptions. The types in the repository represent all valid one-step moves.

3 Logistics Research Lab

The research lab is designed for rapid prototyping of CPS [4]. It is situated in an existing lightweight construction building that is similar to common industrial buildings used in logistics operations. It follows the basic concept of a highly flexible development testbed that is free of fixed or permanently installed equipment. The testbed is surrounded by several observation systems installed on the ceiling, at the walls, and within the floor. It also includes a laser projection system, which is the most important component for this paper. The experimentation space is 22 m long, 15 m wide, and up to 3.5-4 m high. Eight laser projectors cover the full area and can project coloured vector graphics on the ground floor. The selection process favoured projectors with a high frame rate rather than accuracy.

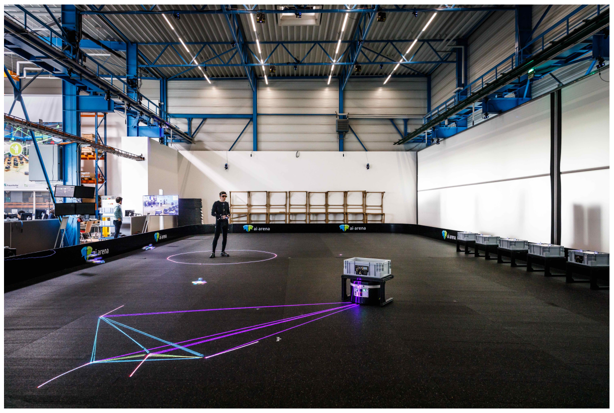


Figure 2: Logistics research lab overview

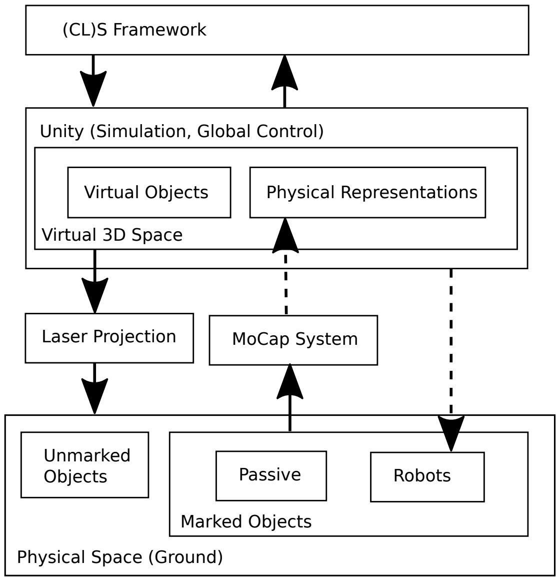


Figure 3: Logistics research lab architecture

Fig. 2 shows a typical use of the system. A person, robot and box are tracked by the MoCap (Motion Capturing) system. The MoCap system consists of 40 infrared cameras by Vicon [15]. The data stream is accessible over network to multiple clients and delivers position and rotation of tracked objects in three dimensions. In addition to simple physical objects, several marker-suits are available for tracking persons which are then used to generate data streams of complex skeletal models including individual body parts. In the foreground, the laser system projects a visual representation of the current state of the steering algorithm of the robot. The projected circle around the person in the background represents the safety area which the robot is not allowed to enter. The box on the robot contains an embedded system (black square on the front) that communicates with the robot for transportation needs.

4 Experiment and Extensions

The debugging facilities discussed in Sec. 2 help with understanding types and to explore domain-independent technical aspects of the synthesis algorithm. However, a layer of interpretation is required to map them to any given concrete scenario. Here diversity of training and mindsets really helped us to go forward. While computer scientists working on formal methods are usually trained to aim for solutions with maximal feasible mathematical generality, engineers are focused on smallest viable solutions. In

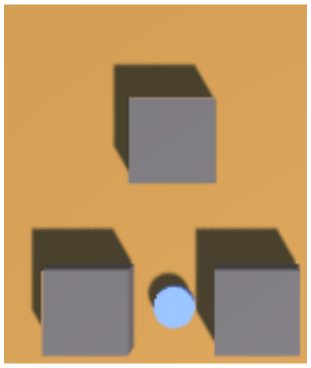


Figure 4: Unity 3D Representation

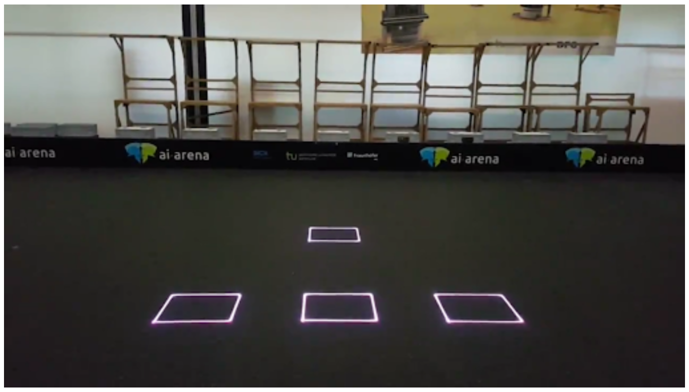


Figure 5: Laser Projection System Representation

the spirit of good engineering practices, we ventured to build a minimal prototype to demonstrate the following properties:
1. (CL)S is applied to a logistical setting within the demonstration capabilities of the logistics lab.
2. Scenario-specific meaningful debugging is possible without consequences of damaged hardware.
3. Users interactively control the debugging process.
4. Hardware failures are separated from logical specification problems.
5. Existing technologies are used unchanged if possible, preferring adapters when necessary.
An explicit non-goal was to expect a realistic solution compliant with industry standards.

The labyrinth example, incidentally, matched the first goal, which is why we decided to use it as a starting point. Fig. 3 shows the architecture connecting it to the logistics lab. It contains four layers. The bottom layer represents the real world. It contains real entities (marked objects), which are robots and obstacles. It is also augmented by laser projections of virtual objects (unmarked objects), e.g. start points and goals. The second layer represents the fixed lab installation discussed above. All marked objects in physical space are mirrored via live MoCap connection into a virtual environment that makes up the third layer. It is realized using the Unity 3D [1] game engine and capable of virtually representing physical states, simulating them when no connection to the lab is available, and controlling them by sending commands to the second layer. Its virtual abstractions of physical state are mapped to component repositories for (CL)S, which is shown in the topmost layer. Solutions synthesized by (CL)S are movement commands for robots and sent back to Unity, which either just simulates them in a 3D model or forwards them to the laboratory equipment. In line with goals 2 and 4, we found it useful to virtualize robots, turning them into laser projections rather than immediately trying to control real hardware. A 3D simulation in Unity and its laser projection are shown in figures 4 and 5. They show the end state of the labyrinth example (s. Section 2) where the robot has already reached the goal position (3, 1). A video of the laser projected movements generated by (CL)S is available online [3]. Future development will fully exploit the possibility to update labyrinths based on real world obstacles detected by motion capturing, which would be in line with goal 3.

Connecting the different layers was a major challenge, because all of them are preexisting components implemented in different programming languages (Python and C++ for physical components, C# for Unity and Scala for (CL)S). The bottom layers were already connected by the ISO-standardized MQTT network protocol [11], which made it natural to also use it for the communication with (CL)S. Here, choosing a standard language to implement our tool was crucial, because we were able to use Eclipse Paho [9], an off-the-shelve implementation of MQTT, instead of going through the tedious error prone process of developing our own networking infrastructure. Currently, all connections in Fig.3

are implemented using MQTT. The dashed arrows have been implemented, but at the time of writing still need to be logically integrated into the mapping from Unity to (CL)S, which means labyrinths are currently specified in source code and robots are visualized with lasers.

5 Lessons Learned and Future Plans

The experiment we presented is rather small and just a starting point, but already provided us with some technical and non-technical insights, which can be useful for other researchers willing to engage in interdisciplinary collaboration. Technically, one of the most important aspects was to have a formal system with clear boundaries. The (CL)S Framework is designed to collaborate with others by not insisting to share any of its platform specific requirements. Instead, the experiment used a complete (even physical) separation with a light-weight MQTT network interface. This allowed almost zero integration overhead and meant no software systems had to be rewritten. The framework and system to control are both implemented in mainstream languages with preexisting library support for network communication. We conjecture that some other platforms, e.g. with external domain specific languages, would have caused much higher development costs. Being compatible with mainstream technology was key. There was just no way we could have foreseen which networking libraries would be useful for our project. Insisting on our own tech-stack would have caused the project to end before it started. Also, on a technical level it quickly became clear, that domain-independent generic visualization methods were not sufficient. While domain experts can appreciate that there is a formal model, for them it is just a vehicle to solve a real world problem. Practically, this means specific tools are always preferred over abstract data. For debugging and testing, users should be presented with a (graphical) language close to what they know. Laser-based visualization and 3D-Modeling, which bridge the gap between virtual and physical systems, are perfect tools to provide a good, tangible user experience. On a social level, small scale prototypes and specific solutions, that do not yet scale to a large system, are much more acceptable to engineers than generic solutions, if the smaller scale implies less effort to get real things moving. This is very different from expert communities with more focus on mathematical theory, where generic solutions are expected and not filling in the details to get to a specific executable system is to be forgiven more easily, or even expected, because details are considered to be time-consuming and repetitive. Another social lesson is to consider the training of the target audience. Technicalities, such as obtaining the framework code from Git or executing it, imposed negligible effort, which is perhaps surprising for theory-minded people. For our small scale experiment, conveying the purpose of synthesis, the meaning of intersection types and establishing a shared vocabulary took more time than the actual development. While for us personal communication was the easiest way, this may not always be easily possible (e.g. if teams are situated in separate locations). Future developments should take explaining the language of the tools into account. We feel, that contact to other researchers is crucial to do this, because identifying conventions of language once they are established, is difficult.

In this spirit, extend our invitation to other researchers to collaborate on new experiments, possibly integrating more formal tools into the logistics lab. Our own upcoming next experiments will scale up the experiment to more realistic scenarios, where (CL)S synthesizes code computing paths instead of directly computing the paths. Integrating some of the much more sophisticated existing path finding algorithms will allow the transition to a more fine granular view of the world than labyrinths with blocked paths. Specifically it will enable the treatment of obstacles with non-rectangular shapes, non straight paths between points, speed and energy considerations during move and perhaps even reactive systems, where the position of obstacles can change (e.g. by having more robots). In future CPS-based logistics systems, large numbers of autonomous and networked entities will arrange themselves ad-hoc in temporary constellations to provide logistics services in coordination with humans. The development of these heterogeneous systems presents a challenge to engineers with their complexity of interacting hard- and software in industrial environments. It is our opinion that user-friendly formally understood development tools will be crucial to have any chance of facing this challenge.

References

[1] *Unity*. Available at `https://docs.unity3d.com/Manual/UnityManual.html`.

[2] Anna Vasileva & Jan Bessai (06.06.2019): *cls-scala-ide*. Available at `https://github.com/combinators/cls-scala-ide`.

[3] Anna Vasileva & Moritz Roidl (06.06.2019): *Laser Demonstration*. Available at `https://github.com/combinators/labyrinth`.

[4] Haci Bayhan, Aswin Karthik Ramachandran Venkatapathy, Johannes Dregger, Felix Zeidler, Moritz Roidl & Michael ten Hompel (2017): *A Concept of an Industry 4.0 Research Lab for Future Intralogistics Technologies and Services*. 3rd Interdisciplinary Conference on Production, Logistics and Traffic, ICPLT.

[5] Jan Bessai, Andrej Dudenhefner, Boris Düdder, Moritz Martens & Jakob Rehof (2016): *Combinatory Process Synthesis*. In: Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I, pp. 266–281, doi:10.1007/978-3-319-47166-2_19.

[6] Jan Bessai & Anna Vasileva (2018): *User Support for the Combinator Logic Synthesizer Framework*. Electronic Proceedings in Theoretical Computer Science 284, pp. 16–25, doi:10.4204/EPTCS.284.2.

[7] Edsger W Dijkstra (1959): *A note on two problems in connexion with graphs*. Numerische mathematik 1(1), pp. 269–271.

[8] Engelfriet, Joost & Heyker, Linda (1992): *Context-free hypergraph grammars have the same term-generating power as attribute grammars*. Acta Informatica.

[9] Eclipse Foundation (2019): *Paho*. Available at `https://www.eclipse.org/paho/`.

[10] George T. Heineman, Jan Bessai, Boris Düdder & Jakob Rehof (2016): *A Long and Winding Road Towards Modular Synthesis*. In: ISoLA 2016, pp. 303–317, doi:10.1007/978-3-319-47166-2_21.

[11] International Organization for Standardization (ISO) (2016): *ISO/IEC 20922:2016: Information technology – Message Queuing Telemetry Transport (MQTT) v3.1.1*. ISO: Geneva, Switzerland, pp. 1–73.

[12] Fadil Kallat, Tristan Schaefer & Anna Vasileva (2019): *CLS-SMT: Bringing Together Combinatory Logic Synthesis and Satisfiability Modulo Theories*. In Proceedings of the Proof eXchange for Theorem Proving 2019.

[13] Rudolph Emil Kalman (1960): *A new approach to linear filtering and prediction problems*. Journal of basic Engineering 82(1), pp. 35–45.

[14] Jakob Rehof (2013): *Towards Combinatory Logic Synthesis*. In: BEAT 2013, 1st International Workshop on Behavioural Types, ACM.

[15] Vicon (05.09.2019): *Motion Tracking Devices*. Available at `https://www.vicon.com/`.

[16] Jan Winkels, Julian Graefenstein, Tristan Schäfer, David Scholz, Jakob Rehof & Michael Henke (2018): *Automatic composition of rough solution possibilities in the target planning of factory planning projects by means of combinatory logic*. In: ISoLA 2018, Springer, pp. 487–503.

Automated deductive verification for Ladder programming

Denis Cousineau David Mentré
Mitsubishi Electric R&D Centre Europe (MERCE)
Rennes, France
{d.cousineau,d.mentre}@fr.merce.mee.com

Hiroaki Inoue
Mitsubishi Electric Corporation
Amagasaki, Japan
Inoue.Hiroaki@ah.MitsubishiElectric.co.jp

Ladder Logic is a programming language standardized in IEC 61131-3 and widely used for programming industrial Programmable Logic Controllers (PLC). A PLC program consists of inputs (whose values are given at runtime by factory sensors), outputs (whose values are given at runtime to factory actuators), and the logical expressions computing output values from input values. Due to the graphical form of Ladder programs, and the amount of inputs and outputs in typical industrial programs, debugging such programs is time-consuming and error-prone. We present, in this paper, a Why3-based tool research prototype we have implemented for automating the use of deductive verification in order to provide an easy-to-use and robust debugging tool for Ladder programmers.

1 Introduction

Programmable logic controllers (PLC) are industrial digital computers used as automation controllers of manufacturing processes, such as assembly lines or robotic devices. PLCs can simulate the hard-wired relays, timers and sequencers they have replaced, via software that expresses the computation of outputs from the values of inputs and internal memory. Ladder language, also known as Ladder Logic, is a programming language used to develop PLC software. This language uses circuits diagrams of relay logic hardware to represent a PLC program by a graphical diagram. This language was the first available to program PLCs. It is now standardized in IEC 61131-3 [1] standard among other languages but is still widely used and very popular among technicians and electrical engineers.

In conventional development of software, a great part of the development time is dedicated to debugging. Debugging programs is crucial in the case of Factory Automation (FA) since bugs in factories can be extremely expensive in terms of human and material damages, and plant downtime. Debugging a Ladder program is particularly difficult, time consuming and costly. Bugs can be depicted as the violation, at some point of a program, of some property concerning values of inputs/outputs and local memory of the program. The objective of debugging consists in detecting those property violations before running the code in production, i.e. finding initial values of inputs and internal memory that lead to a property violation, when executing the program. Since it is almost impossible (and way too costly) to check all possible executions of a program, the usual method consists in developing and running some tests (i.e. executing the program on a particular initial configuration and check its behavior). In industry, tests used to be run directly in the factory, which is very costly and risky. Nowadays, most of the tests are run on a software simulation, but some are often still run in the factory for a last check of the program behavior in real conditions of use, or for bypassing the difficulty to simulate particular sequences of inputs. Even when run on a software simulation, tests-based processes are still time-consuming and cannot be exhaustive.

On the other hand, some research work has been done concerning formal analysis of Ladder programs. Most of this work [7] [6] [9] [10] concerns the verification of temporal properties of Ladder

© D. Cousineau, D. Mentré & H. Inoue

programs (a Ladder program being continuously executed in the PLC), and uses different model checking techniques. Some other work used deductive verification to detect data races [13] and prove safety properties [12] (with some temporal aspects) of Ladder programs. Model-checking techniques are limited by the state explosion problem they face when addressing real-world problems. On the contrary, deductive verification may give full confidence in the obtained results but may also prevent from a full automatization of the process (in terms of proof automation and specification formalization).

Our objective in this work was to make a proof of concept of an easy-to-use and robust tool for debugging Ladder programs, both increasing the quality of the code and decreasing the time to deploy, a crucial point in the context of Industry 4.0, in which assembly lines are more often reconfigured, hence code evolves frequently. We had to determine a good tradeoff to offer a high level of automation, together with providing a strong confidence in the given results (in particular when no bug is found). The solution we chose is similar to what SPARK/GnatPro [8] has done for Ada code. For the easy-to-use part, we targeted a full automated tool, so that it could be used by regular engineers with no needed knowledge in formal methods. We also focused on the information given to the programmer when an error is found, for easing errors fixing. In order to obtain a fully automated and complete tool, we targeted to detect runtime errors by the mean of deductive verification. We focused on runtime errors like integers overflows, divisions by zero, violations of Ladder instructions' pre-conditions, etc...

We based our prototype implementation on the Why3 platform [4]. Why3 offers an expressive formalization language, an efficient Weakest-Precondition (WP) calculus [5] implementation and a rich API to send the obtained verification conditions to several automated solvers. Moreover, with its *labels* mechanism, Why3 allowed us to keep code information during the whole automatic process, for providing rich and useful information to the programmer in case a bug is detected.

2 Ladder Logic

Ladder Logic is a graphical programming language using relay logics diagrams to represent a PLC program. A Ladder program takes inputs values (*contacts*) that correspond to the fact that physical relays are wired, not wired, pulsing (rising edge) or downing (falling edge) and other values stored in the internal memory of the PLC (booleans, integers, floating point, strings, etc...). A Ladder program can output boolean values to the physical relays of the factory (*coils*) or it can call instructions, that may modify the values of the internal memory of the PLC (*devices*). Graphically, contacts are located at the left of the diagram. They can be combined in a serial way or in a parallel way (the obtained value is then the conjunction, resp. the disjunction of the two contacts values). Coils and instructions are activated when the combination of contacts at their left gives a *wired* value, and they can also be parallelized (in that case, there are either all activated or all deactivated). A line with contacts, coils and instructions is called a *rung*, and a program (a *diagram*) is composed of several rungs.

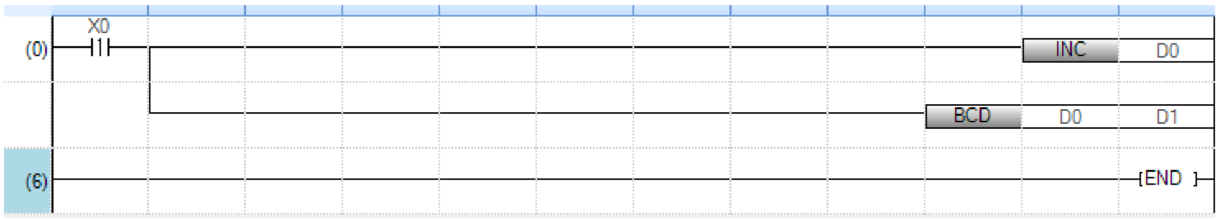
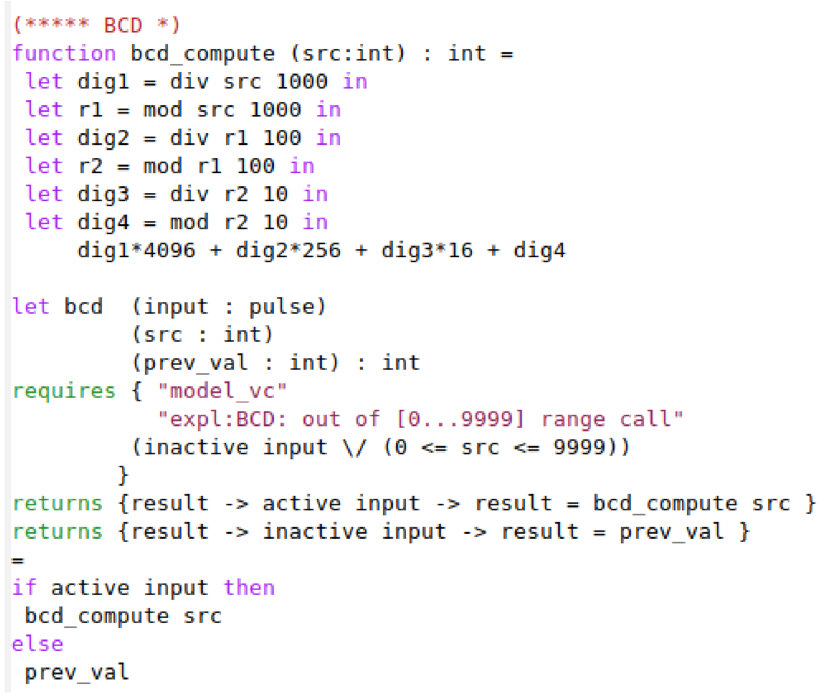


Figure 1: Ladder program example

Figure 1 depicts a very simple Ladder example. This program has one contact `X0`, and when the physical relay corresponding to that input is activated, the program calls instruction `INC` that increments the value of its device argument `D0`, and then calls instruction `BCD` with `D0` and `D1` devices as respectively input and output arguments. The `BCD` instruction converts a 16 bits integer into a 16 bits BCD (Binary-Coded Decimal) integer. The 16 bits BCD format represents 4 digits decimal numbers, using 4 bits to represent each of the 4 digits. It is typically used for display purpose. Since this format can only represent 4 digits decimal numbers, the *BCD* instruction raises an error when it is called on a device value that does not belong to interval $[0;9999]$. This is typically the kind of runtime errors we want to detect with the tool we developed. Regarding this example, we were also interested in overflows that could occur when calling instruction `INC`. The example we present is very simple but typical industrial programs we had access to have hundreds of lines, hundreds of inputs, devices and outputs, and dozens of instructions calls. As a last point, such a Ladder program is executed cyclically in a synchronous way: first inputs are read, then the program is executed and eventually outputs are written. One single execution of the program is called a *scan*.

3 Modelling Ladder in Why3

We chose not to model the temporal/cyclic aspect of execution of Ladder programs, but only one scan in order to detect error scenarios, i.e. values of inputs and devices at scan beginning (before execution) that may lead to a runtime error. We developed a library of Ladder instructions formalizations. We depict here the formalization of the `BCD` instruction. This formalization is composed of two functions. The first one, `bcd_compute`, computes the 4 digits of an decimal integer argument and returns the decimal value of the BCD representation of those 4 digits. The second one, `bcd` takes three arguments: `input` is the wiring value of the line to which the instruction is connected, `src` is the value of the input device, and `prev_val` is the value of the output device before execution of the instruction. The `requires` pre-condition states that either `input` does not activate the instruction or `src` must belong to interval $[0;9999]$. You can notice the two strings *labels* in the pre-condition. The first allows asking solvers to find a counter-example if they cannot prove the verification conditions associated with that pre-condition. The second one allows keeping semantic information during the whole process, in order to give back this information to the programmer in case an error scenario is found. The `returns` post-condition states that the function returns the previous value of the output device when the instruction is not activated, and the actual BCD computation otherwise.



```
(***** BCD *)
function bcd_compute (src:int) : int =
 let dig1 = div src 1000 in
 let r1 = mod src 1000 in
 let dig2 = div r1 100 in
 let r2 = mod r1 100 in
 let dig3 = div r2 10 in
 let dig4 = mod r2 10 in
     dig1*4096 + dig2*256 + dig3*16 + dig4

let bcd  (input : pulse)
         (src : int)
         (prev_val : int) : int
requires { "model_vc"
           "expl:BCD: out of [0...9999] range call"
         (inactive input \/ (0 <= src <= 9999))
        }
returns {result -> active input -> result = bcd_compute src }
returns {result -> inactive input -> result = prev_val }
=
if active input then
 bcd_compute src
else
 prev_val
```

Figure 2: Why3 formalization of Ladder BCD instruction

We developed around fifty such formalizations of Ladder instructions in order to run our tool prototype on the industrial program samples we had access to. Then our translation of Ladder programs to Why3 models consists in translating on-the-fly the logical expressions that correspond to coils and their combinations, and combine them with calls to the instructions formalizations of our library, using a single-state-assignment [11] transformation to handle the iterative aspect of Ladder programs.

4 Prototype architecture

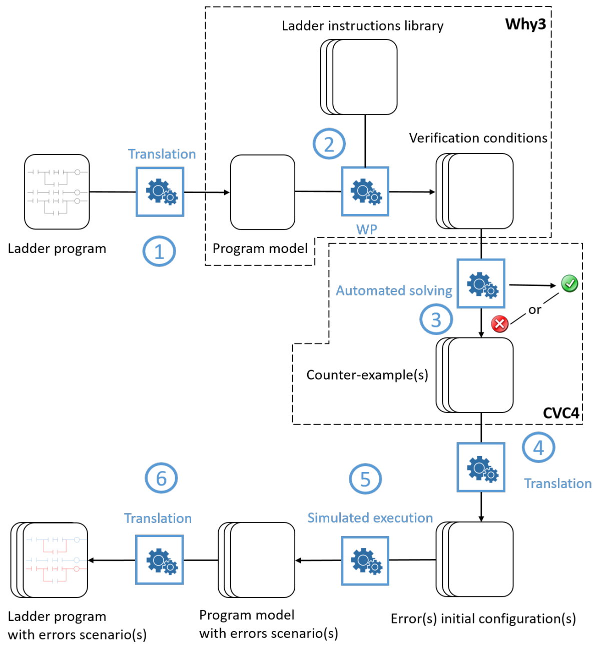
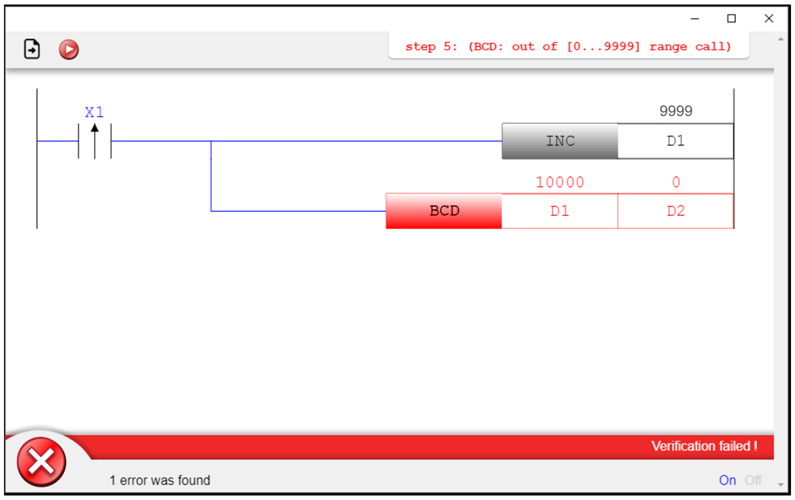


Figure 3: Prototype architecture

Our tool automatically ① translates Ladder programs into Why3 modules that refer to the instructions formalizations described in the previous section. We implemented our own library to produce Why3 text files, to help the reuse of generated modules. During the translation, we use *labels* to keep information on code location of instruction calls to give improved feedback to the programmer when an error is found.
Then we use Why3's WP calculus to ② compute verification conditions that correspond to preconditions of instructions, and use Why3 API to ③ send those verification conditions to SMT-solver CVC4 [3] (we chose CVC4 for its overall good performances and its ability to generate counter-examples when a verification condition cannot be proved). Couter-examples are then ④ interpreted as initial values of the original program and simulated execution ⑤ recomputes, from those inital values, all the intermediate values of devices, wires, etc... from the beginning of the program to the location where the error occurs. Finally, we ⑥ provide a graphical feedback to the programmer, with those intermediate values information and informations concerning the error the tool found.

5 Graphical user feedback

We implemented a proof of concept of a graphical interface which gives back to the programmer information about found bugs, in an easy-to-understand manner. The aim of such an interface is to be directly integrated in Ladder IDEs. We based our prototype implementation on the Ocsigen web framework [2] which allowed us to quickly prototype a web-based graphical interface displaying information coming from our tool prototype implemented in OCaml. We identified three pieces of information that should be displayed to the programmer when a bug is found: the *error location* (where the error occurs), the *error reason* (why the error occurs) and the *error scenario* (when the error occurs). The error location is encoded during the on-the-fly translation from the Ladder program to the Why3 model: to each instruction call is attached a label which contains its location in the original source code. This label is propagated during the WP calculus, appears in the verification condition sent to the automated solver and comes back in the counter-example the solver gives when it finds one. The error reason is encoded in the Why3 instructions library as shown in figure 2. It is attached, with a `"expl:"` label, to pre-conditions of Ladder instructions, and is propagated during the whole process, like code locations labels. As explained in the previous section, the error scenario consists in the initial and intermediate values that lead to the error. It is re-computed from solvers' counter-examples, and is expressed with colors for wiring values (blue when a wire is active, grey otherwise) and figures for other values above the corresponding devices. Figure 4 shows a screenshot of this graphical interface. This is what our prototype returns when run on example of figure 1. In this case, the interface states the errors occurs at BCD instruction call location (it is colored in red). It also states that the error reason is an out-of-range call. And it gives the error scenario: contact `X1` is active hence colored in blue, then the wires at its right are also active and colored in blue. After execution of instruction `INC`, value stored in device `D1` is $10,000$, which leads to the error, when given as argument of instruction `BCD`.


step 5: (BCD: out of [0...9999] range call)

Verification failed !

1 error was found

Figure 4: Graphical feedback

We believe that this interface, somehow inspired by what already exists in Ladder simulation software, in particular for the colors, may be very useful for debugging Ladder programs, in particular when they reach a critical size with hundreds of rungs, inputs, devices, outputs, etc... Indeed, it is much easier to understand why an error occurs with this kind of interface, than when using tests, in which case only the initial configuration of the program is given.

6 Performances

Our proprietary prototype is implemented in OCaml, in about $13,000$ lines of code, including $3,000$ lines for our library to produce Why3 textual files and $4,000$ lines for the graphical user interface. We made some optimization effort concerning the modelization of Ladder language in Why3, in order to obtain a fully automated and fast process. But we made no optimization in our OCaml code, and even did not parallelize the calls to SMT-solvers. Nonetheless our protype has already pretty good performances. We ran our prototype tool on an industrial code sample with $1,657$ *steps* (i.e. contacts, coils and instruction calls), among which three instructions calls could lead to an error. On a virtualized Ubuntu 18.04, running in VirtualBox 5.2, under Windows 10 on a Intel Core i7-7500U 2.70 GHz laptop, it takes only 3 to 4 seconds for our prototype to answer. Almost all the time is taken by CVC4 (about one second for each of the three verification conditions that are handled sequentially). This gives us confidence in the fact that the technology and architecture we chose are relevant for the implementation of a real industrial tool.

7 Conclusion

The objective of this work was to make a proof of concept of a formal methods-based debugging tool for industrial Ladder programs. For such a debugging tool to be incorporated in an industrial process, we

think that it should be transparent and bring strong added value to the user. First, our proof of concept shows that such a formal debugging tool can be transparent: it needs no specific knowledge since all the process is fully automatic (Ladder programmers do not need to write a formal specification, and even less a model of their codes); it is very fast so it may be run during the programming phase of the development process and not in a separated phase; it may be fully integrated in a Ladder IDE as our GUI prototype shows. Second, our proof of concept shows the added value such a tool could have in regard to current debugging tools: our prototype can give back to the programmer very precise and useful information when it detects an error (using a intelligible interface); and the deductive verification technique we used, thanks to the Why3 platform, gives a strong confidence when the tool detects no runtime error (since it is equivalent to test all possible inputs and devices values configurations).
A drawback of our prototype concerns the fact that it may raise false positive alarms, since it only considers one scan of the Ladder program. For example in Figure 1, value of device `D0` may be changed *after* the `BCD` instruction call, such that value $10,000$ is never reached. Nevertheless, our prototype would still raise an alarm. In future work, we plan to decrease the number of false alarms by considering a few consecutive scans in our Why3 Ladder formalization.
Another way to improve our prototype could be to provide some *quickfix*-like mechanisms to programmers. In example of Figure 1, our prototype could propose to the programmer to add automatically, before the `BCD` instruction call, a line that resets `D0` when it does not belong to range $[0;9999]$.

References

[1] (2013): *IEC 61131-3:2013, Programmable controllers - Part 3: Programming languages.*

[2] V. Balat (2006): *Ocsigen: Typing Web Interaction with Objective Caml.* In: *ACM SIGPLAN workshop on ML*, Portland, United States.

[3] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds & C. Tinelli (2011): *CVC4.* In: *nternational Conference on Computer Aided Verification*, CAV'11, Springer-Verlag, pp. 171–177.

[4] F. Bobot, J-C. Filliâtre, C. Marché & A. Paskevich (2015): *Let's Verify This with Why3. Software Tools for Technology Transfer (STTT)* 17(6), pp. 709–727.

[5] E. W. Dijkstra (1997): *A Discipline of Programming.* Prentice Hall PTR, Upper Saddle River, NJ, USA.

[6] B. Fernández Adiego, D. Darvas, E. B. Viñuela, J. Tournier, S. Bliudze, J. O. Blech & V. M. González Suárez (2015): *Applying Model Checking to Industrial-Sized PLC Programs. IEEE Transactions on Industrial Informatics* 11(6), pp. 1400–1410.

[7] G. Frey & L. Litz (2000): *Formal methods in PLC programming.* In: *IEE International conference on systems, man and cybernetics*, 4, pp. 2431–2436 vol.4.

[8] D. Hauzar, C. Marché & Y. Moy (2016): *Counterexamples from Proof Failures in SPARK.* In: *Software Engineering and Formal Methods*, Software Engineering and Formal Methods, Springer.

[9] S. Kottler, M. Khayamy, S. R. Hasan & O. Elkeelany (2017): *Formal verification of ladder logic programs using NuSMV.* In: *SoutheastCon 2017*, pp. 1–5.

[10] T. Ovatman, A. Aral, D. Polat & A. Osman Ünver (2014): *An overview of model checking practices on verification of PLC software. Software and Systems Modeling*, pp. 1–24.

[11] B. K. Rosen, M. N. Wegman & F. K. Zadeck (1988): *Global Value Numbers and Redundant Computations.* In: *Symposium on Principles of Programming Languages*, POPL '88, ACM, pp. 12–27.

[12] J-M. Roussel & B. Denis (2002): *Safety properties verification of ladder diagram programs. Journal Européen des Systèmes Automatisés (JESA)* 36(7), pp. pp. 905–917.

[13] Z. Su (1997): *Automatic Analysis of Relay Ladder Logic Programs.* Technical Report UCB/CSD-97-969, EECS Department, University of California, Berkeley.

Deeply Integrating C11 Code Support into Isabelle/PIDE

Frédéric Tuong
LRI, Univ. Paris-Sud, CNRS, Université Paris-Saclay
ftuong@lri.fr

Burkhart Wolff
LRI, Univ. Paris-Sud, CNRS, Université Paris-Saclay
wolff@lri.fr

We present a framework for C code in C11 syntax deeply integrated into the Isabelle/PIDE development environment. Our framework provides an abstract interface for verification back-ends to be plugged-in independently. Thus, various techniques such as deductive program verification or white-box testing can be applied to the same source, which is part of an integrated PIDE document model. Semantic back-ends are free to choose the supported C fragment and its semantics. In particular, they can differ on the chosen memory model or the specification mechanism for framing conditions.

Our framework supports semantic annotations of C sources in the form of comments. Annotations serve to locally control back-end settings, and can express the term focus to which an annotation refers. Both the logical and the syntactic context are available when semantic annotations are evaluated. As a consequence, a formula in an annotation can refer both to HOL or C variables.

Our approach demonstrates the degree of maturity and expressive power the Isabelle/PIDE subsystem has achieved in recent years. Our integration technique employs Lex and Yacc style grammars to ensure efficient deterministic parsing. We present two case studies for the integration of (known) semantic back-ends in order to validate the design decisions for our back-end interface.

Keywords: User Interface, Integrated Development, Program Verification, Shallow Embedding

1 Introduction

Recent successes like the Microsoft Hypervisor project [14], the verified CompCert compiler [15] and the seL4 microkernel [11, 12] show that the verification of low-level systems code has become feasible. However, a closer look at the underlying verification engines VCC [6], or Isabelle/AutoCorres [8] show that the road is still bumpy: the empirical cost evaluation of the L4.verified project [11] reveals that a very substantial part of the overall effort of about one third of the 28 man years went into the development of libraries and the associated tool-chain. Accordingly, the project authors [11] express the hope that these overall investments will not have to be repeated for “similar projects”.

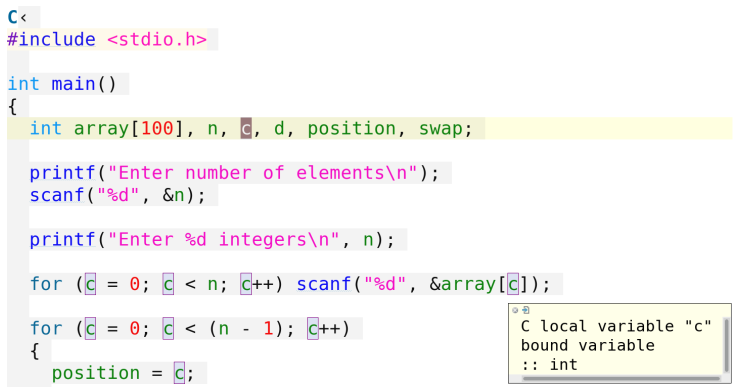
In fact, none of these verifying compiler tool-chains capture all aspects of “real life” programming languages such as C. The variety of supported language fragments seem to contradict the assumption that we will all converge to one comprehensive tool-chain soon. There are so many different choices concerning memory models, non-standard control flow, and execution models that a generic framework is desirable: in which verified compilers, deductive verification, static analysis and test techniques (such as [10], [1]) can be developed and used inside the Isabelle platform as part of an integrated document.

In this paper we present Isabelle/C [1], a generic framework in spirit similar to Frama-C [16]. In contrast to the latter, Isabelle/C is deeply integrated into the Isabelle/PIDE document model [20]. Based on the C11 standard (ISO/IEC 9899:2011), Isabelle/C parses C11 code inside a rich IDE supporting

[1]The current developer snapshot is provided in `https://gitlri.lri.fr/ftuong/isabelle_c`.

Submitted to:
F-IDE 2019

© F. Tuong and B. Wolff
This work is licensed under the
Creative Commons Attribution License.



```
C‹
#include <stdio.h>

int main()
{
  int array[100], n, c, d, position, swap;

  printf("Enter number of elements\n");
  scanf("%d", &n);

  printf("Enter %d integers\n", n);

  for (c = 0; c < n; c++) scanf("%d", &array[c]);

  for (c = 0; c < (n - 1); c++)
  {
    position = c;
```

```
C local variable "c"
bound variable
:: int
```

Figure 1: A C11 Sample in Isabelle/jEdit

static scoping. SML user-programmed extensions can benefit from the parallel evaluation techniques of Isabelle. The plug-in mechanism of Isabelle/C can integrate diverse semantic representations, including those already made available in Isabelle/HOL [17]: AutoCorres [8], IMP2 [13], ORCA [3], or Clean (discussed in this paper). A particular advantage of the overall approach compared to systems like Frama-C or VCC is that all these semantic theories are conservative extensions of HOL, hence no axiom-generators are used that produce the "background theory" and the verification conditions passed to automated provers. Isabelle/C provides a general infrastructure for semantic annotations specific for back-ends, i.e. modules that generate from the C source a set of definitions and derive automatically theorems over them. Last but not least, navigation features of annotations make the logical context explicit in which theorems and proofs are interpreted.

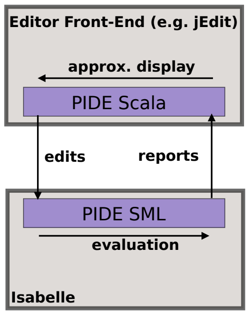
The heart of Isabelle/C, the new **C**‹ .. › command, is shown in Figure 1. Analogously to the existing **ML**‹ .. › command, it allows for editing C sources inside the ‹ .. › brackets, where C code is parsed on the fly in a “continuous check, continuous build” manner. A parsed source is coloured according to the usual conventions applying for Isabelle/HOL variables and keywords. A static scoping analysis makes the bindings inside the source explicit such that editing gestures like hovering and clicking may allow the user to reveal the defining variable occurrences and C type information (see yellow sub-box in the screenshot Figure 1). The C source may contain comments to set up semantic back-ends. Isabelle/C turns out to be sufficiently efficient for C sources such as the seL4 project.

This paper proceeds as follows: in section 2, we briefly introduce Isabelle/PIDE and its document model, into which our framework is integrated. In section 3 and section 4, we discuss the build process and present some experimental results on the integrated parser. The handling of semantic annotations comments — a vital part for back-end developers — is discussed in section 5, while in section 6 we present some techniques to integrate back-ends into our framework at the hand of examples.

2 Background: PIDE and the Isabelle Document Model

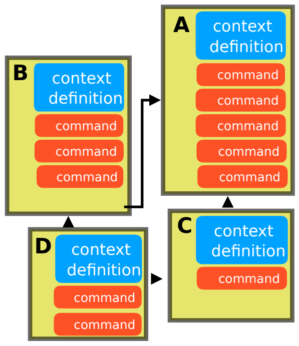
The Isabelle system is based on a generic document model allowing for efficient, highly-parallelized evaluation and checking of its document content (cf. [2, 20, 21] for the fairly innovative technologies underlying the Isabelle architecture). These technologies allow for scaling up to fairly large documents: we have seen documents with 150 files be loaded in about 4 min, and individual files — like the x86 model generated from Antony Fox' L3 specs — have 80 kLoC and were loaded in about the same time.[2]

The PIDE (prover IDE) layer consists of a part written in SML and another in Scala. Roughly speaking, PIDE implements "continuous build and continuous check" functionality over a textual albeit generic document model. It transforms user modifications of text elements in an instance of this model into increments — *edits* — and communicates them to the Isabelle system. The latter reacts by the creation of a multitude of light-weight reevaluation threads resulting in an asynchronous stream of *reports* containing *markup* that is used to annotate text elements in the editor front-end. For example, such markup is used to highlight variables or keywords with specific colours, to hyperlink bound variables to their defining occurrences, or to annotate type information to terms which become displayed by specific user gestures on demand (such as hovering). Note that PIDE is not an editor, it is the framework that coordinates these asynchronous information streams and optimizes their evaluation to a certain extent: outdated markup referring to modified text is dropped, and corresponding re-calculations are oriented to the user focus, for example. For PIDE, several editor applications have been developed, where Isabelle/jEdit (`https://www.jedit.org`) is the most commonly known. More experimental alternatives based on Eclipse or Visual Studio Code exist.



2.1 The PIDE Document Model

The document model foresees a number of atomic sub-documents (files), which are organized in the form of an acyclic graph. Such graphs can be grouped into sub-graphs called *sessions* which can be compiled to binaries in order to avoid long compilation times — Isabelle/C as such is a session. Sub-documents have a unique name (the mapping to file paths in an underlying file-system is done in an integrated build management). The primary format of atomic sub-documents is `.thy` (historically for "theory"), secondary formats can be `.sty`, `.tex`, `.c` or other sub-documents processed by Isabelle and listed in a configuration of the build system.



A `.thy` file consists of a *context definition* and a body consisting of a sequence of *commands*. The context definition includes the sections **imports** and **keywords**. For example our context definition states that *C_Command* is the name of the sub-document depending on *C_Eval* which transitively includes the parser sources as (ML files) sub-documents, as well as the C environment and the infrastructure for defining C level annotations. *Keywords* like **C** or **C_file** must be declared before use.

```
theory C_Command
  imports C_Eval
  keywords "C" :: thy_decl
       and "C_file" :: thy_load
```

For this work, it is vital that predefined commands allow for the dynamic creation of *user-defined* commands similarly to the definition of new functions in a shell interpreter. Semantically, commands are transition functions $\sigma \to \sigma$ where σ represents the system state called *logical context*. The logical context in interactive provers contains — among many other things — the declarations of types, constant symbols as well as the database with the definitions and established theorems. A command starts with a pre-declared keyword followed by the specific syntax of this command; an *evaluation* of a command parses the input till the next command, and transfers the parsed input to a transition function, which can be configured in a late binding table. Thus, the evaluation of the generic document model allows for user programmed extensions including IDE and document generation.

[2]On a modern 6-core MacBook Pro with 32Gb memory, these loading times were counted *excluding* proof checking.

Note that the Isabelle platform supports multiple syntax embeddings, i.e. the possibility of nesting different language syntaxes inside the upper command syntax, using the `⟨ .. ⟩` brackets (such parsing techniques will be exploited in section 5). Accordingly, these syntactic sub-contexts may be nested. In particular, in most of these sub-contexts, there may be a kind of semantic macro — called antiquotation and syntactically denoted in the format `@{name ⟨ .. ⟩}` — that has access to the underlying logical context. Similar to commands, user-defined antiquotations may be registered in a late-binding table. For example, the standard *term*-antiquotation in **ML**`⟨ val t = @{term "3 +"} ⟩` parses the argument `"3 +"` with the Isabelle/HOL term parser, attempts to construct a λ-term in the internal term representation and to bind it to `t`; however, this fails (the plus operation is declared infix in logical context) and therefore the entire command fails.

2.2 Some Basics of PIDE Programming

A basic data-structure relevant for PIDE is *positions*; beyond the usual line and column information they can represent ranges, list of continuous ranges, and the name of the atomic sub-document in which they are contained. It is straightforward to use the antiquotation `@{here}` to infer from the system lexer the actual position of the antiquotation in the global document. The system converts the position to a markup representation (a string representation) and sends the result via `writeln` to the interface.

```
ML⟨ val pos = @{here};
    val markup = Position.here pos;
    writeln ("And a link to the declaration\
             \ of 'here' is " ^ markup) ⟩
```

In return, the PIDE output window shows the little house-like symbol, which is actually hyperlinked to the position of `@{here}`. The ML structures `Markup` and `Properties` represent the basic libraries for annotation data which is part of the protocol sent from Isabelle to the front-end. They are qualified as "quasi-abstract", which means they are intended to be an abstraction of the serialized, textual presentation of the protocol. A markup must be tagged with a unique id; this is done by the library `serial` function. Typical code for taking a string `cid` from the editing window, together with its position `pos`, and sending a specific markup referring to this in the editing window managed by PIDE looks like this:

```
And a link to the declaration of 'here' is ⌂
```

```
ML⟨ fun report_def_occur pos cid = Position.report pos (my_markup true cid (serial ()) pos) ⟩
```

Note that `my_markup` (not shown here) generates the layout attributes of the link and that the `true` flag is used for markup declaring `cid` as a defining occurrence, i.e. as *target* (rather than the *source*) in the hyperlink animation in PIDE.

3 The C11 Parser Generation Process and Architecture

Isabelle uses basically two parsing technologies:

1. Earley parsing [7] intensively used for mixfix-syntax denoting λ-terms in mathematical notation,
2. combinator parsing [9] typically used for high-level command syntax.

Both technologies offer the dynamic extensibility necessary for Isabelle as an interactive platform geared towards incremental development and sophisticated mathematical notations. However, since it is our goal to support *programming languages* in a fast parse-check-eval cycle inside an IDE, we opt for a

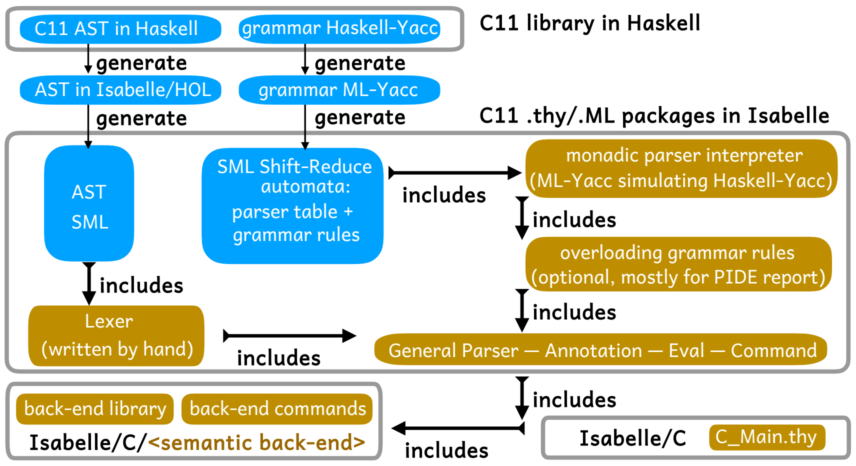

C11 AST in Haskell
grammar Haskell-Yacc
C11 library in Haskell
generate
generate
AST in Isabelle/HOL
grammar ML-Yacc
generate
generate
C11 .thy/.ML packages in Isabelle
AST SML
SML Shift-Reduce automata: parser table + grammar rules
includes
monadic parser interpreter (ML-Yacc simulating Haskell-Yacc)
includes
overloading grammar rules (optional, mostly for PIDE report)
includes
includes
Lexer (written by hand)
includes
General Parser — Annotation — Eval — Command
includes
back-end library
back-end commands
Isabelle/C/<semantic back-end>
includes
Isabelle/C
C_Main.thy

Figure 2: The Architecture of Isabelle/C

Lex and Yacc deterministic grammar approach. It turns out the resulting automata based parser performs well enough for our purpose; the gain in performance is discussed in the next section.

In the following, we describe a novel technique for the construction and integration of this type of parser into the Isabelle platform. Since it is mostly relevant for integrators copying our process to similar languages such as JavaScript or Rust [3], users of the Isabelle/C platform may skip this section: for them, the take-home message is that the overall generation process takes about 1 hour, the compilation of the generated files takes 15s, and that the generated files should be fairly portable to future Isabelle versions.

We base our work on the C11 parsing library `http://hackage.haskell.org/package/language-c` implemented in Haskell by Huber, Chakravarty, Coutts and Felgenhauer; we particularly focus on its open-source Haskell Yacc grammar as our starting point. We would like to emphasize that this is somewhat arbitrary, our build process can be easily adapted to more recent versions when available.

The diagram in Figure 2 presents the architecture of Isabelle/C. The original Haskell library was not modified, it is presented in blue together with generated sources, in particular the final two blue boxes represent about 11 kLoC. In output, the glue code in brown constitutes the core implementation of Isabelle/C, amounting to 6 kLoC (without yet considering semantic back-ends).

3.1 Generating the AST

In the following, we refer to *languages* by $\mathscr{L}$, $\mathscr{I}$. The notation $\mathrm{AST}^{\mathscr{L}}_{\mathscr{I}}$ refers to abstract syntaxes for language $\mathscr{L}$ implemented in language $\mathscr{I}$. For example, we refer by $\mathrm{AST}^{\mathrm{C11}}_{\mathrm{ML}}$ to an AST implementation of C11 implemented in SML. Indices will be dropped when no confusion arises, or to highlight the fact that our approach is sufficiently generic.

For our case, we exploit that from a given Haskell source $\mathrm{AST}_{\mathrm{HS}}$, Haskabelle generates to a maximum extent an Isabelle/HOL theory. Via the Isabelle code generator, an $\mathrm{AST}_{\mathrm{ML}}$ can be obtained from a

[3]E.g. `http://hackage.haskell.org/package/language-javascript` or `http://hackage.haskell.org/package/language-rust`

constructive AST_{HOL} representation. However, the process is challenging for technical reasons in practice due to the enormous size of AST^{C11} (several hundreds of constructors), and due to certain type declarations not initially supported by Haskabelle (we have to implement here the necessary features). Ultimately, the process to compile AST_{HS} to AST_{ML} is done only once at build time, it comprises:

1. the generation of AST_{HOL} from AST_{HS}, represented as a collection of **datatype**,
2. the execution of the **datatype** theory for AST_{HOL} and checking of all their proofs,[4]
3. the generation of an AST_{ML} from AST_{HOL}.

3.2 Constructing a Lexer for C11

We decided against the option of importing the equivalent Haskell lexer, as it is coming under-developed compared to the existing PIDE lexer library, natively supporting Unicode-like symbols (mostly for annotations). Using a more expressive position data-structure, our C lexer is also compatible with the native ML lexer regarding the handling of errors and backtracking (hence the perfect fit when nesting one language inside the other). Overall, the modifications essentially boil down to taking an extreme care of comments and directives which have intricate lexical conventions (see subsection 4.1).

3.3 Generating the Shift-Reduce Parser from the Grammar

In the original C11 library, together with AST_{HS}, there is a Yacc grammar file $G_{\text{HS-YACC}}$ included, which we intend to use to conduct the C parsing. However due to technical limitations of Haskabelle (and advanced Haskell constructs in the associated G_{HS}), we do not follow the same approach as subsection 3.1. Instead, an ultimate grammar G_{ML} is obtained by letting ML-Yacc participate in the generation process. In a nutshell, the overall grammar translation chain becomes: $G_{\text{HS-YACC}} \longrightarrow_{\text{HS}} G_{\text{ML-YACC}} \longrightarrow_{\text{ML}} G_{\text{ML}}$.

$\longrightarrow_{\text{HS}}$ is implemented by modifying the Haskell parser generator Happy, because Happy is already natively supporting the whole $\mathscr{L}_{\text{HS-YACC}}$. Due to the close connection between Happy and ML-Yacc, the translation is even almost linear. However cares must be taken while translating monadic rules [5] of $G_{\text{HS-YACC}}$, as $\mathscr{L}_{\text{ML-YACC}}$ does not support such rules. In G^{C11}, monadic rules are particularly important for scoping analyses, or while building new informative AST nodes (in contrast to disambiguating non-monadic rules, see `@` vs. `&` in section 5). Consequently, applying ML-Yacc $\longrightarrow_{\text{ML}}$ on $G_{\text{ML-YACC}}$ is not enough: after compiling G_{ML} to an efficient Shift-Reduce automaton, we substantially modified the own grammar interpreter of ML-Yacc to implement all features of $\mathscr{L}_{\text{HS-YACC}}$ presented as used in $G_{\text{HS-YACC}}$.

4 Isabelle/C: Syntax Tests and Experimental Results

The question arises, to what extent our construction provides a faithful parser for C11, and if Isabelle/C is sufficiently stable and robust to handle real world sources. A related question is the treatment of `cpp` preprocessing directives: while a minimal definition of the preprocessor is part of C standards since C99, practical implementations vary substantially. Moreover, `cpp` comes close to be Turing complete: recursive computations can be specified, but the expansion strategy bounds the number of unfolding.

[4]Large mutually recursive datatypes in AST_{HOL} might lead to worse performance time, see for instance `https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2016-March/msg00034.html` and `https://lists.cam.ac.uk/pipermail/cl-isabelle-users/2017-April/msg00000.html`.

[5]`https://www.haskell.org/happy/doc/html/sec-monads.html`

Therefore, a complete `cpp` reimplementation contradicts our objective to provide efficient IDE support inside Isabelle. Instead, we restrict ourselves to a common subset of macro expansions and encourage, whenever possible, Isabelle specific mechanisms such as user programmed C annotations. C sources depending critically on a specific `cpp` will have to be processed outside Isabelle. [6]

4.1 Preprocessing Lexical Conventions: Comments and Backslash Newlines

A very basic standard example taken from the GCC / CPP documentation [7] shows the quite intricate mixing of comment styles that represents a challenge for our C lexer. A further complication is that it is allowed and common practice to use backslash-newlines $\backslash\boxed{\hookleftarrow}$ *anywhere* in C sources, be it inside comments, string denotations, or even regular C keywords like $i\backslash\boxed{\hookleftarrow}n\backslash\boxed{\hookleftarrow}t$ (see also Figure 4).

```
C ― ‹Nesting of comments› ‹
/* inside /* inside */ int a = "outside";
// inside // inside until end of line
int a = "outside";
/* inside
  // inside
inside
*/ int a = "outside";
// inside /* inside until end of line
int a = "outside";
›
```

In fact, many C processing tools assume that all comments have already been removed via `cpp` before they start any processing. However, annotations in comments carry relevant information for back-ends as shown in section 5. Consequently, they must be explicitly represented in $\text{AST}^{\text{C11}}_{\text{ML}}$, whereas the initial $\text{AST}^{\text{C11}}_{\text{HS}}$ is not designed to carry such extra information. Annotations inside comments may again contain structured information like programming code, formulas, and proofs, which implies the need for nested syntax. Fortunately, Isabelle is designed to manage multiple parsing layers with the technique of *cascade sources* [8] (see also Figure 3). We exploit this infrastructure to integrate back-end specific syntax and annotation semantics based on the parsing technologies available.

4.2 Preprocessing Side-Effects: Antiquoting Directives vs. Pure Annotations

Whereas *comments* can be safely removed without affecting the meaning of C code, *directives* are semantically relevant for compilation and evaluation.

1. Classical directives: *`#define x TOKS`* makes any incoming C identifier *`x`* be replaced by some *arbitrary* tokens *`TOKS`*, even when included via the *`#include`* directive.

2. Typed (pseudo-)directives as commands: It is easy to overload or implement a new *`#define'`* acting only on a decided subset of well-formed *`TOKS`*. There are actually no differences between Isabelle/C directives and Isabelle commands: both are internally of type $\sigma \rightarrow \sigma$ (see section 2).

3. Non-expanding annotations: Isabelle/C annotations *`/*@`* $\mathscr{L}_{\text{annot}}$ *`*/`* or *`//@`* $\mathscr{L}_{\text{annot}}$ can be freely intertwined between other tokens, even inside directives. In contrast to (antiquoting) directives and similarly as C comments, their designed intent is to not modify the surrounding parsing code.

A limitation of Isabelle and its current document model is that there is no way for user programmed extensions to exploit implicit dependencies between sub-documents. Thus, a sub-document referred to via *`#include <some_file>`* will not lead to a reevaluation of a **C**‹ .. › command whenever modified. (The only workaround is to open all transitively required sub-documents *by hand*.)

[6] Isabelle/C has a particular option to activate (or not) an automated call to `cpp` before any in-depth treatment.

[7] `https://gcc.gnu.org/onlinedocs/cpp/Initial-processing.html`

[8] `http://isabelle.in.tum.de/repos/isabelle/file/83774d669b51/src/Pure/General/source.ML`

4.3 A Validation via the seL4 Test Suite

The AutoCorres environment contains a C99 parser developed by Michael Norrish [12]. Besides a parser test-suite, there is the entire seL4 codebase (written in C99) which has been used for the code verification part of the seL4 project. While the parser in itself represents a component belonging to the trusted base of the environment, it is arguably the most tested parser for a semantically well-understood translation in a proof environment today.

It is therefore a valuable reference for a comparison test, especially since $\mathrm{AST}^{\mathrm{C99}}$ and $\mathrm{AST}^{\mathrm{C11}}$ are available in the same implementation language. From $\mathrm{AST}^{\mathrm{C11}}_{\mathrm{HOL}}$ to $\mathrm{AST}^{\mathrm{C99}}_{\mathrm{HOL}}$ we construct an abstraction function $C^{\downarrow}$. A detailed description of $C^{\downarrow}$ is out of the scope of this paper; we would like to mention that it was 4 man-months of work due to the richness of $\mathrm{AST}^{\mathrm{C11}}$. As such, the abstraction function $C^{\downarrow}$ is at the heart of the AutoCorres integration into our framework described in subsection 6.2. Note that $\mathrm{AST}^{\mathrm{C99}}$ seems to be already an abstraction compared to the C99 standard. This gives rise to a particular testing methodology: we can compile the test suites as well as the seL4 source files by both ML parsers $\mathrm{PARSE}^{\mathrm{C99}}_{stop}$ and $\mathrm{PARSE}^{\mathrm{C11}}_{report}$, abstract the output of the latter via $C^{\downarrow}$ and compare the results.

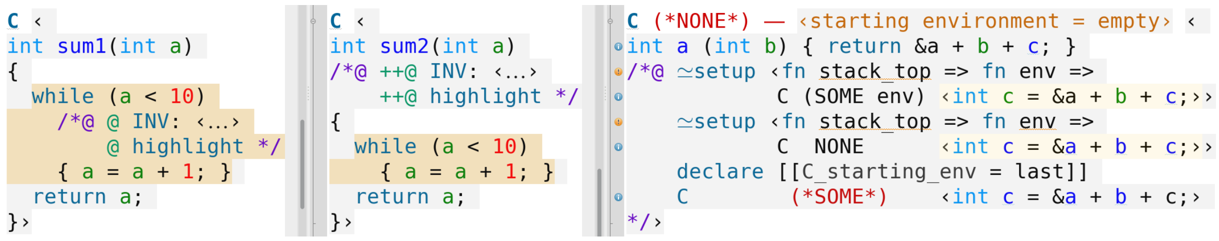
Our test establishes that both parsers agree on the entire seL4 codebase. However trying to compare the two parsers using other criteria is not possible, for example we had to limit ourselves to C programs written in a subset of C99. Fundamentally, the two parsers are achieving different tasks: the one of PARSE_{stop} is to just return a parsed AST. In contrast, PARSE_{report} intends to maximize markup reporting, irrespective of a final parsing success or failure, and reports are provided in parallel during its (monadic) parsing activity. Thus, in the former scenario, the full micro-kernel written in 26 kLoC can be parsed in 0.1s. In the latter, all reports we have thought helpful to implement are totally rendered before 20s. Applying $C^{\downarrow}$ takes 0.02 seconds, so our PARSE_{report} gives an average of 2s for a 2-3 kLoC source. By interweaving a source with proofs referring to the code elements, the responsiveness of PIDE should therefore be largely sufficient.

5 Generic Semantic Annotations for C

With respect to interaction with the underlying proof-engine, there are essentially two lines of thought in the field of deductive verification techniques:

1. either programs and specifications — i.e. the pre- and post-condition contracts — are clearly separated, or
2. the program is annotated with the specification, typically by using some form of formal comment.

Of course, it is possible to inject the essence of annotated specifications directly into proofs, e.g. by instantiating the `while` rule of the Hoare calculus by the needed invariant inside the proof script. The resulting clear separation of programs from proofs may be required by organisational structures in development projects. However, in many cases, modelling information may be interesting for programmers, too. Thus, having pre- and post-conditions locally in the source close to its point of relevance increases its maintainability. It became therefore common practice to design languages with annotations, i.e. structured comments *inside* a programming source. Examples are ACSL standardized by ANSI/ISO (see `https://frama-c.com/download/acsl.pdf`) or UML/OCL [5] for static analysis tools. Isabelle/C supports both the inject-into-proof style and annotate-the-source style in its document model; while the former is kind of the default, we address in this section the necessary technical infrastructure for the latter.



```
C ‹
int sum1(int a)
{
  while (a < 10)
    /*@ @ INV: ‹...›
        @ highlight */
    { a = a + 1; }
  return a;
}›
```

```
C ‹
int sum2(int a)
/*@ ++@ INV: ‹...›
    ++@ highlight */
{
  while (a < 10)
    { a = a + 1; }
  return a;
}›
```

```
C (*NONE*) — ‹starting environment = empty› ‹
int a (int b) { return &a + b + c; }
/*@ ≃setup ‹fn stack_top => fn env =>
             C (SOME env) ‹int c = &a + b + c;››
    ≃setup ‹fn stack_top => fn env =>
             C  NONE      ‹int c = &a + b + c;››
    declare [[C_starting_env = last]]
    C         (*SOME*)    ‹int c = &a + b + c;›
*/›
```

Figure 3: Advanced Annotation Programming

Generally speaking, a generic annotation mechanism which is sufficiently expressive to capture idioms used in, e.g., Frama-C, Why3, or VCC is more problematic than one might think. Consider this:

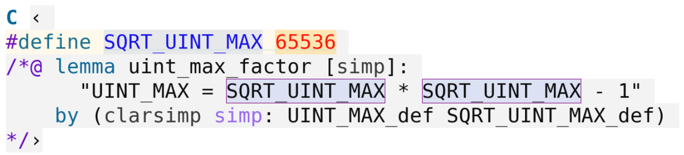
```
for (int i = 0; i < n; i++) a+= a*i /*@ annotation */
```

To which part of the AST does the annotation refer? To `i`? `a*i`? The assignment? The loop? Some verification tools use prefix annotations (as in Why3 for procedure contracts), others even a kind of parenthesis of the form:

```
/*@ annotation_begin */ .../*@ annotation_end */
```

The matter gets harder since the C environment — a table mapping C identifiers to their type and status — changes according to the reference point in the AST. This means that the context relevant to type-check an annotation such as `/*@ assert ⟨a > i⟩ */` strongly differs depending on the annotation's position. And the matter gets even further complicated since Isabelle/C lives inside a proof environment; here, local theory development (rather than bold ad-hoc axiomatizations) is a major concern.

The desire for fast impact analysis resulting from changes may inspire one to annotate local proofs near directives, which is actually what is implemented in our Isabelle/C/AutoCorres example (section 6).



```
C ‹
#define SQRT_UINT_MAX 65536
/*@ lemma uint_max_factor [simp]:
      "UINT_MAX = SQRT_UINT_MAX * SQRT_UINT_MAX - 1"
    by (clarsimp simp: UINT_MAX_def SQRT_UINT_MAX_def)
*/›
```

In the example, the semantic back-end converts the cpp macro into a HOL *definition*, i.e. an extension of the underlying theory context by the conservative axiom `SQRT_UINT_MAX` ≡ `65536` bound to the name `SQRT_UINT_MAX_def`. This information is used in the subsequent proof establishing a new theory context containing the lemma `uint_max_factor` configured to be used as rewrite rule whenever possible in future proofs. This local lemma establishes a relation of `SQRT_UINT_MAX` to the maximally representable number `UINT_MAX` for an unsigned integer according to the underlying memory model.

Obviously, the scheduling of these transformations of the underlying theory contexts is non-trivial.

5.1 Navigation for Annotation Commands

In order to overcome the problem of syntactic ambiguity of annotations, we slightly refine the syntax of semantic annotations by the concept of a navigation expression:

```
𝓛annot = ∅ | <navigation-expr> <annotation-command> 𝓛annot
```

A `<navigation-expr>` string consists of a sequence of `+` symbols followed by a sequence consisting of `@` or `&` symbols. It allows for navigating in the syntactic context, by advancing tokens with several `+`, or taking an ancestor AST node with several `@` (or `&` which only targets monadic grammar rules). This corresponds to a combination of right-movements in the AST, and respectively parent-movements. This way, the "focus" of an `<annotation-command>` can be modified to denote any C fragment of interest.

As a relevant example for debugging, consider Figure 3. The annotation command **highlight** is a predefined Isabelle/C ML-library function that is interpreted as C annotation. Its code is implicitly parameterized by the syntactical context, represented by `stack_top` whose type is a subset of AST^{C11}, and the lexical environment `env` containing the lexical class of identifiers, scopes, positions and serials for markup. The navigation string before **highlight** particularly influences which `stack_top` value gets ultimately selected. The third screenshot in Figure 3 demonstrates the influence of the static environment: an Isabelle/C predefined command ≃**setup** allows for "recursively" calling the C environment itself. This results in the export of definitions in the surrounding logical context, where the propagation effect may be controlled with options like *`C_starting_env`*. ≃**setup** actually mimics standard Isabelle **setup** command, but extends it by `stack_top` and `env` [9]. In the example, the first recursive call uses `env` allowing it to detect that *`b`* is a local parameter, while the second ignores it which results in a treatment as a free global variable. Note that bound global variables are not green but depicted in black.

5.2 Defining Annotation Commands

Extending the default configuration of commands, text and code antiquotations from the Isabelle platform to Isabelle/C is straightforward. For example, the central Isabelle command definition:

```
Outer_Syntax.command: K_cmd -> (σ -> σ) parser -> unit
```

establishes the dynamic binding between a command keyword K_{cmd} = **definition** | **lemma** | … and a parser, whose value is a system transition.[10] The `parser` type stems from the aforementioned parser combinator library: `'a parser = Token.T list -> 'a * Token.T list`.

Analogously, Isabelle/C provides an internal late-binding table for *annotation commands*:

```
C_Annotation.command : K_cmd -> (<navigation-expr> -> R_cmd c_parser) -> unit
C_Annotation.command': K_cmd -> (<navigation-expr> -> R_cmd c_parser) -> σ -> σ
C_Token.syntax': 'a parser -> 'a c_parser
```

where in this paper we define $R_{cmd} = \sigma$ `->` σ as above.[11] Since the type `c_parser` is isomorphic to `parser`, but accepting C tokens, one can use `C_Token.syntax'` to translate and carry the default Isar commands *inside* the **C**⟨ .. ⟩ scope, such as **lemma** or **by**. Using ≃**setup**, one can even define an annotation command **C** taking a C code as argument, as the ML code of ≃**setup** has type α^{AST} `-> env ->` R_{cmd} (which is enough for calling `C_Annotation.command'` in the ML code). Here, whereas the type `env` is always the same, the type $\alpha^{\text{AST}} \subseteq \text{AST}^{\text{C11}}$ varies depending on `<navigation-expr>` (see subsection 5.3).

Note, however, that the user experience of the IDE changes when nesting commands too deeply. In terms of error handling and failure treatment, there are some noteworthy implementation differences between the outermost commands and C annotation commands. Naturally, the PIDE toplevel has been optimized to maximize the error recovery and parallel execution. Inside a command, the possibilities to mimic this behaviour are somewhat limited. As a workaround useful during development and debugging, we offer a further pragma for a global annotation, namely `*` (in complement to the violet `@`), that controls a switch between a strict and a permissive error handling for nested annotation commands.

[9] cf. `https://isabelle.in.tum.de/doc/isar-ref.pdf`

[10] σ has actually the internal Isabelle type `Toplevel.transition`.

[11] In some parallel work, we focus on running commands in native efficient speed with R_{cmd} = `(`K_{cmd}`*(`σ`->`σ`))list`. [19]

5.3 Evaluation Order

We will now explain why positional languages are affecting the evaluation time of annotation commands in Figure 3. This requires a little zoom on how the parsing is actually executed.

The LALR parsing of our implemented C11 parser can be summarized as a sequence of alternations between Shift and Reduce actions. By definition of LALR, whereas a unique Shift action is performed for each C token read from left to right, some unlimited number of Reduce actions are happening between two Shifts. Internally, the parser manages a stack-like data-structure called α^{AST} `list` representing all already encountered Shift and Reduce actions (SR). A given α^{AST} `list` can be seen as a *forest of SR nodes*: all leafs are tagged with a Shift, and any other parent node is a Reduce node. After a certain point in the parsing history, the top stack element α^{AST} (cast with the right type) is returned to ≃**setup**.

Since a SR-forest is a list of SR-trees, it is possible to go forward and backward at will in the actually unparsed SR-history, and execute a sequence of SR parsing steps only when needed. While every annotation command like ≃**setup** is by default attached to a closest previous Shift leaf, navigation expressions modify the attached node, making the presentation of α^{AST} referring to another term focus.

```
C ‹int _;
/*@ @ C ‹//@ C1 ‹int _; //@ @ ≃setup⇓ ‹@{C_def ⇑ C2}› \
                             @ C1  ‹//* C2 ‹int _;››   \
                             @ C1⇓ ‹//* C2 ‹int _;››    ››
   @ C ‹//* C2 ‹int _;›                                  ›
     ≃setup ‹@{C_def ⇑ (* bottom-up *)  C1  }›
     ≃setup ‹@{C_def ⇓ (* top-down  *) "C1⇓"}›
*/›
```

Instead of visiting the AST in the default bottom-up direction during parsing, it is possible to store the intermediate results, so that it can be revisited by using another direction strategy, for example top-down after parsing (where a parent node is executed before any of its children, and knows how they have been parsed thanks to α^{AST}). This enables commands to decide if they want to be executed during parsing, or after the full AST has been built. This gives rise to the implementation of different versions of annotation commands that are executed at different moments, relative to the parsing process. For example, the annotation command ≃**setup** has been defined for being executed at bottom-up time, whereas the execution of the variant ≃**setup**⇓ happens at top-down time. In the above example, **C1** is a new command defined by `C_def`, a shorthand antiquotation for `C_Annotation.command'`. Since **C1** is meant to be executed during bottom-up time (during parsing), it is executed before **C2** is defined (which is directly after parsing).

Note that the C11 grammar has enough scoping structure for the full inference of the C environment `env` be at bottom-up time. In terms of efficiency, we use specific *static* rule wrappers having the potential of overloading default grammar rules (see Figure 2), to assign a wrapper to be always executed as soon as a Shift-Reduce rule node of interest is encountered. The advantage of this construction is that the wrappers are statically compiled, which results in a very efficient reporting of C type information.

6 Semantic Back-Ends

In this section, we briefly present two integrations of verification back-ends for C. We chose Clean used for program-based test generation [10], and AutoCorres [8], arguably the most developed deductive verification environment for machine-oriented C available at present.

Note that we were focusing on keeping modifications of integrated components minimal, particularly for the case of AutoCorres. Certain functionalities like position propagation of HOL terms in annotations, or "automatic" incremental declarations [12] may require internal revisions on the back-end side. This is out of the scope of this paper.

[12] `https://github.com/seL4/l4v/blob/master/tools/autocorres/tests/examples/Incremental.thy`

6.1 A Simple Typed Memory Model: Clean

Clean (pronounced as: “Céline” [selin]) is based on a simple, shallow-style execution model for an imperative target language. It is based on a “no-frills” state-exception monad **type_synonym** *('o, 'σ) MON*$_{SE}$ = ⟨*'σ ⇀ ('o × 'σ)*⟩ with the usual definitions of *bind* and *unit*. In this language, sequence operators, conditionals and loops can be integrated. From a concrete program, the underlying state *'σ* is constructed by a sequence of extensible record definitions:

1. Initially, an internal control state is defined to give semantics to *break* and *return* statements:

   ```
   record control_state = break_val :: bool return_val :: bool
   ```

   control_state represents the σ_0 state.

2. Any global variable definition block with definitions $a_1 : \tau_1 \ \dots \ a_n : \tau_n$ is translated into a record extension:

   ```
   record σ_{n+1} = σ_n + a_1 :: τ_1;  ...; a_n :: τ_n
   ```

3. Any local variable definition block (as part of a procedure declaration) with definitions $a_1 : \tau_1 \dots a_n : \tau_n$ is translated into the record extension:

   ```
   record σ_{n+1} = σ_n + a_1 :: τ_1 list;  ...; a_n :: τ_n list; result :: τ_{result−type} list;
   ```

   where the *list*-lifting is used to model a *stack* of local variable instances in case of direct recursions and the *result* used for the value of the *return* statement.

The **record** package creates an *'σ* extensible record type *'σ control_state_ext* where the *'σ* stands for extensions that were subsequently “stuffed” in them. Furthermore, it generates definitions for the constructor, accessor and update functions and automatically derives a number of theorems over them (e.g., “updates on different fields commute”, “accessors on a record are surjective”, “accessors yield the value of the last update”). The collection of these theorems constitutes the *memory model* of Clean. This model might be wrong in the sense that it does not reflect the operational behaviour of a particular compiler, however, it is by construction *logically consistent* since it is impossible to derive falsity from the entire set of rules.

On this basis, assignments, conditionals and loops are reformulated into *break*-aware and *return*-aware versions as shown in the figure aside. The Clean theory contains about 600 derived theorems containing symbolic evaluation and Hoare-style verification rules.

```
definition assign :: "('σ control_state_scheme  ⇒
                       'σ control_state_scheme) ⇒
                      (unit, 'σ control_state_scheme) MON_SE"
  where     "assign 𝒰 = (λσ. if break_val σ ∨ return_val σ
                               then Some((), σ) else Some((), 𝒰 σ))"

definition if_Clean :: "['σ control_state_ext ⇒ bool,
                         ('β, 'σ control_state_ext) MON_SE,
                         ('β, 'σ control_state_ext) MON_SE] ⇒
                        ('β, 'σ control_state_ext) MON_SE"
  where     "if_Clean ℬ T F = (λσ. if break_val σ ∨ return_val σ
                                  then Some (undefined, σ) —
                                ‹state unchanged, return arbitrary›
                                  else if ℬ σ then T σ else F σ)"
```

Importing Clean into a theory with its activated back-end proceeds as in Figure 4. Clean generates for the C program a common type for the state, based on two generated extensible records — in the figure: just a global variable *k* and a local variable with a stack of result values for *prime*$_C$. Clean maps machine integers simply and naively on the HOL type *int*. The core of this program is represented by two generated definitions available subsequently in the logical context, where they are ready to be used in symbolic executions or proofs.

```
theory Prime imports Isabelle_C_Clean.Backend
                     — ‹Clean back-end is now on›
begin
C ‹
//@ definition ‹primeHOL (p :: nat) =                  \
          (1 < p ∧ (∀ n ∈ {2..<p}. ¬ n dvd p))›
#   define SQRT_UINT_MAX 65536
unsigned int k = 0;
unsigned int primeC(unsigned int n) {
//@ preClean  ‹C‹n› ≤ UINT_MAX›
//@ postClean ‹C‹primeC(n)› ≠ 0 ⟷ primeHOL C‹n››
    if (n < 2) return 0;
    for (unsigned i = 2; i < SQRT_UINT_MAX
                      && i * i <= n; i++) {
      if (n % i == 0) return 0;
      k++;
    }
    return 1;
}›
```

```
primeC_core_def: "primeC_core n ≡
  ifClean ‹(n < 2) ›then return 0 else skip;-
  ‹ i := 2 ›;-
  whileClean ‹i < SQRT_UINT_MAX ∧ i * i ≤ n›
    (ifClean ‹n mod i = 0›
     then return 0 else skip;
      ‹k:=k+1›; assert ‹k≤UINT_MAX ›
      ‹i:=i+1›; assert ‹i≤UINT_MAX ›) ;-
  return 1"

primeC_def: "primeC n ≡
  blockClean push_local_primeC_state
             (is_prime_core n)
             pop_local_primeC_state"
```

Figure 4: Isabelle/C/Clean Semantic Back-End on an Example

Generated definitions include push and pop operations for local variable blocks, for the entire variable space of procedures. Additionally, a specific syntax is introduced to represent assignments on global and local variables. For example, *i := 2* internally rewrites to *assign (λσ. ((i_upd o map_hd) (λ_. 2)) σ)*. The *return* operation is syntactically equivalent to the assignment of the result variable in the local state (stack) and sets the *return_val* flag. On this representation of the C program, the HOL term $prime_C$ *n* can be decomposed into program test-cases according to a well-established coverage criterion. Technically, this is done by a variant of the program-based testing method

```
apply (branch_and_loop_coverage "Suc (Suc (Suc 0))")
```

developed in [10], which also uses Clean as semantic basis. Note that the testing approach does not need the formulation of an invariant, which is already non-trivial in the given example.

Finally, we will have a glance at the code for the registration of the annotation commands used in the example. Thanks to Isabelle/C's function `C_Annotation.command'`, the registration of user-defined annotations is very similar to the registration of ordinary commands in the Isabelle platform.

```
ML ‹fun command keyword f =
  C_Annotation.command' keyword ""
    (C_Token.syntax'
      (Parse.token Parse.cartouche)
     >>> toplevel f)›
setup ‹command ("preClean", ⌂) Spec
    #> command ("postClean", ⌂) End_spec
    #> command ("invClean", ⌂) Invariant›
```

6.2 The Case of AutoCorres

The AutoCorres environment consists of a C99 parser, compiling to a deepish embedding of a generic imperative core programming language, over a refined machine word oriented memory model, and a translator of this presentation into a shallow language based on another Monad for non-deterministic computations. This translator has been described in [8, 22] in detail. However, the original use of AutoCorres implies a number of protocol rules to follow, and is only loosely integrated into the Isabelle document model, which complicates the workflow substantially.

Our running example *prime*$_C$ for Isabelle/C/AutoCorres basically differs in what the theory is importing in its header. Similarly to Clean, AutoCorres constructs a memory model and represents the program as a monadic operation on it. Actually, it generates even two presentations, one on a very precise word-level memory model taking aspects of the underlying processor architecture into account, and another one more abstract, then it automatically proves the correspondence in our concrete example. Both representations become the definitions *prime*$_C$*_def* and *prime*$_C$*'_def*. A Hoare-calculus plus a derived verification generator *wp* from the AutoCorres package leverage finally the correctness proof.

```
theory Prime
  imports Isabelle_C_AutoCorres.Backend
begin
C ‹
   ⋮
theorem (in primeC) primeC'_correct:
  ‹⦃ λ _. n ≤ UINT_MAX ⦄ primeC' n
   ⦃ λ primeC' _. primeC' ≠ 0 ⟷ primeHOL n ⦄!›
proof (rule validNF_assume_pre)
 assume 1: ‹n ≤ UINT_MAX›
 have   2: ‹n = 0 ∨ n = 1 ∨ n > 1› by linarith
 show ?thesis
  proof (insert 2, elim disjE)
   assume ‹n = 0›
   then show ?thesis
        by (clarsimp simp: primeC'_def, wp, auto)
  next
   ⋮
```

Note that the integration of AutoCorres crucially depends on the conversion $\mathrm{AST}^{\mathrm{C11}} \Rightarrow \mathrm{AST}^{\mathrm{C99}}$ of $C^{\downarrow}$ discussed in subsection 4.3. In particular, for the overall seL4 annotations **INVARIANT**, **INV**, **FNSPEC**, **RELSPEC**, **MODIFIES**, **DONT_TRANSLATE**, **AUXUPD**, **GHOSTUPD**, **SPEC**, **END-SPEC**, **CALLS**, and **OWNED_BY**, we have extended our implementation of $C^{\downarrow}$ in such a way that the conversion places the information at the right position in the target AST. Obviously, this works even when navigation is used, as in Figure 3 left.

7 Conclusions

We presented Isabelle/C a novel, generic front-end for a deep integration of C11 code into the Isabelle/PIDE framework. Based on open-source Lex and Yacc style grammars, we presented a build process that constructs key components for this front-end: the lexer, the parser, and a framework for user-defined annotations including user-defined annotation commands. While the generation process is relatively long, the generated complete library can be loaded in a few seconds constructing an environment similar to the usual **ML** environment for Isabelle itself. 20 kLoC large C sources can be parsed and decorated in PIDE within seconds.

Our framework allows for the deep integration of the C source into a global document model in which literate programming style documentation, modelling as well as static program analysis and verification co-exist. In particular, information from the different tools realized as plugin in the Isabelle platform can flow freely, but based on a clean management of their semantic context and within a framework based on conservative theory development. This substantially increases the development agility of such type of sources and may be attractive to conventional developers, in particular when targeting formal certification [4].

Isabelle/C also forms a basis for future semantically well-understood combinations of back-ends based on different semantic interpretations: inside Isabelle, bridge lemmas can be derived that describe the precise conditions under which results from one back-end can be re-interpreted and used in another. Future tactic processes based on these bridge lemmas may open up novel ways for semantically safe tool combinations.

Acknowledgments. The authors warmly thank David Sanán and Yang Liu for encouraging the development and reuse of $C^{\downarrow}$, initially started in the Securify project [18] (`http://securify.sce.ntu.edu.sg/`).

References

[1] Romain Aïssat, Frédéric Voisin & Burkhart Wolff (2016): *Infeasible Paths Elimination by Symbolic Execution Techniques - Proof of Correctness and Preservation of Paths*. In: *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, pp. 36–51. `https://doi.org/10.1007/978-3-319-43144-4_3`.

[2] Bruno Barras, Lourdes Del Carmen González-Huesca, Hugo Herbelin, Yann Régis-Gianas, Enrico Tassi, Makarius Wenzel & Burkhart Wolff (2013): *Pervasive Parallelism in Highly-Trustable Interactive Theorem Proving Systems*. In Jacques Carette, David Aspinall, Christoph Lange, Petr Sojka & Wolfgang Windsteiger, editors: *Intelligent Computer Mathematics - MKM, Calculemus, DML, and Systems and Projects 2013, Held as Part of CICM 2013, Bath, UK, July 8-12, 2013. Proceedings*, *Lecture Notes in Computer Science* 7961, Springer, pp. 359–363. `https://doi.org/10.1007/978-3-642-39320-4_29`.

[3] Joshua A Bockenek, Peter Lammich, Yakoub Nemouchi & Burkhart Wolff (2018): *Using Isabelle/UTP for the Verification of Sorting Algorithms A Case Study*. `https://easychair.org/publications/preprint/CxRV`. Isabelle Workshop 2018, Colocated with Interactive Theorem Proving. As part of FLOC 2018, Oxford, GB.

[4] Achim D. Brucker, Idir Aït-Sadoune, Paolo Crisafulli & Burkhart Wolff (2018): *Using the Isabelle Ontology Framework - Linking the Formal with the Informal*. In: *Intelligent Computer Mathematics - 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings*, pp. 23–38. `https://doi.org/10.1007/978-3-319-96812-4_3`.

[5] Achim D. Brucker, Frédéric Tuong & Burkhart Wolff (2014): *Featherweight OCL: A Proposal for a Machine-Checked Formal Semantics for OCL 2.5*. *Archive of Formal Proofs* 2014. `https://www.isa-afp.org/entries/Featherweight_OCL.shtml`.

[6] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte & Stephan Tobies (2009): *VCC: A Practical System for Verifying Concurrent C*. In: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pp. 23–42. `https://doi.org/10.1007/978-3-642-03359-9_2`.

[7] Jay Earley (1970): *An Efficient Context-Free Parsing Algorithm*. *Commun. ACM* 13(2), pp. 94–102. `https://doi.org/10.1145/362007.362035`.

[8] David Greenaway, Japheth Lim, June Andronick & Gerwin Klein (2014): *Don't sweat the small stuff: formal verification of C code without the pain*. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pp. 429–439. `http://doi.acm.org/10.1145/2594291.2594296`.

[9] Graham Hutton (1992): *Higher-Order Functions for Parsing*. *J. Funct. Program.* 2(3), pp. 323–343. `https://doi.org/10.1017/S0956796800000411`.

[10] Chantal Keller (2018): *Tactic Program-Based Testing and Bounded Verification in Isabelle/HOL*. In: *Tests and Proofs - 12th International Conference, TAP 2018, Held as Part of STAF 2018, Toulouse, France, June 27-29, 2018, Proceedings*, pp. 103–119. `https://doi.org/10.1007/978-3-319-92994-1_6`.

[11] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby C. Murray, Thomas Sewell, Rafal Kolanski & Gernot Heiser (2014): *Comprehensive formal verification of an OS microkernel*. *ACM Trans. Comput. Syst.* 32(1), pp. 2:1–2:70. `http://doi.acm.org/10.1145/2560537`.

[12] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch & Simon Winwood (2009): *seL4: formal verification of an OS kernel*. In Jeanna Neefe Matthews & Thomas E. Anderson, editors: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, ACM, pp. 207–220. `https://doi.org/10.1145/1629575.1629596`.

[13] Peter Lammich & Simon Wimmer (2019): *IMP2 - Simple Program Verification in Isabelle/HOL*. *Archive of Formal Proofs* 2019. `https://www.isa-afp.org/entries/IMP2.html`.

[14] Dirk Leinenbach & Thomas Santen (2009): *Verifying the Microsoft Hyper-V Hypervisor with VCC*. In: *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, pp. 806–809. `https://doi.org/10.1007/978-3-642-05089-3_51`.

[15] Xavier Leroy (2009): *Formal verification of a realistic compiler*. *Commun. ACM* 52(7), pp. 107–115. `http://doi.acm.org/10.1145/1538788.1538814`.

[16] CEA LIST (2019): *The Frama-C Home Page*. `https://frama-c.com`. Accessed March 24, 2019.

[17] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. *Lecture Notes in Computer Science* 2283, Springer. `https://doi.org/10.1007/3-540-45949-9`.

[18] David Sanán, Yongwang Zhao, Zhe Hou, Fuyuan Zhang, Alwen Tiu & Yang Liu (2017): *CSimpl: A Rely-Guarantee-Based Framework for Verifying Concurrent Programs*. In Axel Legay & Tiziana Margaria, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, *Lecture Notes in Computer Science* 10205, pp. 481–498. `https://doi.org/10.1007/978-3-662-54577-5_28`.

[19] Frédéric Tuong & Burkhart Wolff (2015): *A Meta-Model for the Isabelle API*. *Archive of Formal Proofs* 2015. `https://www.isa-afp.org/entries/Isabelle_Meta_Model.shtml`.

[20] Makarius Wenzel (2014): *Asynchronous User Interaction and Tool Integration in Isabelle/PIDE*. In Gerwin Klein & Ruben Gamboa, editors: *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, *Lecture Notes in Computer Science* 8558, Springer, pp. 515–530. `https://doi.org/10.1007/978-3-319-08970-6_33`.

[21] Makarius Wenzel (2014): *System description: Isabelle/jEdit in 2014*. In Christoph Benzmüller & Bruno Woltzenlogel Paleo, editors: *Proceedings Eleventh Workshop on User Interfaces for Theorem Provers, UITP 2014, Vienna, Austria, 17th July 2014.*, *EPTCS* 167, pp. 84–94. `https://doi.org/10.4204/EPTCS.167.10`.

[22] Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David Cock & Michael Norrish (2009): *Mind the Gap*. In Stefan Berghofer, Tobias Nipkow, Christian Urban & Makarius Wenzel, editors: *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, *Lecture Notes in Computer Science* 5674, Springer, pp. 500–515. `https://doi.org/10.1007/978-3-642-03359-9_34`.

A Component-Based Formal Language Workbench

Peter D. Mosses
Delft University of Technology, Delft, The Netherlands
p.d.mosses@tudelft.nl

The CBS framework supports component-based specification of programming languages. It aims to significantly reduce the effort of formal language specification, and thereby encourage language developers to exploit formal semantics more widely. CBS provides an extensive library of reusable language specification components, facilitating co-evolution of languages and their specifications.

After introducing CBS and its formal definition, this short paper reports work in progress on generating an IDE for CBS from the definition. It also considers the possibility of supporting component-based language specification in other formal language workbenches.

1 Introduction

Developers of major programming languages always give *formal* specifications of *syntax*. For *semantics*, however, they usually resort to *informal explanations*. They sometimes define the formal semantics of sublanguages, but scaling up to full languages is usually regarded as a huge effort, and not worthwhile.

To encourage language developers to specify formal semantics of their full languages, it is essential to reduce the effort required – not only for an initial specification, but also for co-evolution of language specifications with the specified languages. The CBS framework aims to do just that, by providing an extensible library of *reusable language specification components*. The semantics of the components is defined once and for all, so simply translating a programming language to compositions of components specifies the language semantics. And specifying such translations can be significantly less effort than specifying language semantics directly.

Crucially, the definition of each component in the CBS library can be validated independently: adding new components to the library cannot invalidate proven properties of previous components. After a component has been validated and released, its defined behaviour cannot be changed, so each occurrence of a particular component name in CBS specifications refers to the same definition. When a specified language evolves, its translation to components always has to change accordingly, as the components themselves cannot change.

Use of CBS is supported by an IDE for editing and validating specifications of components and languages. Validation is currently based on testing prototype implementations generated from specifications. The IDE is itself generated from a formal definition of CBS, which is specified in declarative meta-languages provided by the Spoofax language workbench. CBS and its IDE have been developed by the PLanCompS project.[1]

Here, we give a progress report on the formal definition of CBS (§3). We explain how an IDE for CBS is generated from the definition (§4), and how editors and interpreters for programming languages are generated from their specifications in CBS (§5). We also compare CBS with some other language specification frameworks regarding the possibility of defining libraries of reusable components (§6).

[1] Programming Language Components and Specifications, http://plancomps.org.

Submitted to:
F-IDE 2019

© P.D. Mosses
This work is licensed under the
Creative Commons Attribution License.

2 Component-based specification of programming languages

We start by briefly recalling the main features of the CBS framework for component-based specification. For more detailed expositions, see [5, 7, 10, 15].

A CBS for a programming language is a specification of an inductively-defined translation function, mapping well-formed program phrases to terms formed from so-called *fundamental programming constructs* ('funcons'). The specification includes a *grammar* for the (concrete and abstract) syntax of the language, and a *translation equation* for each alternative of the grammar. The semantics of funcons, together with the translation of programs to funcon terms, determines the semantics of the programs. The funcon definitions are reusable components.

```
PL ===============>  |---------|   transition
...  translations    | Funcons | --------------> Computations
PL ===============>  |---------|      rules
```

Funcons. A funcon usually corresponds to a simple ingredient of constructs commonly found in mainstream programming languages. For example, an assignment in a program might be an expression, returning either the target variable or the assigned value; the funcon for assigning a value to a variable simply has that effect, and does not return either of its arguments. Loops in programs can often be terminated abruptly by break statements; the funcons for loops simply propagate abrupt termination of their bodies, and other funcons are used to handle abrupt termination, when needed. Funcons for integer operations return the unbounded mathematical results, leaving it to other funcons to enforce boundedness. Etc.

CBS uses a modular variant of structural operational semantics (MSOS) [9, 13] to define funcons. Implicit propagation of unmentioned semantic entities (environments, stores, etc.) provides conciseness as well as modularity. Funcons are defined independently of any particular programming language, and of each other. Adding new funcons to the CBS library does not require any changes at all to previous definitions.

Syntax. CBS includes a variant of BNF context-free grammars, extended with standard notation for regular expressions. Concise grammars for expression syntax are usually highly ambiguous; CBS allows disambiguation (associativity, relative priority, etc.), so the same grammar can be used for both abstract and concrete syntax. Alternative productions for the same nonterminal are grouped together, and AST constructors are left implicit. Lexical syntax can be context-free (e.g., for nested comments).

Translation functions. The equations used in CBS to specify translations from program ASTs to funcons look like semantic equations in denotational semantics: the left side is an application of a translation function to an AST pattern, with meta-variables matching sub-trees; the right side is a funcon term containing applications of translation functions to meta-variables. The meta-variables range over specified sorts of sub-trees. Syntactic desugaring equations relate pairs of (composite) AST patterns.

Figure 1 illustrates the notation used in CBS for specifying funcons, syntax, and translation functions. The signature of the funcon if-true-else implies that its first argument is to be pre-evaluated; the two rules specify its subsequent reduction to one of its two unevaluated arguments. The syntax for the sort exp introduces *Exp* as the stem of meta-variables ranging over phrases of sort exp. The rule for the translation function *rval* (signature omitted here) maps ASTs of expressions of the form 'Exp_1 && Exp_2' to funcon terms formed from if-true-else, the translations of Exp_1 and Exp_2, and the funcon false.

Funcon **if-true-else**(_ : **booleans**, _ : T, _ : T) : T
Rule **if-true-else**(**true**, X, _) $\rightsquigarrow X$
Rule **if-true-else**(**false**, _, Y) $\rightsquigarrow Y$

Syntax Exp : exp ::= exp '&&' exp

Rule $\mathit{rval}[\![\mathit{Exp}_1$ '&&' $\mathit{Exp}_2]\!]$ = **if-true-else**($\mathit{rval}[\![\mathit{Exp}_1]\!]$, $\mathit{rval}[\![\mathit{Exp}_2]\!]$, **false**)

Figure 1: A funcon, and fragments of a language specification using it [5]

3 Definition of the CBS meta-language

Syntax. We use SDF3 [20] to define the context-free syntax of CBS. In SDF3, the alternative forms for each sort of CBS construct are written as separate productions, as illustrated in Fig. 2. The form of lexical productions is illustrated in Fig. 3. SDF3 disambiguation includes associativity and relative priority specification for context-free syntax, and rejections and follow-restrictions for lexical syntax.

```
CONCLUSION.FormulaDynamic =
  CONTEXT? STATE DYNAMIC STATE
PREMISE.FormulaDynamic =
  CONTEXT? STATE DYNAMIC STATE
```

Figure 2: Illustrating context-free syntax of CBS [14, CBS/syntax/CBS-RULE]

```
lexical syntax
  VAR-STEM = [A-Z] [a-zA-Z]* ([\-] [a-zA-Z]+)* ([\-] [0-9])?
  VAR-STEM = KEYWORD {reject}
```

```
lexical restrictions

  VAR-STEM -/- [a-zA-Z\-]
```

Figure 3: Illustrating lexical syntax of CBS [14, CBS/syntax/CBS-LEX]

The SDF3 definition of the syntax of CBS has not only a formal interpretation (based on its transformation to SDF2 and the formal definition of SDF2 [21]) but also tool support for parser generation [22], which allows empirical validation of the grammar.

Static analysis. We have used NaBL2 [4] to define name resolution for CBS. NaBL2 rules map ASTs to sets of constraints involving generated scope graphs. Figure 4 illustrates an NaBL2 rule that constrains the use of variables in the premises and conclusions of CBS rules to be 'source dependent'. Other rules check that each funcon has a unique definition, and that all funcon names used in language specifications are defined.

```
Conclusion[[ FormulaDynamic(context_q, state1, dynamic, state2) ^ (s, s') ]] :=
  Decl[[ context_q ^ (s) ]],
  Decl[[ state1 ^ (s) ]],
  Conclusion[[ dynamic ^ (s, s') ]],
  [[ state2 ^ (s') ]].
```

Figure 4: Illustrating name resolution of CBS [14, CBS/trans/static-semantics/CBS-RULE]

NaBL2 is not sufficiently expressive to define the full type checking of CBS, which involves structural subtyping. We have been able to define rules for arity-checking; this includes checking that partial funcon applications are not used as arguments unless allowed by the declared argument types. Our NaBL2 rules also check whether the syntax patterns used in specifications of translation functions match the specified grammar alternatives. NaBL2 is currently being superseded by a more powerful meta-language, Statix [3], which will allow the definition of rigorous type checking for CBS.

Interpretation. The intended interpretation of funcon definitions in CBS is based on their translation to MSOS [13], and thereby as value-computation transition systems [9]. The translation from a precursor of CBS (MSDF) to MSOS was originally defined in Prolog [10], and validated empirically by its use in generating prototype implementations from component-based language specifications. The CBS tool chain now uses Haskell to translate CBS declaratively to monadic Haskell code [5]; the monads involved correspond directly to the entities used in MSOS.

CBS uses context-free grammars (with standard notation for regular expressions) to specify the abstract syntax of programming languages; the interpretation of such grammars as datatypes of ASTs is well established. To specify concrete syntax, CBS currently allows the use of SDF3 notation [20] for associativity, (relative) priority, and follow-restrictions. Note that the interpretation of disambiguation in SDF3 has recently been improved [19, Ch. 2].

The equations used in CBS to specify translations of programs to funcon terms are interpreted inductively as functions on ASTs, as usual.

4 Generation of the CBS IDE

The IDE for CBS is implemented in Spoofax [22], which is an Eclipse plug-in. Spoofax generates the IDE directly from the definition of CBS: it generates a parser for CBS from its SDF3 grammar (with automatic error recovery, and syntax highlighting); it generates name resolution and arity checking constraints for CBS from the NaBL2 rules. Parsing and analysis errors in CBS specifications are flagged in the Eclipse project navigator panel, and in editor windows on erroneous files.

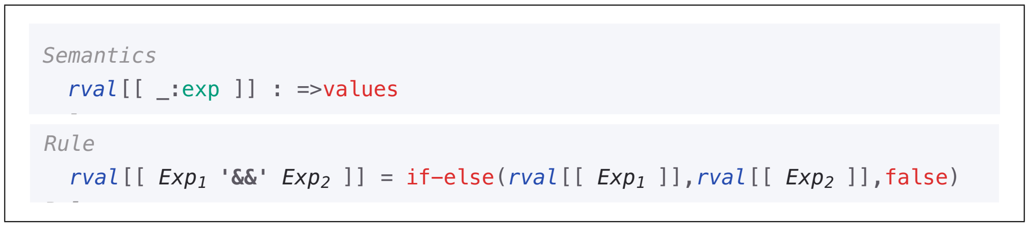
Each language specification in CBS is a separate Eclipse project, with shared access to the funcons library, which is stored in a remote repository. A language specification can be split into (optionally numbered and hierarchical) sections, and stored in any number of files. Multi-file name resolution generates hyperlinks from names to their definitions; funcon names are global, whereas syntactic sorts, translation functions, and meta-variables are local to the enclosing language specifications. Paths to files are never mentioned in CBS specifications, so the distribution of items between files is irrelevant.

5 Generation of language interpreters from CBS

The CBS IDE has a menu to generate various artefacts from a language specification: a syntax-aware editor to parse programs and translate them to funcons, a list of reused funcons, and Markdown pages with embedded HTML for hyperlinks and highlighting. When a language specification is changed, stale files are regenerated by clicking a menu, then the editor project is rebuilt.

The Haskell package *funcon.tools* [6] is used to generate Haskell code for funcon interpreters from the CBS definitions of the funcons. The generated interpreters are highly modular [5].

GitHub generates a responsive website [17] from the generated Markdown pages, allowing language specifications and funcon definitions to be conveniently browsed (online or offline) without installation of Eclipse or Spoofax. Figure 5 shows a fragment of a generated web page.



```
Semantics
  rval[[ _:exp ]] : =>values

Rule
  rval[[ Exp1 '&&' Exp2 ]] = if-else(rval[[ Exp1 ]],rval[[ Exp2 ]],false)
```

Figure 5: A fragment of a generated web page, from [17, Languages-beta/SIMPLE]

6 Related work

CBS appears to be the only framework that currently provides a library of reusable components of programming language specifications. Here, we consider whether similar libraries could be specified in other frameworks. Porting the CBS funcon library to another framework is attractive, as it would make that framework's tool support available to CBS users.

The K-framework [18] is highly modular, and has been used to specify the semantics of several major languages. The funcons used in the CBS specification of SIMPLE [17, Languages-beta/SIMPLE] have already been re-specified in K, allowing the CBS of SIMPLE programs to be run with the K tools [16]. However, the specification of the K configurations was monolithic, and depended on the set of funcons.

Reduction semantics is a popular form of operational semantics, with mature tool support including Redex [12]. Language extensions can be specified smoothly in Redex, but it is unclear how to define an open-ended collection of funcons: reduction semantics requires grammars for evaluation contexts, and the evaluation contexts for a particular funcon seem likely to depend on which other funcons are needed.

XASM [1] is a component-based language for Abstract State Machines [8], supporting the use of Montages [2] for specifying programming languages. However, it appears that the original home page (`xasm.org`) and the subsequent SourceForge project (`xasm.sourceforge.net`) are no longer in use, and that no library of reusable XASM components has ever been released.

The Overture F-IDE platform [11] supports dialects of VDM. Those specification languages contain many constructs which, like funcons, correspond closely to constructs of high-level programming languages: assignment statements, while-loops, exception-handling, etc. Thus translating a programming language to a VDM dialect would be similar to specifying its CBS. It could be interesting to investigate the possibility of defining an extensible library of funcons in VDM, with IDE support in Overture.

7 Conclusion

The syntax and static semantics of CBS have been formally defined [14] using declarative meta-languages: SDF3 and NaBL2. The NaBL2 definition of CBS type checking needs to be reformulated in the new Statix meta-language [3], to specify rigorous subtype checks, after which the IDE generated from the definition of CBS is to be released as an Eclipse plugin. The responsiveness of the IDE should be significantly improved by the incremental implementation of Statix analysis.

Current examples of language specifications in CBS include OCaml-Light; scaling up to full OCaml (and other major languages) remains to be demonstrated. Bisimulation properties of funcons can be proved once and for all [9]; the proofs could be included with the definitions in the library.

The CBS development is part of the PLanCompS project. The project was initially supported by an EPSRC grant (2011–16), and it is continuing as an open collaboration; new participants are welcome.

References

[1] M. Anlauff (2000): *XASM – An Extensible, Component-Based ASM Language*. In: *ASM 2000*, *LNCS* 1912, Springer, pp. 69–90, doi:10.1007/3-540-44518-8_6.

[2] M. Anlauff, P.W. Kutter & A. Pierantonio (1999): *Tool Support for Language Design and Prototyping with Montages*. In: *CC'99*, *LNCS* 1575, Springer, pp. 296–299, doi:10.1007/978-3-540-49051-7_22.

[3] H. van Antwerpen, C. Bach Poulsen, A. Rouvoet & E. Visser (2018): *Scopes As Types*. *Proc. ACM Program. Lang.* 2, pp. 114:1–114:30, doi:10.1145/3276484.

[4] H. van Antwerpen et al. (2016): *A Constraint Language for Static Semantic Analysis Based on Scope Graphs*. In: *Proc. PEPM 2016*, ACM, pp. 49–60, doi:10.1145/2847538.2847543.

[5] L.T. van Binsbergen, P.D. Mosses & N. Sculthorpe (2019): *Executable Component-Based Semantics*. *J. Log. Algebr. Meth. Program.* 103, pp. 184–212, doi:10.1016/j.jlamp.2018.12.004.

[6] L.T. van Binsbergen & N. Sculthorpe (2019): *funcons-tools: A Modular Interpreter for Executing Funcons*. Available at `https://hackage.haskell.org/package/funcons-tools`. Hackage package.

[7] L.T. van Binsbergen, N. Sculthorpe & P.D. Mosses (2016): *Tool Support for Component-Based Semantics*. In: *Companion Proc. Modularity 2016*, ACM, pp. 8–11, doi:10.1145/2892664.2893464.

[8] E. Börger (2017): *The Abstract State Machines Method for Modular Design and Analysis of Programming Languages*. *J. Logic Comput.* 27, pp. 417–439, doi:10.1093/logcom/exu077.

[9] M. Churchill & P.D. Mosses (2013): *Modular Bisimulation Theory for Computations and Values*. In: *FOSSACS 2013*, *LNCS* 7794, Springer, pp. 97–112, doi:10.1007/978-3-642-37075-5.

[10] M. Churchill, P.D. Mosses, N. Sculthorpe & P. Torrini (2015): *Reusable Components of Semantic Specifications*. *LNCS Trans. Aspect Oriented Softw. Dev.* 12, pp. 132–179, doi:10.1007/978-3-662-46734-3_4.

[11] L.D. Couto, P. Gorm Larsen, M. Hasanagic, G. Kanakis, K. Lausdahl & P.W.V. Tran-Jørgensen (2015): *Towards Enabling Overture as a Platform for Formal Notation IDEs*. In: *F-IDE 2015*, *EPTCS* 187, pp. 14–27, doi:10.4204/EPTCS.187.

[12] C. Klein et al. (2012): *Run Your Research: On the Effectiveness of Lightweight Mechanization*. In: *POPL 2012*, ACM, pp. 285–296, doi:10.1145/2103656.2103691.

[13] P.D. Mosses (2004): *Modular Structural Operational Semantics*. *J. Log. Algebr. Program.* 60-61, pp. 195–228, doi:10.1016/j.jlap.2004.03.008.

[14] P.D. Mosses (2019): *CBS IDE*. Available at `https://plancomps.github.io/CBS-beta/docs/F-IDE-2019/CBS.zip`. Language specification project for Spoofax-2.5.7, unreleased prototype.

[15] P.D. Mosses (2019): *Software Meta-language Engineering and CBS*. *J. Comput. Lang.* 50, pp. 39–48, doi:10.1016/j.jvlc.2018.11.003.

[16] P.D. Mosses & F. Vesely (2014): *FunKons: Component-Based Semantics in K*. In: *WRLA 2014*, *LNCS* 8663, Springer, pp. 213–229, doi:10.1007/978-3-319-12904-4_12.

[17] PLanCompS Project (2019): *CBS: A Framework for Component-Based Specification of Programming Languages*. Available at `https://plancomps.github.io/CBS-beta`. Beta release.

[18] G. Rosu (2017): *K: A Semantic Framework for Programming Languages and Formal Analysis Tools*. In: *Dependable Software Systems Engineering*, IOS Press, pp. 186–206, doi:10.3233/978-1-61499-810-5-186.

[19] L.E. de Souza Amorim (2019): *Declarative Syntax Definition for Modern Language Workbenches*. Ph.D. thesis, Delft University of Technology, doi:10.4233/uuid:43d7992a-7077-47ba-b38f-113f5011d07f.

[20] L.E. de Souza Amorim, E. Visser & G. Wachsmuth (2014): *Developing SDF3*. In: *Parsing@SLE 2014*. Available at `https://www.sleconf.org/2014/parsing-slides/2-sdf3-slides.pdf`.

[21] E. Visser (1997): *Syntax Definition for Language Prototyping*. Ph.D. thesis, University of Amsterdam.

[22] E. Visser et al. (2014): *A Language Designer's Workbench: A One-Stop-Shop for Implementation and Verification of Language Designs*. In: *Proc. Onward! 2014*, ACM, pp. 95–111, doi:10.1145/2661136.2661149.

An Integrated Development Environment for the Prototype Verification System

Paolo Masci
National Institute of Aerospace
Hampton, VA, USA
paolo.masci@nianet.org

César A. Muñoz
NASA
Hampton, VA, USA
cesar.a.munoz@nasa.gov

The steep learning curve of formal technologies is a well-known barrier to the adoption of formal verification tools in industry. This paper presents VSCode-PVS, a modern integrated development environment for the Prototype Verification System (PVS). This new environment integrates the editing and proof management functionalities of PVS in Visual Studio Code, a popular code editor widely used by software developers. VSCode-PVS provides functionalities that developers expect to find in modern verification tools, but are not available in the standard Emacs front-end of PVS, such as auto-completion, point-and-click navigation of definitions, live diagnostics for errors, and literate programming. The main features and architecture of the environment are presented, along with a comparison with other similar tools.

1 Introduction

Early detection of design anomalies and increased confidence that the system will operate as intended are some of the benefits of the use of formal verification technologies in industry. However, outside safety-critical domains such as avionics, the use of verification tools is still rather limited. One of the reasons is the steep learning curve of verification technologies, which creates an initial cost that is often deemed excessive with respect to the long term benefits.

The work presented in this paper aims to reduce the learning curve of the Prototype Verification System (PVS) [10], a verification tool for formal modeling and analysis of system designs. The PVS modeling language is based on higher-order logic. It supports basic types (`boolean`, `int`, `real`, etc), as well as datatypes such as `string`, `set`, `list`. The proof engine is based on Gentzen's sequent calculus, and supports the use of proof strategies for automated analysis. The verification system provides an evaluation environment, called PVSio [8], for animation of executable specifications.

PVS is a powerful analysis tool with a long history of success stories in a range of different domains [9]. PVS is widely used at NASA Langley Research Center for the analysis of algorithms and protocols for avionics systems[1]. Research groups have also applied PVS to the analysis of human-machine interfaces in medical systems [6] and for co-simulation of Cyber-Physical Systems [12] and semi-autonomous systems [13].

Becoming fluent with PVS, however, usually requires several weeks. Common difficulties faced by developers are often rooted in the PVS front-end:

- **Conceptual gap.** The PVS front-end is based on the Emacs editor. Emacs does not provide separate visual components for editing files, executing commands, and browsing the file system. Rather, it provides *buffers*, abstract entities that can be used to interact with any resource. When

[1] `https://shemesh.larc.nasa.gov/fm/fm-main-research.html`

a buffer is linked to a text file, e.g., a PVS specification, Emacs acts like a code editor. When the buffer is linked to an interactive process, e.g., the PVS theorem prover, Emacs provides a command-line interface for sending commands to the process. Developers need to learn the set of commands for operating with buffers, as well as to recognize the buffers — the visual appearance of different buffers is identical to the untrained eye.

- **Knowledge gap.** The PVS Emacs interface favors the use of the command line integrated in Emacs. Developers need to learn several commands and keyboard shortcuts to be fluent when editing, parsing, and analyzing PVS specifications. Means for efficient navigation of the libraries are not provided in PVS Emacs. This can significantly slow down the development of PVS specifications and proofs. For example, the NASA PVS Library [2] includes over 100,000 theorems and definitions useful for modeling and analysis of different aspects of safety-critical systems. Without appropriate tool support, finding anything within these libraries is a prohibitive task.

The main contribution of this work is VSCode-PVS, a new integrated development environment designed to reduce substantially the conceptual and knowledge gaps faced by PVS users. A preliminary but fully functional version of the new environment is presented, along with a comparison with other environments. The environment is publicly available at GitHub[3] under NASA's Open Source Agreement.

The rest of this paper is organized as follows. Section 2 provides background information on the standard PVS front-end, and on Visual Studio Code. Sections 3 and 4 introduce the architecture and main features of the environment. Section 5 identifies a set of metrics that are general across the front-end of different modeling and analysis tools, and uses these metrics to compare VSCode-PVS to other environments. Section 6 presents related work. Finally, Section 7 concludes the paper.

2 Background

This section provides background information on PVS Emacs, the standard front-end of PVS, and Visual Studio Code, the baseline technology used for the development of VSCode-PVS.

2.1 PVS Emacs

PVS Emacs is a text-based environment built on the Emacs editor to support user interaction with PVS. Commands necessary to interact with the PVS system are invoked by typing the command name and its arguments, if any, in the Emacs minibuffer. To enter a command in the minibuffer, users need to pre-fix the command with the key sequence `M-x`, where `M` is usually the Alt key. An example is `M-x typecheck`, which executes the PVS type-checker command on the PVS file opened in the Emacs editor. Over 100 commands are provided (see [11]). Additionally, an extensive list of keyboard shortcuts is implemented to speed up command entry. For example, type-checking can be executed by typing `M-x tc` in the minibuffer, or by performing the key sequence `C-c C-t`, where `C-` is the Ctrl key.

2.2 Visual Studio Code

Visual Studio Code is a cross-platform open-source code editor created in 2015 by Microsoft. The editor provides a rich graphical user interface that integrates the essential components typically used by programmers: a source code editor that supports auto-completion, hovers, embedded mini-editors, and

[2] `https://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library`.

[3] `https://github.com/nasa/vscode-pvs`.

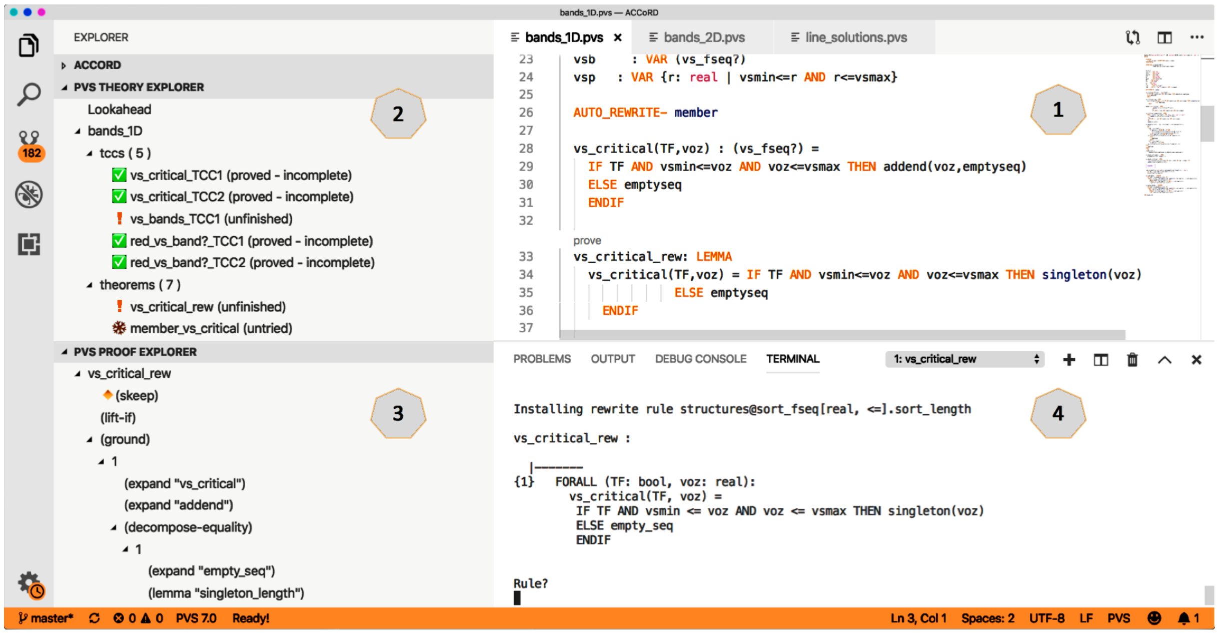
contextual menus; an integrated graphical debugger, which allows the user to set break-points and perform step-by-step execution of source code; a tree-based view for browsing files in the file system; an integrated terminal for executing commands; integration with source code management tools (e.g., Git). The behavior of all these components can be extended or re-programmed to provide support for a specific programming language. The community has already created extensions for over 80 programming languages, including C++, Java, JavaScript, and Python. Besides programming languages, there is also a growing interest in integrating verification tools in Visual Studio Code. Examples include Dafny[4] and Lean[5]. Some of these extensions will be discussed further below, in Section 5.

3 VSCode-PVS

VSCode-PVS is a new integrated development environment for creating, evaluating and verifying PVS specifications. The environment, shown in Figure 1, redefines the way developers interact with PVS, and better aligns the PVS front-end to the functionalities provided by development environments used by software developers. The main features provided by the environment are as follows:

- **Syntax highlighting.** PVS keywords and library functions are automatically highlighted.
- **Autocompletion and code snippets**. Tooltips suggesting function names and language keywords are automatically presented when the user types a symbol in the editor. Code snippets are provided for frequent modeling blocks, e.g., `if-then-else`.
- **Hover information for symbol definitions**. Hover boxes providing information about identifiers are automatically displayed when the developer places the cursor over an identifier.
- **Jump-to declaration**. Navigation of symbol declarations can be performed with simple point-and-click actions: the user places the cursor over the name of an identifier, and a click on the name of the identifier while holding the Ctrl key down opens a window with the location where the identifier is declared.
- **Live diagnostics**. Parsing is automatically performed in the background, and errors are reported in-line in the editor. Problematic expressions are underlined with red wavy lines. Tooltips presenting the error details are shown when the user places the cursor over the wavy lines.
- **In-line actionable commands**. Actionable commands are available for PVS theorems. They are rendered in-line in the editor, above the name of the theorem, and can be used to start a new prover session for the theorem with a simple click action.
- **Overview of PVS theories**. The overall structure of a set of PVS theories is rendered using an interactive tree-based view. It shows the set of PVS theories in the active workspace, as well as the name and status (proved, unfinished, etc.) of the theorems defined in each theory. Point-and-click actions can be used to jump to theory definitions and type-check the theories.
- **Interactive proof tree visualizer and editor**. An interactive tree-based view shows the proof associated with a theorem. Point-and-click actions are provided for step-by-step execution of proof commands. Functionalities for editing the proof are currently under development.
- **Integrated PVS and PVSio Command Line Interfaces**. Integrated command line interfaces allow interaction with the theorem prover and the PVSio evaluator. Auto-completion is provided for prover commands, as well as access to the commands history.

[4]`https://github.com/DafnyVSCode/Dafny-VSCode`
[5]`https://github.com/leanprover/vscode-lean`



```
Lookahead
bands_1D
  tccs ( 5 )
    vs_critical_TCC1 (proved - incomplete)
    vs_critical_TCC2 (proved - incomplete)
    vs_bands_TCC1 (unfinished)
    red_vs_band?_TCC1 (proved - incomplete)
    red_vs_band?_TCC2 (proved - incomplete)
  theorems ( 7 )
    vs_critical_rew (unfinished)
    member_vs_critical (untried)
```

```
vs_critical_rew
  (skeep)
  (lift-if)
  (ground)
    1
      (expand "vs_critical")
      (expand "addend")
      (decompose-equality)
        1
          (expand "empty_seq")
          (lemma "singleton_length")
```

```
vsb     : VAR (vs_fseq?)
vsp    : VAR {r: real | vsmin<=r AND r<=vsmax}

AUTO_REWRITE- member

vs_critical(TF,voz) : (vs_fseq?) =
  IF TF AND vsmin<=voz AND voz<=vsmax THEN addend(voz,emptyseq)
  ELSE emptyseq
  ENDIF

prove
vs_critical_rew: LEMMA
  vs_critical(TF,voz) = IF TF AND vsmin<=voz AND voz<=vsmax THEN singleton(voz)
                        ELSE emptyseq
    ENDIF

```

```
Installing rewrite rule structures@sort_fseq[real, <=].sort_length

vs_critical_rew :

   |-------
{1}   FORALL (TF: bool, voz: real):
        vs_critical(TF, voz) =
         IF TF AND vsmin <= voz AND voz <= vsmax THEN singleton(voz)
         ELSE empty_seq
         ENDIF


Rule?
█
```

Figure 1: VSCode-PVS: (1) Main Editor; (2) Theory Explorer; (3) Proof Explorer; (4) Integrated PVS Command Line Interface (CLI).

3.1 Architecture

The overall architecture of VSCode-PVS is depicted in Figure 2. It builds on the Language Server Protocol (LSP), a tool-independent communication protocol for exchanging data and events between two architectural elements: an editor front-end and a language server back-end.

The *editor front-end* is responsible for rendering visual feedback to the user, and transforms user interactions with the editor into corresponding LSP events to be dispatched to the language server. The *language server* defines the functions necessary to support the syntax and semantics of the language (e.g., in the case of PVS, parsing, typechecking, etc.) and continuously listens to LSP events. An example LSP event is `onHover`. This event is triggered by the editor front-end when the user places the cursor over an identifier in the text document. This event is sent to the language server, along with information on the path of the text document and the location of the cursor in the document. The language server acts upon this event, in this case, by sending the identifier's definition back to the editor. The editor, in turn, displays the server response to the user as a hover box. The language server can also generate events. For example, an event `sendDiagnostics`, is used by the server to publish diagnostics information (e.g., parsing error).

The LSP protocol is extensible. It builds on Remote Procedure Calls (RPCs) and the JavaScript Object Notation (JSON) format. It allows the definition of new event types to accommodate language-specific features. In the case of PVS, this feature is used to implement commands necessary for interactive analysis of PVS specifications, e.g., type-checking, discharging proof obligations, proving theorems, etc. The LSP-based architecture has been chosen for the implementation of VSCode-PVS because it promotes reuse of modules and facilitates sustainability of the overall development effort — LSP-compliant editors can be connected to the Language Server back-end. All major code editors support the LSP, which makes it relatively simple to connect the PVS language server to a different editor front-end. Fea-

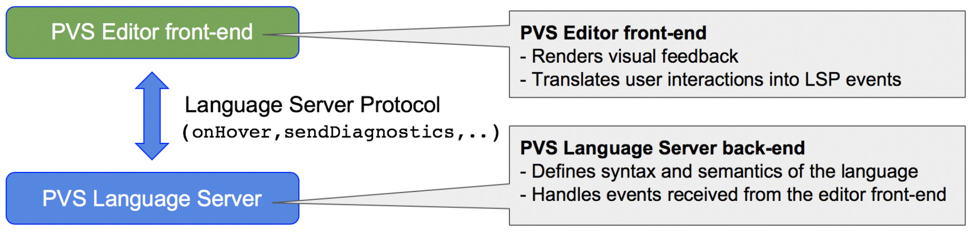

PVS Editor front-end
Language Server Protocol
(onHover,sendDiagnostics,..)
PVS Language Server
PVS Editor front-end
- Renders visual feedback
- Translates user interactions into LSP events
PVS Language Server back-end
- Defines syntax and semantics of the language
- Handles events received from the editor front-end

Figure 2: Overall architecture of VSCode-PVS.

tures implemented in the language server for standard LSP events become automatically available in the connected editor.

3.2 The Editor Front-End

The PVS Editor front-end builds on Visual Studio Code[6], an open-source code editor widely used by software developers. The main software modules of the front-end are shown in Figure 3. They are illustrated in the following.

Editor Extensions. These modules are used to tailor the functionalities of the textual editor provided by Visual Studio Code to the PVS language. Functionalities implemented in these modules include: decorators necessary for syntax highlighting; code snippets for rapid creation of PVS code blocks; code folding; key bindings and contextual menus for PVS commands.

Explorer Extensions. These modules customize Explorer View, a graphical tree-based view integrated in Visual Studio Code. A new module, *Theory Explorer*, introduces support for click-and-point navigation of PVS specifications. A second module, *Proof Explorer*, aims to support interactive visualization and editing of proofs using click-and-point operations.

Integrated PVS Terminals. These modules seamlessly link the integrated terminal of Visual Studio Code to the interactive read-eval-print loops of the PVS theorem prover and the PVSio [8] evaluator. That is, in the *prover terminal*, developers can type proof commands for the theorem prover, and watch the proof state returned by the prover directly in the terminal. Similarly, in the *evaluator terminal*, developers can type ground expressions and thus execute fragments of a PVS specification.

VSCode APIs. This is a library provided by Visual Studio Code for extending and customizing the functionalities of the editor. It includes communication primitives necessary to support the LSP.

3.3 The Language Server

The PVS Language Server includes the following modules (see Figure 4):

LSP Service Providers. These modules handle LSP events received by the language server. They reconcile the APIs of PVS with the logic of the LSP. For example, *Hover Provider*, which is activated when an `onHover` event is received, uses the APIs of PVS to gather the information to be shown in hover boxes, and then sends the response to the editor front-end using the LSP format. *Definition Provider* defines the logic necessary to support functions such as go-to definition and peek definition, which are used in the editor front-end for point-and-click navigation of PVS specifications. *CodeLens Provider* defines the logic behind actionable commands embedded in-line in the PVS specification — this is used

[6]`https://code.visualstudio.com`

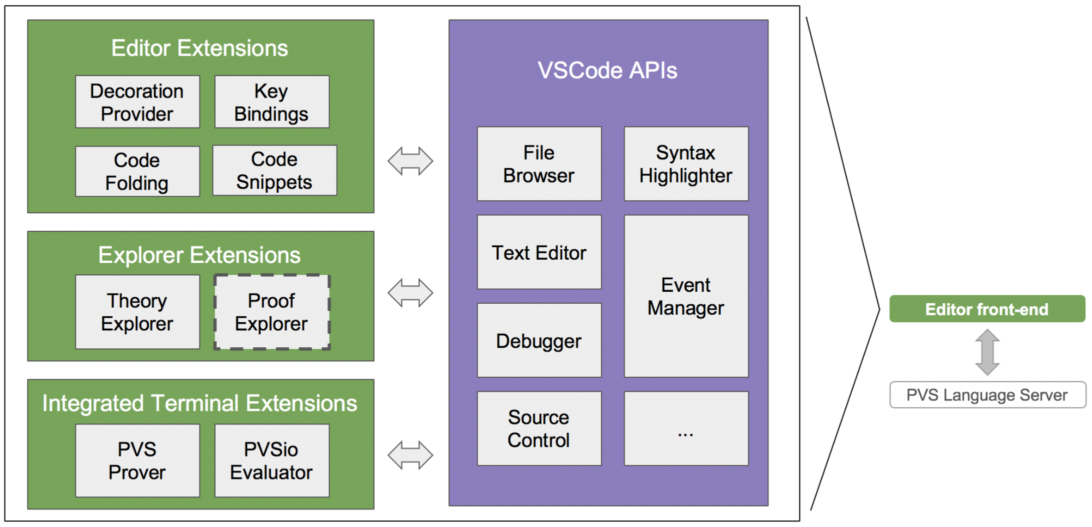


Figure 3: Inner architecture of the PVS editor front-end. Modules are represented as boxes. Communication between modules is indicated with arrows. Dashed line indicates under development.

to introduce in-line actionable *prove* commands at the location of theorem definitions. *PVS Commands Provider* provides support for language-specific commands used for analysis of PVS specifications, e.g., type-check, show proof obligations, prove theorem. *Diagnostics Provider* is a background process that continuously sends diagnostics information to the editor front-end to report syntax / type-checking errors.

LSP Connection Manager. This is a routing module for managing the exchange of events and data with the client front-end. The module listens for client connections, receives LSP events from connected clients, and dispatches the events to the appropriate service provider.

PVS Process Workers. These modules embed the functionalities of the PVS verification system in the language server. Each worker executes a PVS instance in a self-contained execution environment. A pool of workers supports parallel execution of multiple PVS instances — this is used for running background services like parsing and type-checking. Furthermore, in contrast to PVS Emacs, VSCode-PVS supports running simultaneously proofs of different formulas.

3.4 Implementation

VSCode-PVS is implemented in TypeScript[7], an extension of JavaScript that supports type annotations. Typescript programs can be statically checked for type correctness. A transpiler translates TypeScript code into plain JavaScript, which allows the execution of TypeScript programs in standard JavaScript engines. The editor front-end builds on the APIs of Visual Studio Code. The language server builds on NodeJS[8], a JavaScript environment that provides libraries necessary for creating web-services, including functions for spawning processes and performing operations on file systems. Process workers use the native Lisp interface of PVS to exchange commands and data with the PVS system. TSLint[9], a static analyzer for TypeScript, is routinely used for checking compliance with established coding conventions.

[7] `https://www.typescriptlang.org`
[8] `https://nodejs.org`
[9] `https://palantir.github.io/tslint`

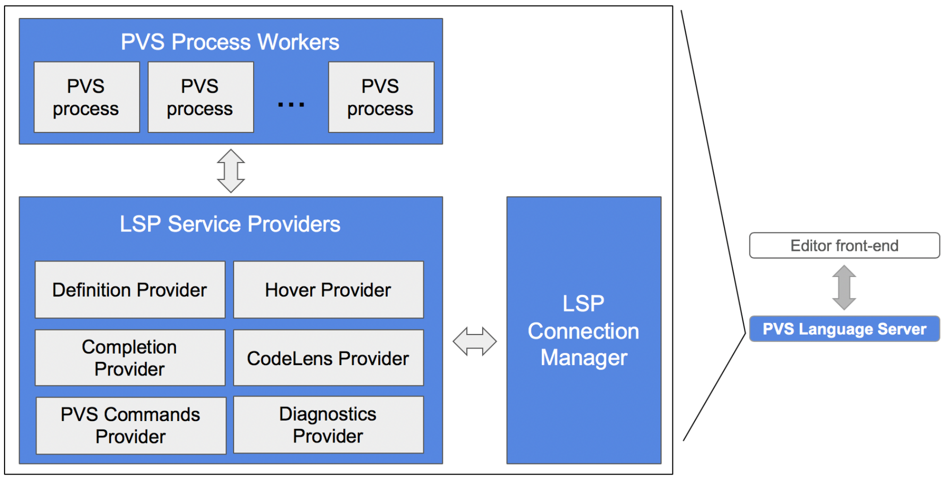


Figure 4: Inner architecture of the PVS Language Server.

Jasmine[10] is used for testing the APIs of the developed modules.

The implementation effort to date amounts to 7K LoC (3K LoC for the editor front-end, and 4K LoC for the language server). Most of the code developed for the editor front-end is associated with the interactive tree-based views for visualizing theories and proofs. The server back-end required the development of a Lisp interface to dialogue with PVS, as well as additional logic for LSP events that are not directly supported by PVS, e.g., auto-completion for identifiers, hover information, live diagnostics.

4 Example use of VSCode-PVS

This section showcases the main features of VSCode-PVS for two representative tasks typically carried out by PVS users. Tasks are provided in the form of short descriptions presenting the overall *goal* of the task and the *context* within which the task is carried out. A comparison with the standard PVS Emacs front-end is included in each task to better appreciate the improvements introduced by VSCode-PVS.

4.1 Task 1: Navigation of symbol declarations

Goal: Inspect function and type declarations from imported PVS files.

Context: When developing a new PVS specification or a new proof, users typically need to navigate symbol declarations imported from other PVS theories. This is necessary, e.g., to inspect the structure of complex datatypes defined in the PVS libraries.

Workflow with VSCode-PVS. The standard workflow for inspecting a symbol declaration involves using the hover functionality (see Figure 5b). The user can place the cursor over the symbol definition, and a tooltip will be automatically shown. The tooltip includes three main elements: a brief description of the symbol (e.g., built-in type); a clickable link for jumping to the location of the declaration; a preview of the symbol declaration. This standard workflow can always be adopted when a PVS specification type-checks correctly. An alternative workflow is also available for theories that are still not type-checked, e.g., because the user has not finished yet typing the content of the specification. In these cases, the

[10]`https://jasmine.github.io`

resolution of symbol declarations can still be performed, but can be less accurate when symbol names are overloaded. That is, when the symbol to be resolved is overloaded and the theory is not type-checked, an array of candidate declarations is presented. In these cases, the *peek declaration* functionality is employed (see Figure 5c). It opens a mini-editor and a file browser in the current editor window that can be used to inspect the candidate declarations.

Workflow with Emacs. Symbol declarations can be inspected with the command `show-declaration`. The command takes the name of the symbol as argument. The current location of the cursor can be used to auto-complete the symbol name. The command opens a new Emacs buffer with a preview of the declaration. The command `goto-declaration` can then be used to jump to the location of the declaration. The command takes the name of the symbol as argument. When the theory is not type-checked, a command `find-declaration` can be used to inspect a list of possible candidates. The jump-to functionality is however not available in this case, and the user needs to manually open the file (command `C-c C-f` followed by the filename) and scroll the text to the position of the declaration.

4.2 Task 2: Proving a theorem

Goal: Verify that a PVS specification satisfies given formal (mathematical) properties.

Context: For complex systems, it is important to analyze a system design before the actual system is built. This helps developers gain confidence that the system design complies to given specifications, and identify and fix potential design issues early in the development process, when the cost of design changes is still relatively low. In safety-critical application domains, such as avionics and healthcare, such design analysis is usually mandated by regulatory frameworks. Proving mathematical theorems that capture properties of the intended characteristics and functionalities of the system is the core approach used in formal methods. It provides means to check properties of a system design for all possible inputs in all possible system states.

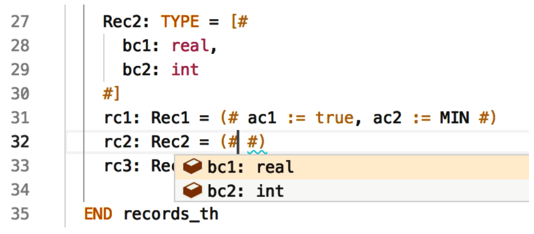
Workflow with VSCode-PVS. In-line actionable commands are provided next to each theorem. For example, in Figure 1, an actionable command `prove` is shown in the main editor window, above the theorem name at line 33). A click on the actionable command, triggers type-checking, and launches a new prover sessions in the integrated PVS Command Line Interface (see Figure 1, lower-right panel). Proof commands can be typed in the command line interface. The current proof is shown on the side, using an interactive tree-based view, called Proof Explorer (see Figure 1, lower-left panel). Nodes in the tree view can be collapsed/expanded to facilitate inspection of large proofs. Editing of the proof tree from Proof Explorer is under development.

Workflow with Emacs. The user needs to type-check the file (`M-x tc`) and then start a theorem prover session with the command `M-x prove`. The command opens a new Emacs buffer that can be used to interact with the theorem prover. Proof commands are typed in this new buffer. A command `M-x x-show-current-proof` can be used to start open a window showing the proof tree (see Figure 6d). The proof tree cannot be edited, and it does not allow collapsing/expanding of proof branches.

5 Comparing VSCode-PVS to Other Analysis Tools

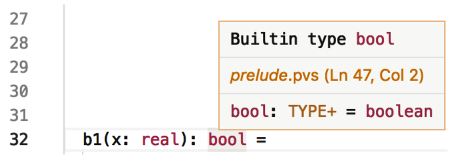
This section presents a comparison between VSCode-PVS and other similar environments. The following verification environments are considered:

- PVS Emacs, the standard front-end of PVS;



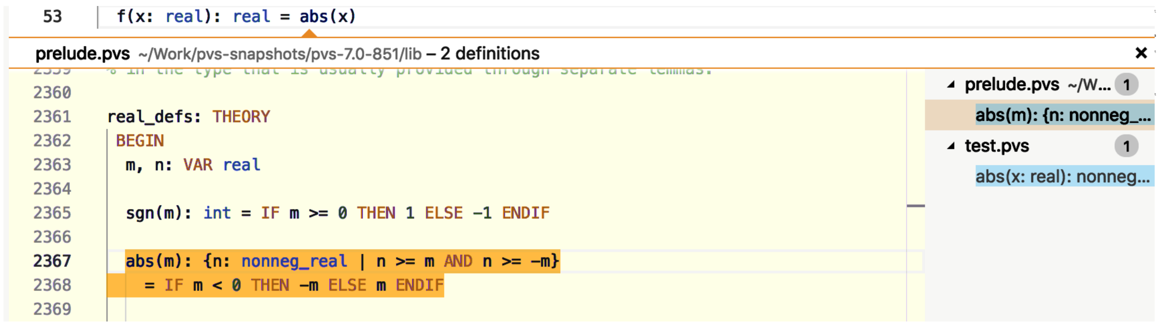
```
  Rec2: TYPE = [#
    bc1: real,
    bc2: int
  #]
  rc1: Rec1 = (# ac1 := true, ac2 := MIN #)
  rc2: Rec2 = (# #)
  rc3: Re  bc1: real
           bc2: int
END records_th
```

(a) Autocompletion.



```
Builtin type bool
prelude.pvs (Ln 47, Col 2)
bool: TYPE+ = boolean
b1(x: real): bool =
```

(b) Hover.



```
f(x: real): real = abs(x)
```

prelude.pvs ~/Work/pvs-snapshots/pvs-7.0-851/lib – 2 definitions

```
  % in the type that is usually provided through separate lemmas.

real_defs: THEORY
 BEGIN
  m, n: VAR real

  sgn(m): int = IF m >= 0 THEN 1 ELSE -1 ENDIF

  abs(m): {n: nonneg_real | n >= m AND n >= -m}
    = IF m < 0 THEN -m ELSE m ENDIF
```

(c) Peek definitions.

Figure 5: Example functionalities of VSCode-PVS.

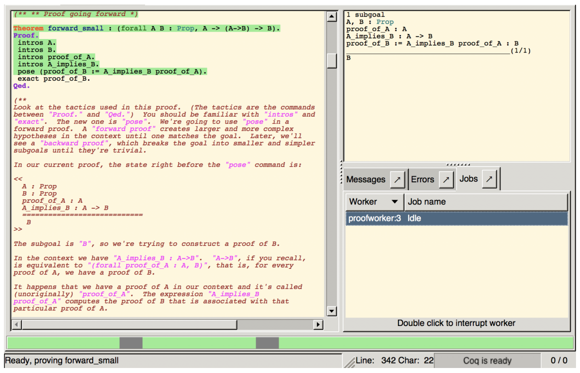
- Isabelle/jEdit [18, 19], the standard front-end of the Isabelle/HOL theorem proving system;
- SublimeHOL[11], a front-end to the HOL4 theorem prover;
- CoqIDE, the default front-end of the Coq theorem proving systems;
- Proof General [1], a generic front-end for theorem provers;
- Lean [7], a new open-source theorem proving system from Microsoft;
- IntelliJ-Arend[12], a new proof assistant under development at JetBrains research;
- KeYmaera-X [2], an interactive theorem prover for hybrid systems.

5.1 Metrics

A set of metrics has been defined to guide the comparison. The set of metrics is not exhaustive. Rather, they capture core functionalities necessary to support common modeling and analysis tasks.

1. **Autocompletion**: ability to suggest keywords and identifiers while typing;
2. **Hover information**: ability to show informative pop-ups for keywords and identifiers;
3. **Jump-to-definition**: ability to open a file at the location of an identifier's definition;
4. **Refactoring**: ability to rename identifiers;
5. **Live diagnostics**: ability to show diagnostics information while typing;
6. **Animation**: ability to evaluate executable specifications;
7. **Proof visualizer**: ability to visualize a proof tree.

[11]`https://github.com/JamesShaker/SublimeHOL`
[12]`https://github.com/JetBrains/intellij-arend`



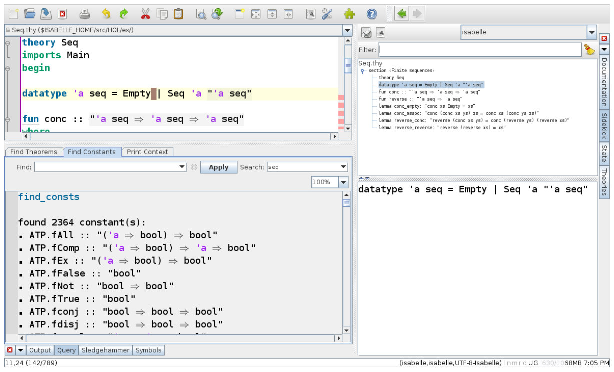
```
(** ** Proof going forward *)

Theorem forward_small : (forall A B : Prop, A -> (A->B) -> B).
Proof.
 intros A.
 intros B.
 intros proof_of_A.
 intros A_implies_B.
 pose (proof_of_B := A_implies_B proof_of_A).
 exact proof_of_B.
Qed.

(**
Look at the tactics used in this proof.  (The tactics are the commands
between "Proof." and "Qed.")  You should be familiar with "intros" and
"exact".  The new one is "pose".  We're going to use "pose" in a
forward proof.  A "forward proof" creates larger and more complex
hypotheses in the context until one matches the goal.  Later, we'll
see a "backward proof", which breaks the goal into smaller and simpler
subgoals until they're trivial.

In our current proof, the state right before the "pose" command is:

<<
  A : Prop
  B : Prop
  proof_of_A : A
  A_implies_B : A -> B
  ============================
   B
>>

The subgoal is "B", so we're trying to construct a proof of B.

In the context we have "A_implies_B : A->B".  "A->B", if you recall,
is equivalent to "(forall proof_of_A : A, B)", that is, for every
proof of A, we have a proof of B.

It happens that we have a proof of A in our context and it's called
(unoriginally) "proof_of_A".  The expression "A_implies_B
proof_of_A" computes the proof of B that is associated with that
particular proof of A.
```

```
1 subgoal
A, B : Prop
proof_of_A : A
A_implies_B : A -> B
proof_of_B := A_implies_B proof_of_A : B
______________________________________(1/1)
B
```

```
Worker      Job name
proofworker:3  Idle
```

Messages
Errors
Jobs
Double click to interrupt worker
Ready, proving forward_small
Line: 342 Char: 22
Coq is ready
0 / 0

(a) CoqIDE: editor and side view for proof inspection.



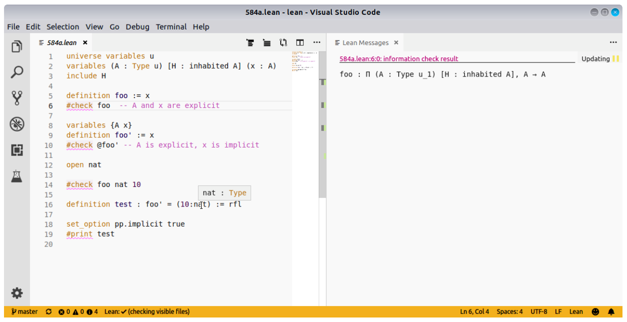
```
theory Seq
imports Main
begin

datatype 'a seq = Empty | Seq 'a "'a seq"

fun conc :: "'a seq ⇒ 'a seq ⇒ 'a seq"
where
```

```
find_consts

found 2364 constant(s):
▪ ATP.fAll :: "('a ⇒ bool) ⇒ bool"
▪ ATP.fComp :: "('a ⇒ bool) ⇒ 'a ⇒ bool"
▪ ATP.fEx :: "('a ⇒ bool) ⇒ bool"
▪ ATP.fFalse :: "bool"
▪ ATP.fNot :: "bool ⇒ bool"
▪ ATP.fTrue :: "bool"
▪ ATP.fconj :: "bool ⇒ bool ⇒ bool"
▪ ATP.fdisj :: "bool ⇒ bool ⇒ bool"
```

```
Seq.thy
 section ‹Finite sequences›
   theory Seq
   datatype 'a seq = Empty | Seq 'a "'a seq"
   fun conc :: "'a seq ⇒ 'a seq ⇒ 'a seq"
   fun reverse :: "'a seq ⇒ 'a seq"
   lemma conc_empty: "conc xs Empty = xs"
   lemma conc_assoc: "conc (conc xs ys) zs = conc xs (conc ys zs)"
   lemma reverse_conc: "reverse (conc xs ys) = conc (reverse ys) (reverse xs)"
   lemma reverse_reverse: "reverse (reverse xs) = xs"
```

```
datatype 'a seq = Empty | Seq 'a "'a seq"
```

(b) Isabelle/jEdit.

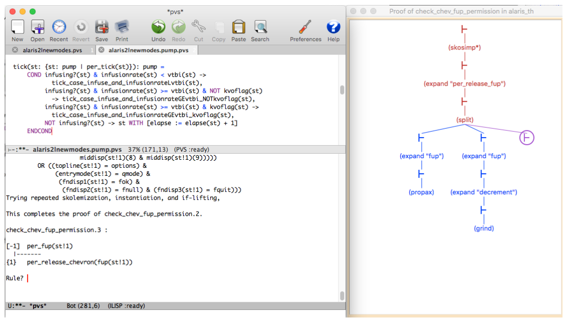


```
universe variables u
variables (A : Type u) [H : inhabited A] (x : A)
include H

definition foo := x
#check foo  -- A and x are explicit

variables {A x}
definition foo' := x
#check @foo' -- A is explicit, x is implicit

open nat

#check foo nat 10

definition test : foo' = (10:nat) := rfl

set_option pp.implicit true
#print test
```

```
584a.lean:6:0: information check result
foo : Π (A : Type u_1) [H : inhabited A], A → A
```

584a.lean - lean - Visual Studio Code
File Edit Selection View Go Debug Terminal Help
584a.lean
Lean Messages
Updating
nat : Type
master
0 0 4 Lean: ✓ (checking visible files)
Ln 6, Col 4 Spaces: 4 UTF-8 LF Lean

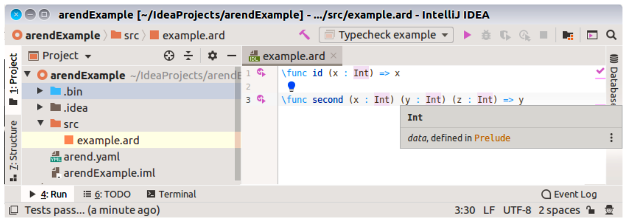
(c) Lean.



```
tick(st: {st: pump | per_tick(st)}): pump =
    COND infusing?(st) & infusionrate(st) < vtbi(st) ->
           tick_case_infuse_and_infusionrateLvtbi(st),
         infusing?(st) & infusionrate(st) >= vtbi(st) & NOT kvoflag(st)
           -> tick_case_infuse_and_infusionrateGEvtbi_NOTkvoflag(st),
         infusing?(st) & infusionrate(st) >= vtbi(st) & kvoflag(st) ->
           tick_case_infuse_and_infusionrateGEvtbi_kvoflag(st),
         NOT infusing?(st) -> st WITH [elapse := elapse(st) + 1]
    ENDCOND
```

```
-:**-  alaris2newmodes.pump.pvs   37% (171,13)   (PVS :ready)
                  middisp(st!1)(8) & middisp(st!1)(9)))))
         OR ((topline(st!1) = options) &
              (entrymode(st!1) = qmode) &
               (fndisp1(st!1) = fok) &
                (fndisp2(st!1) = fnull) & (fndisp3(st!1) = fquit)))
Trying repeated skolemization, instantiation, and if-lifting,

This completes the proof of check_chev_fup_permission.2.

check_chev_fup_permission.3 :

[-1]  per_fup(st!1)
  |-------
{1}   per_release_chevron(fup(st!1))

Rule?
U:**-  *pvs*        Bot (281,6)    (ILISP :ready)
```

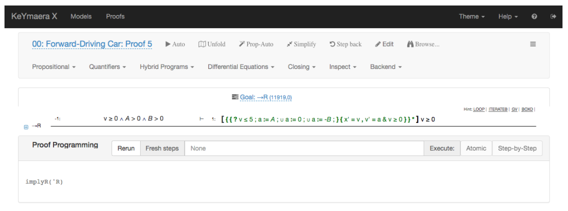
(d) PVS Emacs: editor and proof window.



```
\func id (x : Int) => x

\func second (x : Int) (y : Int) (z : Int) => y
```

```
Int
data, defined in Prelude
```

(e) IntelliJ-Arend.



$$-1:\quad v \geq 0 \land A > 0 \land B > 0 \quad \vdash \quad 1:\ [\{\{?\,v \leq 5;\ a := A;\ \cup\ a := 0;\ \cup\ a := -B;\}\{x' = v, v' = a \,\&\, v \geq 0\}\}^*]\, v \geq 0$$

```
implyR('R)
```

(f) KeYmaera-X.

Figure 6: Screenshots of other analysis tools.

5.2 Assessment

VSCode-PVS. Autocompletion is provided for language keywords, as well as for types and functions defined in the standard PVS library (the `prelude`). Context-sensitive autocompletion is supported for record types: items suggested by the editor range over record accessors (see Figure 5a). Hover information is provided for identifiers. It shows a preview of the definition of the identifier, as well as an hyperlink that can be used to jump-to the location of the definition (see Figure 5b). Live diagnostics for syntax errors are automatically provided after a period of inactivity in the editor — syntax errors are underlined with red wavy lines, and a tooltip with details on the error is provided when placing the cursor at the error position. Integrated terminals can be used to animate specifications and prove theorems. An interactive tree-based view allows for the visualization of proof trees and proof commands step-by-step.

PVS Emacs. The environment, shown in Figure 6d, provides syntax highlighting and a basic form of jump-to-definition through commands and key bindings. A basic form of autocompletion is provided for proof commands. The logic for hover information is not implemented. Diagnostics are obtained on-demand, when the developer decides to parse or type-check the specification. Animation is provided through an interactive read-eval-print loop. Proofs can be visualized in a tree-based view. Nodes in the tree represent either proof sequents or proof commands. An interactive text-based view is used to show proof tactics and allows the user to perform step-by-step execution of proof commands.

CoqIDE. The environment is shown in Figure 6a. It provides syntax highlighting. Autocompletion, hover information, and jump-to-definition are not provided. Diagnostics are provided on-demand, when the developer decides to attempt verification. A linear workflow is enforced: verification always starts from the beginning of the file, and a marker is advanced in the editor to indicate what has been verified. Everything above the marker is locked and cannot be modified. A side panel shows the proof state, including current goal, tactic state, and error messages. Point-and-click operations allow step-by-step execution of proof commands. The development of an alternative front-end, vscoq[13], based on Visual Studio Code was attempted in recent years. However, the implementation appear to have stopped at the very early stages and does not provide significant new functionalities with respect to CoqIDE.

Proof General. The environment builds on Emacs, and its visual appearance is similar to PVS Emacs. The main design goal of this environment is to facilitate the execution of proof commands and the navigation of proof strategies. A plug-in based architecture allows the introduction of language-specific extensions. Plug-ins for Coq and Isabelle/HOL are available. They provide syntax highlighting and basic forms of autocompletion. Proof visualization builds on a tree-based view similar to that used in PVS Emacs. Commands are provided for step-by-step execution of proof tactics.

SublimeHOL. The environment builds on the Sublime[14] editor. It provides syntax highlighting for the HOL4 specification language. A basic form of autocompletion is provided for keywords and mathematical symbols. Hover information, jump-to-definition and visualization of proof trees are not provided. Interactive panels allow to exchange commands with HOL4 and edit/inspect the proof state.

Isabelle/jEdit. The environment, shown in Figure 6b builds on jEdit[15]. Context-sensitive autocompletion is provided based on the syntax of the language, as well as on name-space information provided by the prover engine. A dictionary-based spell-checker is used to suggest completion items for comments and other sections in the specification that contain sentences in natural language. Hover information shows the type of language symbols. Hover boxes can be detached from the current editor and turned

[13]`https://github.com/siegebell/vscoq`

[14]`https://www.sublimetext.com`

[15]`http://www.jedit.org/`

into separate windows to facilitate navigation of file content. Live diagnostics are provided for errors in the form of red wavy lines, along with hover information at the error location. A panel shows proof information and allows to perform step-by-step evaluation of a proof method. A query panel enables the filtering of information displayed for the proof state. A basic form of model animation can be achieved through Isabelle's counter-example finding functionality.

Lean. The environment is shown in Figure 6c. It builds on Visual Studio Code. Because of this, its overall visual appearance is similar to VSCode-PVS. Context-sensitive autocompletion is provided for language symbols. A reasoning engine is always active in the background and tries to autocomplete expressions based on context information, in a manner similar to the *hole* functionality provided in Haskell. Hover information shows the definition of identifiers. Navigation of definitions is supported by the jump-to functionality and mini-editors rendered in line in the current editor. Live diagnostics are provided for errors, in the usual form of red wavy lines at the location of the error, and a tooltip with information about the error. Proof commands are embedded in the specification and can be activated with point-and-click operations. Similarly to Coq and jEdit, the proof state is shown in a side panel.

IntelliJ-Arend. The environment builds on IntelliJ IDEA[16]. Autocompletion is provided for keywords and identifiers. An *auto import* function is available that automatically imports library modules. Hover information is used to present a description in natural language of identifiers, as well as a link to the definition of the identifier (see Figure 6e). Refactoring allows to rename identifiers, and move definitions across modules. Live diagnostics are presented for syntax errors. Additional diagnostics can be obtained on-demand by type-checking the specification. Animation and proof visualization are not available.

KeYmaera-X. The environment is the successor of the KeYmaera IDE [16]. Web-based technologies are used to implement the front-end (see Figure 6f). The main focus is on proof development. Only basic functionalities are provided to support modeling tasks: syntax highlighting is provided only for language keywords; hover information, jump-to definition and refactoring are not available. The prover interface renders proofs in sequent form, with horizontal lines as in the Gentzen-style layout. An extensive set of menus provides access to all proof commands. Heuristics are used to suggest proof tactics.

5.3 Results

An overview of the comparison is in Table 1. It can be seen that VSCode-PVS already provides several features that other similar environment are still missing. Most of the environments are mainly designed to provide an interface for exchanging proof commands with the theorem prover. Few environments provide adequate support for modeling activities. For example, almost all environments currently lack *refactoring*, and developers need to rely on search-and-replace functionalities of the editor when renaming identifiers. However, this solution is not robust, as careful inspection is necessary for overloaded identifiers. Two other important features commonly used during modeling activities are also missing in most of the environments: *animation* of executable fragments and *live diagnostics*. Animation provides a means to developers to test a specification, e.g., to check whether it correctly captures what the developer wants to model. This functionality can be especially useful to software engineers that approach formal verification, as it is congruent with the testing methods they routinely use for software. It also provides a form a lightweight formal verification — properties can be checked for specific execution traces prior to running the full formal proof. Live diagnostics promote immediate identification of specification errors. This may facilitate understanding and resolution of errors, as a developer's focus of attention is already at the location of the error.

[16]`https://www.jetbrains.com/idea/`

	Version	Base editor	Autocompletion	Hover information	Jump-to-definition	Refactoring	Live diagnostics	Animation	Proof visualizer
VSCode-PVS	1.0.12	VSCode	●	●	●	○	●	●	●
PVS-Emacs	6.0	Emacs	◒	⊗	◒	○	⊗	●	●
CoqIDE	8.9.0	N/A	⊗	⊗	⊗	⊗	⊗	⊗	◒
SublimeHOL	2018	Sublime	◒	⊗	⊗	⊗	⊗	⊗	◒
Proof General	4.5	Emacs, Eclipse	◒	⊗	⊗	⊗	⊗	⊗	◒
Isabelle/jEdit	2019	jEdit	●	●	●	○	●	◒	●
Lean	0.14.1	VSCode	●	●	●	⊗	●	⊗	●
IntelliJ-Arend	1.0.0	IntelliJ IDEA	●	●	●	●	●	⊗	⊗
KeYmaeraX	4.6.3	N/A	⊗	⊗	⊗	⊗	⊗	⊗	●

Table 1: Overview of the comparison results. The following symbols summarize the characteristics of a feature: fully implemented (●); basic implementation (◒); planned feature (○); not available (⊗).

6 Related Work

VSCode-PVS aims to align the functionalities of the PVS front-end to those of program analyzers such as Dafny [3], or reasoning engines like Imandra [14]. The front-end of these tools provides all functionalities typically available in modern IDEs for programming languages, including context-aware help and an integrated debugger. Verification is carried out in the background, by continuously querying a pool of solvers while the user types the code. Design solutions are adopted to keep the interface responsive and provide an overall smooth programming experience to the user.

The verification technology used by VSCode-PVS is not automatic, as in the case of Dafny and Imandra. However, when the PVS analysis targets routine tasks such as discharging proof obligations necessary to prove type correctness, automatic analysis is usually feasible thanks to the powerful proof strategies provided by PVS. This opportunity needs to be exploited, as it would allow the completion of simple but time-consuming activities that developers need to carry out while creating a formal specification. Appropriate mechanisms need to be developed to limit the use of CPU time and memory resources that could make the interface not responsive. The split architecture adopted in VSCode-PVS and the asynchronous nature of the LSP protocol facilitate the implementation of these mechanisms. The possibility of creating an integrated debugger for executable fragments of a PVS specification is also another interesting option that needs to be explored for VSCode-PVS. The Visual Studio Code editor provides already the graphical elements necessary for interacting with the logic of the debugger, including breakpoints, an interactive panel with the usual run/step-into/step-over commands, as well as an interactive view for inspecting the value of variables, call stack, etc. These elements need to be customized for the PVS language, and appropriate hooks need to be implemented in the back-end to provide the logic necessary for debugging.

In [15], a Proof General plugin is developed that introduces syntax highlighting and autocompletion for the Coq specification language. In [17], a generic user interface for theorem proving systems is introduced. The editor front-end builds on jEdit, and a prototypical specification language for declaring formal terms such as theories, terms, and context. This approach proves useful to implementing a generic version of basic front-end features such as autocompletion, abstract syntax display, error highlighting, and tooltips. These and other similar efforts are certainly worth exploring. However, it is unclear if in the long run they will stand against the rapid evolution of editors such as Visual Studio Code and Atom.

7 Conclusion and Future Directions

A new development environment for the PVS verification system has been presented that aims to align the PVS front-end to that of main stream tools used by software developers. A split architecture is adopted, where an editor front-end communicates with a server back-end. The back-end uses process workers to adapt the APIs of PVS to the Language Server Protocol, a de-facto standard communication protocol for code editors and analyzers. The editor front-end builds on the features of Visual Studio Code, a modern open-source code editor.

VSCode-PVS is under active development. The environment is still in its infancy, but it already advances the standard Emacs front-end of PVS in many respects — live diagnostics, context-sensitive auto-completion, point-and-click navigation, interactive tree-based view for proof exploration.

Previous attempts carried out by others to develop a new front-end for PVS had little success. One attempt aimed to integrate PVS in Eclipse. Difficulties were encountered to align the APIs provided by PVS to the functionalities required for Eclipse, and the development was ultimately abandoned. Another attempt involved the development of a Python front-end for PVS, using the wxPython[17] graphic library. A simple interface was created to exchange commands with PVS. Fragments of these implementations can be found in the GitHub repository of PVS[18].

Current work on the VSCode-PVS front-end focuses on the integration with the next release of PVS, which provides a new XMLRPC interface that will improve performance and robustness of the language server. The creation of an integrated debugger is also planned. It will align the functionalities of the PVSio evaluator to those of debuggers used in programming languages. Integration with the PVSio-web [5] prototyping environment is another future direction. PVSio-web enables the creation of interactive prototypes based on formal models. The prototypes resemble the visual appearance of a final system. They can be used to create scenario-based simulations that facilitate engagement between PVS experts and developers that are not familiar with PVS or formal methods (see [4] for application examples and success stories).

Acknowledgement. Work by the first author is supported by NASA's System Wide Safety Project under NASA/NIA Cooperative Agreement NNL09AA00A.

References

[1] David Aspinall (2000): *Proof General: A generic tool for proof development*. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Springer, pp. 38–43, doi:10.1007/3-540-46419-0_3.

[2] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp & André Platzer (2015): *KeYmaera X: An axiomatic tactical theorem prover for hybrid systems*. In: *International Conference on Automated Deduction*, Springer, pp. 527–538, doi:10.1007/978-3-319-21401-6_36.

[17] `https://wxpython.org`

[18] `https://github.com/SRI-CSL/PVS`

[3] K Rustan M Leino (2010): *Dafny: An automatic program verifier for functional correctness*. In: *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, Springer, pp. 348–370, doi:10.1007/978-3-642-17511-4_20.

[4] Paolo Masci (2019 (to appear)): *Experiences on Streamlining Formal Methods Tools*. In: *International Workshop on Practical Formal Verification for Software Dependability (AFFORD'19)*.

[5] Paolo Masci, Patrick Oladimeji, Yi Zhang, Paul Jones, Paul Curzon & Harold Thimbleby (2015): *PVSio-web 2.0: Joining PVS to HCI*. In: *International Conference on Computer Aided Verification*, Springer, pp. 470–478, doi:10.1007/978-3-319-21690-4_30.

[6] Paolo Masci, Yi Zhang, Paul Jones, Paul Curzon & Harold Thimbleby (2014): *Formal verification of medical device user interfaces using PVS*. In: *International Conference on Fundamental Approaches to Software Engineering*, Springer, pp. 200–214, doi:10.1007/978-3-642-54804-8_14.

[7] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn & Jakob von Raumer (2015): *The Lean theorem prover (System Description)*. In: *International Conference on Automated Deduction*, Springer, pp. 378–388, doi:10.1007/978-3-319-21401-6_26.

[8] César A Muñoz (2003): *Rapid prototyping in PVS*. Technical Report, NASA/CR-2003-212418, NIA Report No. 2003-03. Available at `https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/20040046914.pdf`.

[9] César A. Muñoz & Ramiro A. Demasi (2012): *Advanced Theorem Proving Techniques in PVS and Applications*. In Bertrand Meyer & Martin Nordio, editors: *Tools for Practical Software Verification: LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 96–132, doi:10.1007/978-3-642-35746-6_4.

[10] Sam Owre, John M Rushby & Natarajan Shankar (1992): *PVS: A Prototype Verification System*. In: *International Conference on Automated Deduction*, Springer, pp. 748–752, doi:10.1007/3-540-55602-8_217.

[11] Sam Owre, Natarajan Shankar, John M Rushby & David WJ Stringer-Calvert (1999): *PVS system guide*. Technical Report, Computer Science Laboratory, SRI International, Menlo Park, CA. Available at `https://pvs.csl.sri.com/doc/pvs-system-guide.pdf`.

[12] M. Palmieri, C. Bernardeschi & P. Masci (2019, to appear): *A Framework for FMI-based Co-Simulation of Human-Machine Interfaces*. *Software and Systems Modeling*.

[13] Maurizio Palmieri, Cinzia Bernardeschi & Paolo Masci (2018): *Co-simulation of Semi-autonomous Systems: The Line Follower Robot Case Study*. In Antonio Cerone & Marco Roveri, editors: *Software Engineering and Formal Methods*, Springer International Publishing, pp. 423–437, doi:10.1007/978-3-319-74781-1_29.

[14] Grant Olney Passmore & Denis Ignatovich (2017): *Formal verification of financial algorithms*. In: *International Conference on Automated Deduction*, Springer, pp. 26–41, doi:10.1007/978-3-319-63046-5_3.

[15] Clément Pit-Claudel & Pierre Courtieu (2016): *Company-Coq: Taking Proof General one step closer to a real IDE*. In: *CoqPL'16: The Second International Workshop on Coq for PL*, Zenodo, doi:10.5281/zenodo.44331.

[16] André Platzer & Jan-David Quesel (2008): *KeYmaera: A hybrid theorem prover for hybrid systems*. In: *International Joint Conference on Automated Reasoning*, Springer, pp. 171–178, doi:10.1007/978-3-540-71070-7_15.

[17] Florian Rabe (2014): *A Logic-Independent IDE*. *Electronic Proceedings in Theoretical Computer Science* 167, p. 48–60, doi:10.4204/eptcs.167.7.

[18] Makarius Wenzel (2012): *Isabelle/jEdit: A Prover IDE within the PIDE framework*. In: *International Conference on Intelligent Computer Mathematics*, Springer, pp. 468–471, doi:10.1007/978-3-642-31374-5_38.

[19] Makarius Wenzel (2018): *Isabelle/PIDE after 10 years of development*. In: *UITP workshop: User Interfaces for Theorem Provers*. Available at `https://sketis.net/wp-content/uploads/2018/08/isabellepide-uitp2018.pdf`.

The TLA$^+$ Toolbox

Markus A. Kuppe
Microsoft Research, Redmond, WA, USA
makuppe@microsoft.com

TLA$^+$ is a high-level, math-based, formal specification language used at companies such as Amazon and Microsoft to verify designs of distributed and concurrent systems. TLA$^+$ specifications are written with the publicly available TLA$^+$ Toolbox and verified by tools run from it. Those tools include a model checker and a proof system that can be used together on the same spec.

The paper discusses the rationale and drawbacks of the Toolbox's main features. Specifically, it reports on two Toolbox features which — to the best of our knowledge — are novel in the scope of formal IDEs: *CloudTLC* connects the Toolbox with cloud computing to scale up model checking. Moreover, CloudTLC allows to verify large numbers of models concurrently which enables users to explore a specification's design space faster. The expressiveness of TLA$^+$ results in stated expressions that present a challenge for the model checker to evaluate efficiently. To address this challenge, the Toolbox provides a *Profiler* to analyze the evaluation to override inefficient expressions with efficient ones. We provide a summary about the Toolbox's architecture and its test infrastructure to show how others can add new features. The paper concludes with outlining future engineering and research-related work.

1 Introduction

TLA$^+$ is a high-level, math-based, formal specification language used at companies such as Amazon or Microsoft to design, specify, and document mainly concurrent and distributed systems. Systems are specified as formulas expressed in the Temporal Logic of Actions [9, 10], a variant of Pnueli's original linear-time temporal logic [20]. Data structures are represented with Zermelo-Fränkel set theory with choice. TLA$^+$ is an untyped language and thus simpler and more expressive than programming languages [16]. Emitting executable code is not part of the scope of TLA$^+$. It is implementation language agnostic and intends to aid users in finding bugs above the code level.

The PlusCal algorithm language is a formally defined and verifiable pseudocode [11] that translates into TLA$^+$. PlusCal resembles an imperative programming language and is especially well suited to express sequential and shared-memory multithreaded algorithms. However, the PlusCal language is less expressive compared to TLA$^+$ and does not support TLA$^+$ language features such as refinement or fine-grained fairness constraints.

Supporting TLA$^+$ tools to check and reason about TLA$^+$ and PlusCal specifications are the explicit state model checker TLC and the TLA$^+$ proof system (TLAPS). While TLC [23] is used to check a finite model of a specification, TLAPS [1] supports deductive reasoning about a specification with unbounded domains. Auxiliary TLA$^+$ tools are the SANY parser to syntactically and semantically check a TLA$^+$ specification, the PlusCal translator to translate a PlusCal algorithm to TLA$^+$, and a pretty-printer to render TLA$^+$ as $\LaTeX$. The tools listed above are command-line tools.

As the metaphor implies, the TLA$^+$ Toolbox combines the previously mentioned tools into an integrated development environment (IDE). The Toolbox and the underlying tools are publicly available at

To appear in EPTCS.

© Markus A. Kuppe
This work is licensed under the
Creative Commons Attribution License.

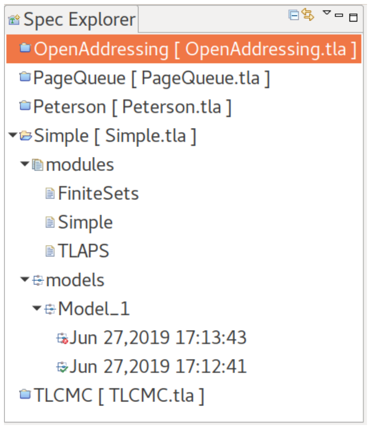
GitHub under the permissive MIT license.[1] The TLA$^+$ project is open to community contributions and has attracted a number of contributors.

This paper is about the 1.6 release of the TLA$^+$ Toolbox and assumes basic knowledge of TLA$^+$ due to space constraints. Relevant TLA$^+$ background material, which is referenced throughout this paper, is [10, 11, 12, 15]. Section 2 describes the main, user-visible features of the Toolbox by showing the *Simple* specification which is part of the TLA$^+$ examples.[2] Section 3 provides an overview of the Toolbox's architecture and test infrastructure, explaining how others can add new features to the Toolbox.

2 Toolbox Features

At a high level, the Toolbox is organized into the Spec Explorer, Spec Editors, a collection of Models, a Trace Explorer, and auxiliary parts such as a general purpose logging part. We subsequently call an individual UI element a part following Eclipse terminology. Parts can be re-arranged freely and stacked on top of each other to best suit a user's preference.

2.1 Spec Explorer



```
Spec Explorer
OpenAddressing [ OpenAddressing.tla ]
PageQueue [ PageQueue.tla ]
Peterson [ Peterson.tla ]
Simple [ Simple.tla ]
  modules
    FiniteSets
    Simple
    TLAPS
  models
    Model_1
      Jun 27,2019 17:13:43
      Jun 27,2019 17:12:41
TLCMC [ TLCMC.tla ]
```

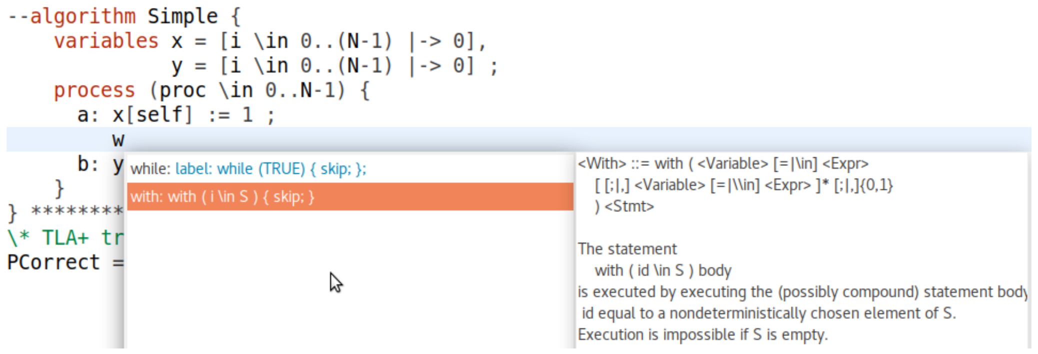
Figure 1: Spec Explorer showing five TLA$^+$ specifications. The active Simple specification consists of three modules and a single Model with two history items.

Following the Eclipse paradigm, the Toolbox can host multiple TLA$^+$ specifications. A specification is a non-empty set of TLA$^+$ modules including a root module and optionally a collection of Models. All specifications are presented in the Spec Explorer (see figure 1) but users can only work on one specification at a time. The Spec Explorer organizes the specifications and provides the main entry point. Specifications can be exported from and imported into the Toolbox on the file-system level. Contrary to most Eclipse-based IDEs, the Toolbox does not use the Eclipse file-system abstraction supporting direct access to the TLA$^+$ modules on the file-system.

2.2 Spec Editor

The central UI part of the Toolbox, using most screen estate, is the Spec Editor. The Spec Editor can stack multiple editors for each module. It provides a set of commands to format TLA$^+$ comments and integrates the pretty-printer. A pretty-printed module opens in an auxiliary part and optionally updates automatically when the specification is saved. This functionality is useful when formatting and layouting a specification for publication. Underpinning the Spec Editor are the syntax and semantic checker (SANY) and the PlusCal translator. The former is automatically invoked when the Spec Editor is saved while the latter translates a PlusCal algorithm to TLA$^+$ when invoked by the user.

[1]`https://github.com/tlaplus/tlaplus`
[2]`https://github.com/tlaplus/Examples/tree/master/specifications/TeachingConcurrency`



```
--algorithm Simple {
    variables x = [i \in 0..(N-1) |-> 0],
              y = [i \in 0..(N-1) |-> 0] ;
    process (proc \in 0..N-1) {
      a: x[self] := 1 ;
         w
      b: y
    }
} ********
\* TLA+ tr
PCorrect =
```

```
while: label: while (TRUE) { skip; };
with: with ( i \in S ) { skip; }
```

```
<With> ::= with ( <Variable> [=|\in] <Expr>
   [ [;|,] <Variable> [=|\\in] <Expr> ]* [;|,]{0,1}
   ) <Stmt>

The statement
   with ( id \in S ) body
is executed by executing the (possibly compound) statement bod
 id equal to a nondeterministically chosen element of S.
Execution is impossible if S is empty.
```

Figure 2: Completion for PlusCal expressions. The syntax and semantics of expressions are shown as well.

2.2.1 PlusCal

For PlusCal, the Spec Editor embeds the PlusCal algorithm as well as its TLA$^+$ translation in a single TLA$^+$ module such that the PlusCal algorithm is nested in a TLA$^+$ comment (compare left editor in figure 3). This design decision allows users to quickly navigate to and from a TLA$^+$ translation which helps to understand the PlusCal expression. The exposure to TLA$^+$ has also been found to encourage users to learn TLA$^+$. Lastly, users will modify the translation, such as adding TLA$^+$ print statements, when debugging of a specification becomes necessary.

Combining a PlusCal algorithm and its TLA$^+$ translation into a single specification file poses the risk that the translation runs out of sync unnoticed. To alleviate this problem, a TLC feature is pending implementation which — prior to model checking — will issue a warning if a translation has become stale.[3]

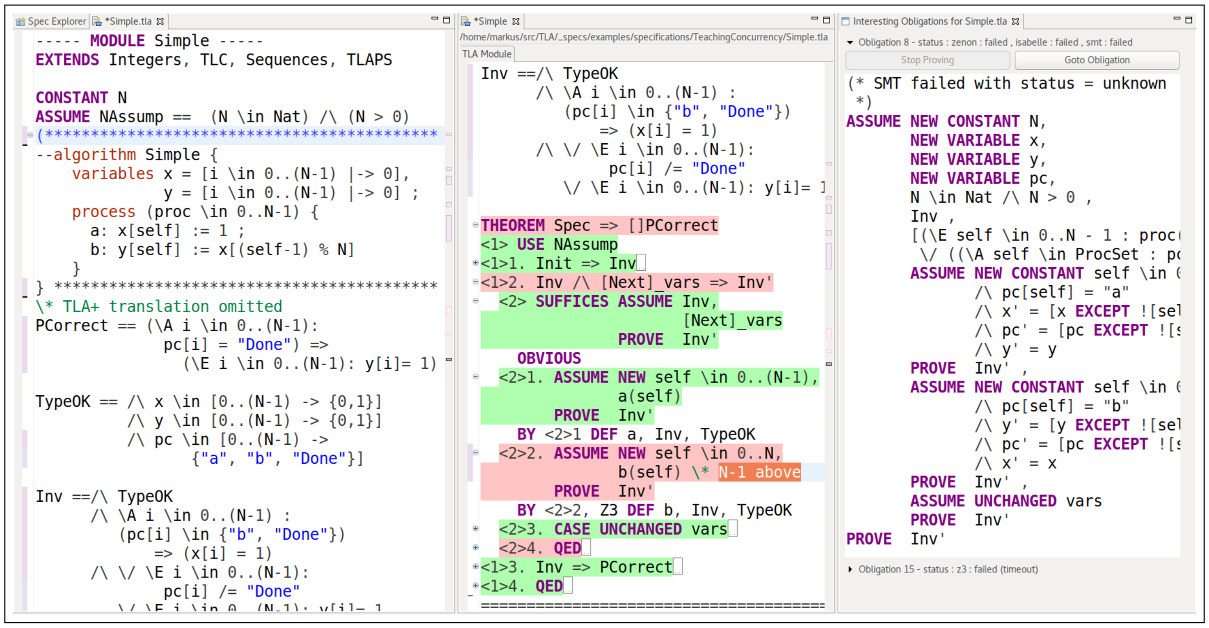
As seen in figure 2, the Spec Editor provides templates for PlusCal expressions. The goal of templates is not to speed up typing — which is of lower importance when specifying systems — but to guide novice users by putting the syntactical and semantic documentation of PlusCal expressions at their fingertips. It also provides engineers with a more familiar user-experience known from programming IDEs. Additionally, completion for TLA$^+$ operators and variable names is available but limited compared to code completion due to the lack of types or type inference.

2.2.2 TLAPS

As shown in figure 3, a hierarchical TLA$^+$ proof (compare [13]) is represented textually as opposed to structurally. However, the structure is explicitly engraved in the hierarchical nature of the proof. A user can focus on individual proof-steps by collapsing other, irrelevant steps. Steps can be collapsed at any hierarchical level. Furthermore, a proof usually follows below the TLA$^+$ behavior formula of the system being specified, but proofs can also be organized into separate TLA$^+$ modules. In this case it is convenient to open each module in an editor and re-arrange the editors side-by-side.

Decomposing a step from sub-steps down to leaf-steps, that are sufficiently low-level to be accepted by some of the TLAPS back-end verifiers, is handled by a built-in editor command. The command is implemented in the Toolbox as a single-page wizard such that users can interactively explore the logical decomposition of a step. When a user is satisfied, the chosen decomposition replaces the decomposed

[3] `https://github.com/tlaplus/tlaplus/issues/296`


Spec Explorer
*Simple.tla
*Simple
/home/markus/src/TLA/_specs/examples/specifications/TeachingConcurrency/Simple.tla
TLA Module
Interesting Obligations for Simple.tla
Stop Proving
Goto Obligation

```
----- MODULE Simple -----
EXTENDS Integers, TLC, Sequences, TLAPS

CONSTANT N
ASSUME NAssump ==  (N \in Nat) /\ (N > 0)
(*******************************************
--algorithm Simple {
    variables x = [i \in 0..(N-1) |-> 0],
              y = [i \in 0..(N-1) |-> 0] ;
    process (proc \in 0..N-1) {
      a: x[self] := 1 ;
      b: y[self] := x[(self-1) % N]
    }
} ******************************************
\* TLA+ translation omitted
PCorrect == (\A i \in 0..(N-1):
              pc[i] = "Done") =>
                (\E i \in 0..(N-1): y[i]= 1)

TypeOK == /\ x \in [0..(N-1) -> {0,1}]
          /\ y \in [0..(N-1) -> {0,1}]
          /\ pc \in [0..(N-1) ->
                 {"a", "b", "Done"}]

Inv ==/\ TypeOK
      /\ \A i \in 0..(N-1) :
         (pc[i] \in {"b", "Done"})
            => (x[i] = 1)
      /\ \/ \E i \in 0..(N-1):
              pc[i] /= "Done"
         \/ \E i \in 0..(N-1): y[i]= 1
```

```
Inv ==/\ TypeOK
      /\ \A i \in 0..(N-1) :
         (pc[i] \in {"b", "Done"})
            => (x[i] = 1)
      /\ \/ \E i \in 0..(N-1):
              pc[i] /= "Done"
         \/ \E i \in 0..(N-1): y[i]= 1

THEOREM Spec => []PCorrect
<1> USE NAssump
<1>1. Init => Inv
<1>2. Inv /\ [Next]_vars => Inv'
  <2> SUFFICES ASSUME Inv,
                      [Next]_vars
               PROVE  Inv'
    OBVIOUS
  <2>1. ASSUME NEW self \in 0..(N-1),
               a(self)
        PROVE  Inv'
    BY <2>1 DEF a, Inv, TypeOK
  <2>2. ASSUME NEW self \in 0..N,
               b(self) \* N-1 above
        PROVE  Inv'
    BY <2>2, Z3 DEF b, Inv, TypeOK
  <2>3. CASE UNCHANGED vars
  <2>4. QED
<1>3. Inv => PCorrect
<1>4. QED
========================================
```

```
Obligation 8 - status : zenon : failed , isabelle : failed , smt : failed
(* SMT failed with status = unknown
 *)
ASSUME NEW CONSTANT N,
       NEW VARIABLE x,
       NEW VARIABLE y,
       NEW VARIABLE pc,
       N \in Nat /\ N > 0 ,
       Inv ,
       [(\E self \in 0..N - 1 : proc
        \/ ((\A self \in ProcSet : p
       ASSUME NEW CONSTANT self \in
              /\ pc[self] = "a"
              /\ x' = [x EXCEPT ![se
              /\ pc' = [pc EXCEPT ![
              /\ y' = y
       PROVE  Inv' ,
       ASSUME NEW CONSTANT self \in
              /\ pc[self] = "b"
              /\ y' = [y EXCEPT ![se
              /\ pc' = [pc EXCEPT ![
              /\ x' = x
       PROVE  Inv' ,
       ASSUME UNCHANGED vars
       PROVE  Inv'
PROVE  Inv'
Obligation 15 - status : z3 : failed (timeout)
```

Figure 3: A (split) Spec Editor showing a PlusCal algorithm and related safety properties (left) as well as its partially collapsed TLAPS proof (center). The <2>2 leaf step failed to prove (red). The right-hand side displays the expanded hypothesis of the failed obligation.

step in the actual proof. Another editor command exists in order to renumber steps if necessary. After invoking the proof system to verify a proof obligation, the selected proof steps are colorized to indicate successfully proven, omitted, and failed steps. Failed proof obligations are additionally displayed in the logging part with detailed information about the hypothesis (compare right-hand side of figure 3).

TLAPS does not re-prove an obligation by maintaining a fingerprint of every proof obligation (see [3]). A user can force TLAPS to forget fingerprints when launching TLAPS from the Toolbox. The Toolbox additionally provides the functionality to select specific TLAPS back-end provers. This functionality is orthogonal to pragmas that can be explicitly stated at individual proof-steps.

Contrary to the model checker TLC, which is part of the Toolbox, users are required to install TLAPS manually. The Toolbox however will automatically detect TLAPS and expose its controls, provided that TLAPS has been installed into the OS-specific standard locations.

2.3 Model

A TLC model constraints the potentially infinite-state machine described by the behavior formula to a finite-state machine. This is done by restricting declared constants to finite domains and by stating bounds for otherwise unbounded variables.[4] Furthermore, the model lists the safety and liveness properties to be checked.

The Toolbox represents a model by a Model part. The Model part however is a superset of the TLC model, because it not only contains the information discussed above but additionally stores the TLC command-line parameters and Java VM settings.

Contrary to the proof system (compare section 2.2), the Toolbox provides a structural representation

[4]An unbounded message buffer will e.g. grow infinitely large.

of a Model by grouping related inputs into tabs and sections. To guide users, inputs are validated when entered. Warnings as well as suggested fixes are placed next to the input controls. For usability reasons and to lower cognitive load, advanced controls are initially collapsed or hidden. The Model is kept in sync with the specification such that relevant changes, e.g. the addition or removal of a constant, are automatically reflected in the Model.

A Model is associated with its respective specification in the Spec Explorer. The Spec Explorer also maintains a history of model checker runs s.t. each history entry stores the model checking result and snapshots of the specification's modules (see figure 1 and compare section 2.3.2). Modules can be diffed to identify changes between long model checking runs.

At the file-system level, a Model is stored as an XML file and can thus be stored in e.g. a source code management system to later reproduce model checking runs.

2.3.1 CloudTLC

The CloudTLC feature is a push-button solution to remotely run model checking on a set of cloud instances. It first provisions cloud instances by deploying a Java VM and TLC and by copying the specification as well as the Model to be checked. Afterwards, CloudTLC starts TLC as defined by the Model (compare section 2.3). If a user opted to run model checking on multiple instances, CloudTLC starts TLC in distributed mode (see [6]). Upon completion of model checking, CloudTLC instances terminate automatically after a grace period.

During model checking, the Toolbox remains connected to the CloudTLC instance to receive progress information and final results. Alternatively, the Toolbox may disconnect at any time. CloudTLC sends the model checking results to a user-provided email address formatted s.t. the result can be imported into the Toolbox. In other words, CloudTLC is completely transparent: Toolbox features such as the Profiler or Trace Explorer (see section 2.3.3 and 2.3.4 below) work as if model checking runs locally.

CloudTLC can alternatively be started by running the Toolbox in command-line mode which is useful for e.g. automation. This feature is used by the TLC performance test suite (compare section 3.2).

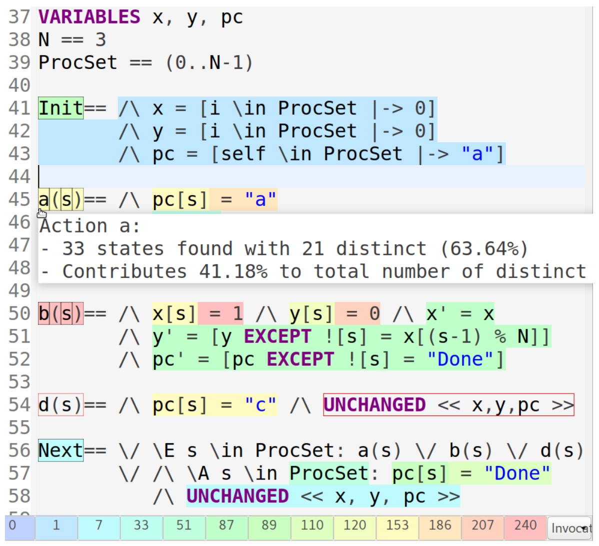
CloudTLC has been implemented for Microsoft Azure, Amazon AWS, and Packet Net. Adding additional IaaS providers requires little effort because CloudTLC is based on a multi-cloud toolkit (see 3.1.3). Its startup time is dominated by how long it takes an IaaS provider to spin-up instances. To reduce startup time, subsequent model checker runs re-use previously provisioned instances unless instances have already terminated.

CloudTLC allows users to shift from a primarily sequential model checking workflow to one where any number of models can be checked concurrently. Additionally, it scales up model checking to more powerful hardware. This allows design variants or competing optimizations to be quickly explored in parallel which either reduces overall specification time or enables a broader exploration of the design space.

2.3.2 Results

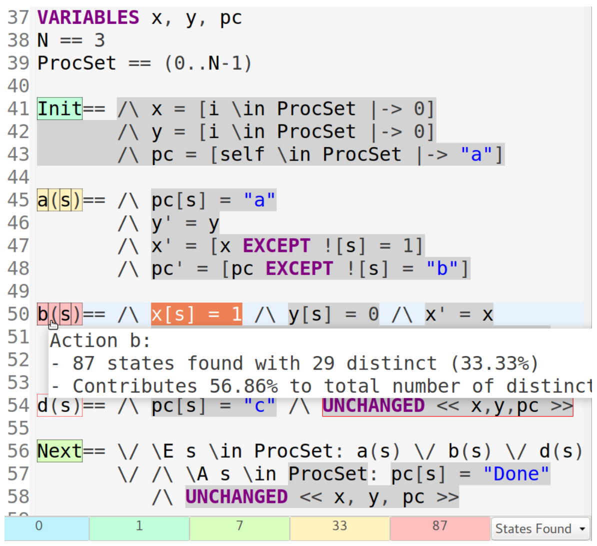
Model checking progress as well as final results are reported in the Results part. Results include start and end measured in wall clock time, the probability of an incomplete state space exploration due to distinct states falsely being considered equivalent, as well as action (see section 2.3.3 below) and state space statistics.

State space statistics include the diameter of the state graph, the distinct as well as the total number of states. Distinct states equals the order of the state graph where total states counts the number of states



```
VARIABLES x, y, pc
N == 3
ProcSet == (0..N-1)

Init== /\ x = [i \in ProcSet |-> 0]
       /\ y = [i \in ProcSet |-> 0]
       /\ pc = [self \in ProcSet |-> "a"]

a(s)== /\ pc[s] = "a"
Action a:
- 33 states found with 21 distinct (63.64%)
- Contributes 41.18% to total number of distinct

b(s)== /\ x[s] = 1 /\ y[s] = 0 /\ x' = x
       /\ y' = [y EXCEPT ![s] = x[(s-1) % N]]
       /\ pc' = [pc EXCEPT ![s] = "Done"]

d(s)== /\ pc[s] = "c" /\ UNCHANGED << x,y,pc >>

Next== \/ \E s \in ProcSet: a(s) \/ b(s) \/ d(s)
       \/ /\ \A s \in ProcSet: pc[s] = "Done"
          /\ UNCHANGED << x, y, pc >>
```

(a)



```
VARIABLES x, y, pc
N == 3
ProcSet == (0..N-1)

Init== /\ x = [i \in ProcSet |-> 0]
       /\ y = [i \in ProcSet |-> 0]
       /\ pc = [self \in ProcSet |-> "a"]

a(s)== /\ pc[s] = "a"
       /\ y' = y
       /\ x' = [x EXCEPT ![s] = 1]
       /\ pc' = [pc EXCEPT ![s] = "b"]

b(s)== /\ x[s] = 1 /\ y[s] = 0 /\ x' = x
Action b:
- 87 states found with 29 distinct (33.33%)
- Contributes 56.86% to total number of distinct
d(s)== /\ pc[s] = "c" /\ UNCHANGED << x,y,pc >>

Next== \/ \E s \in ProcSet: a(s) \/ b(s) \/ d(s)
       \/ /\ \A s \in ProcSet: pc[s] = "Done"
          /\ UNCHANGED << x, y, pc >>
```

(b)

Figure 4: Action and evaluation metrics for the global scope overlayed onto the Spec Editor. Red boxes indicate actions that are never enabled and dead expressions. The hover help displays TLA$^+$ action metrics. Clicking the heatmaps at the bottom selects the corresponding expression or action.

generated as part of model checking. By definition, the number of distinct states is always equal to or less than the number of total states. The ratio of distinct to total states approximates the degree of the state graph.

Statistics can be plotted to predict the remaining model checking time.[5] Furthermore, a state graph may be visualized graphically. This however is restricted to small graphs due to the inherent complexity of layouting directed graphs. Large graphs may be exported to specialized tools such as Cytoscape [22].

2.3.3 Profiler

Experience shows that engineers primarily rely on model checking as opposed to theorem proving to verify designs and optimizations. Such users are therefore interested in checking large models which makes them not only the target of CloudTLC (see section 2.3.1), but also of the Toolbox's Profiler.

Profiling a TLA$^+$ specification collects four different types of metrics grouped into evaluation and action metrics. Evaluation metrics collect the invocation count and the cost of evaluating expressions at the global as well as at the call-chain level. To explain their differences, we will first assume an identical, constant cost for all expressions which would allow to identify the biggest contributor to overall model checking time by simply looking at invocations. However some expressions require the model checker to explicitly enumerate data structures as part of their evaluation for which costs are the quantitative measure: Let S be a set of natural numbers from N to M such that $N \ll M$ and let $\forall s \in SUBSET\ S : s \subseteq S$ be a TLA$^+$ expression. The cost of evaluating the expression equals the number of operations required by the model checker to enumerate the powerset of S which will clearly be a major contributor to model checking time even if its number of invocations is low. Action metrics on the other hand are the number of total and distinct states reported at the TLA$^+$ action level (compare section 2.3.2). They

[5]The cardinality of the set of unexplored states plotted over time usually describes a parabola.

allow to quantify state space explosion — exponential growth of the state space associated with a linear increase in the size of the specification [2] — at the action level. Action metrics distinguish TLC's Profiler from implementation profilers.

The Toolbox overlays a metric upon the Spec Editor by highlighting locations with colors chosen from a one-dimensional heatmap based on the corresponding metric values. For example, expressions with the highest number of invocations will be highlighted with a red color whereas expression with zero invocations will be dark blue. The heatmap shows up as a legend at the bottom of the Spec Editor and reveals the corresponding editor location when clicked. Users can switch the overlay between invocation counts, costs, the sum of invocation counts and costs, the total number as well as the distinct number of states. Additionally, users can narrow the overlay to single call-chains by selecting individual expressions on the call-chain. To further discuss the Profiler's functionality and representation, we will consider the TLA$^+$ specification shown in figure 4a, which is a deliberately inefficient translation of the Simple algorithm shown in figure 3. The enablement predicate on line 50 of action b, consisting of the two expressions $x\,[s] = 1$ and $y\,[s] = 0$, has been invoked the most globally. Related, action b has been found to produce 87 states of which only 29 are distinct (see hover help in figure 4b). In contrast, action a is much more efficient in terms of its total to distinct states ratio. This draws the user's attention to the enablement predicate of action b which is weaker compared to the predicate $pc\,[s] = "a"$ of action a. The enablement predicate of b is true of states for any value of pc. While this does not violate the specification's safety properties, changing the enablement predicate of b to $pc\,[s] = "b"$ puts the total to distinct state ratio in the region of action a and corresponds to the regular translation of the Simple algorithm. Additionally, the enablement of action d on line 54 is evaluated 153 times. Yet the numbers of total and distinct states are zero as indicated by the red boxes. This either indicates a spec error or suggests the removal of action d.

In summary, the Profiler makes different kinds of inefficiencies explicit including the ones discussed above. On this information users can take appropriate measures. For inefficiencies related to the evaluation of expressions, users can override TLA$^+$ operators with more efficient variants up to the extreme where TLA$^+$ operators are overridden with functions implemented in Java. These TLC module overrides are usually significantly faster to evaluate.[6]

The Profiler neither requires the specification nor the model to be modified in order to collect metrics. However, profiling has a non-negligible performance overhead and should be disabled when checking large models. Collecting profiling measurements is unavailable when TLC runs in distributed mode.

2.3.4 Trace Explorer

Should model checking find a violation of any of the stated safety or liveness properties, the corresponding error trace will be visualized in the Trace Explorer as shown in figure 5. An error trace is a sequence of states. A state is an assignment of values to variables.

For enhanced readability, the Trace Explorer uses a color mapping to visually discern changed top-level and nested variable values. Navigating from a state in the trace to the location of the corresponding TLA$^+$ action and — if applicable — PlusCal expressions in the Spec Editor is supported too. The Trace Explorer can handle traces with thousands of states.

To study traces, the Trace Explorer supports the evaluation of trace expressions. A trace expression is an ad-hoc definition of a TLA$^+$ expression with the expressiveness up to TLA$^+$ actions; relations between a state and its successor state [9, p. 4-5]. A trace expression may optionally be named to

[6] `http://jmh.morethan.io/?sources=https://raw.githubusercontent.com/tlaplus/tlaplus/master/tlatools/test-benchmark/tlc2/tool/ModuleOverwrites-1531220029-80dc6de2b.json`

facilitate expression composition. A trace expression may be built from all operators in the scope of the root module. In addition, two built-in operators are available:

_TEPosition Equal to the position of the corresponding state in the error trace

_TETrace A TLA$^+$ sequence of states such that `_TETrace[_TEPosition]` equals the state at position `_TEPosition` in the error trace

Note that the two built-in operators increase the expressiveness of trace expressions beyond the expressiveness of actions by allowing expressions to be formed from the collection of variables of all states of the trace. For example, an expression can compare the values of variables of two or more arbitrary states. This is e.g. useful to format a trace such that it can be copied verbatim to third-party tools (see trace variables `Clock, Process, X,` and `Y` in figure 5 or listing 1 and compare [8]).

At the technical level, the Toolbox evaluates trace expressions by generating a special TLA$^+$ module which is checked with TLC using the functionality discussed in section 3.1.

3 Toolbox Architecture

The TLA$^+$ Toolbox aims to be an industrial strength integrated development environment with support for all aspects of designing, documenting, and specifying TLA$^+$ specifications and PlusCal algorithms. It is designed to be lightweight such that features are implemented at the underlying tool layer to satisfy command-line aficionados, allow re-use by adopters, and integration with automation. Features not core to TLA$^+$, such as graphical visualizations of state spaces or error traces, are left to specialized, third-party tools by exporting relevant information in compatible data formats.

For model checking, the Toolbox stores the system under specification as well as each Model in dedicated files. For a TLAPS proof, the system specified and the proof usually exist in the same file.

To attract a wide range of users, the Toolbox is compatible with the three most common operating systems: macOS, Windows, and Linux. The Toolbox requires no external dependencies except for auxiliary functionality such as generating PDFs. To a large extent, this is due to the fact that the Toolbox is written in Java and built on the Eclipse Rich Client Platform [18]. The Eclipse foundation provides a number of commonly found IDE features such as a help system, a desktop notification system, and an update manager. More abstractly, Eclipse additionally defines usability guidelines and best practices that — while not always applicable to a formal integrated development environment - help enforce a consistent user experience. The Eclipse foundation also allows third-parties to add functionality by contributing extensions and OSGi services [17]. However, to the best of our knowledge this has not yet been leveraged by others.

Building the Toolbox on top of Eclipse is not without drawbacks. The annual release schedule of the Eclipse foundation has recently changed to quarterly releases. Yet, updating the Toolbox to a new Eclipse foundation oftentimes leads to subtle bugs. These bugs are not in the scope of the Toolbox's test suite because they e.g. originate at the Eclipse layers which have their own test suites. Remaining on an older Eclipse release is no alternative either because of the rate at which macOS, Windows, and Linux innovate in turn. Thus, an update requires time-consuming manual testing.

The Spec Explorer (see section 2.1) is the primary interface to work with specification files. For convenience reasons however, users are also allowed to create, move, or modify files at the native file-system layer. This design causes many incompatibilities since Eclipse only supports file modifications through a built-in file-system abstraction. However, Eclipse's higher-level building blocks depend on functionality provided by this abstraction, causing e.g. the Spec Explorer and Spec Editor to lose sync with the native file-system. In hindsight, file modifications should have been restricted to the Toolbox. Alternatively, the limitation could have been addressed at the source - the Eclipse's file-system abstraction - instead of adding workarounds at the Toolbox layer.

Throughout the development of the Toolbox, it has become clear to the authors that the Eclipse foundation has to be considered a whitebox component of the Toolbox. In other words, projects built on top of Eclipse have to accept the burden of co-ownership of Eclipse and help with maintenance and innovation efforts.

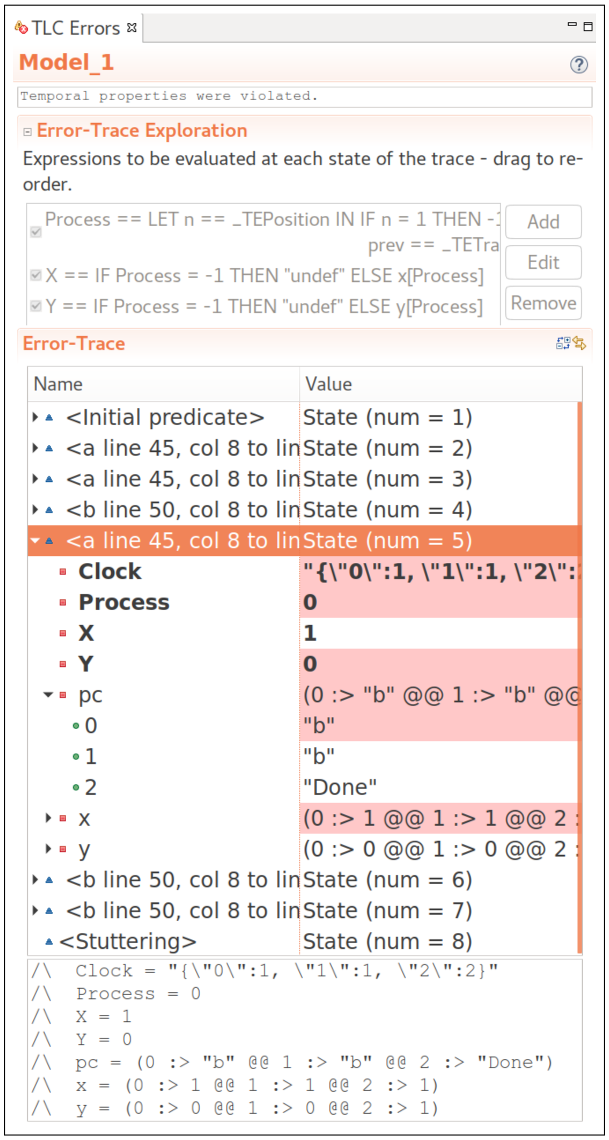
3.1 Back-end integration

The short-lived, lightweight TLA$^+$ tools such as the SANY parser, PlusCal translator, or pretty-printer are executed as part of the Toolbox process for latency reasons. Extending this functionality is only possible via extensions and OSGi services as discussed in section 3. The heavyweight model checker and proof system back-ends however are spawned as separate processes for each verification request. The reasons to spawn separate processes are as follows:

- A back-end cannot be executed in-process because it is implemented in a language that cannot execute on the Java VM. This is true for TLAPS which is written in OCaml.

- Process separation acts as a circuit breaker where a crash of either a back-end or the Toolbox does not interfere with the other process. For example, we do not want a Toolbox crash to also crash a long-running model checker run. This safeguard is especially important while a back-end matures.

- The Java VM's runtime parameters are fixed after Toolbox startup. Running a back-end as part of the Toolbox causes the back-end to inherit the Toolbox's parameters. The resource requirements of back-ends usually do not intersect with those of the Toolbox, i.e. model checkers have very high resource requirements (compare section 2.3.1).



Model_1

```
Temporal properties were violated.
```

Error-Trace Exploration

Expressions to be evaluated at each state of the trace - drag to re-order.

```
Process == LET n == _TEPosition IN IF n = 1 THEN -1
                                prev == _TETra
X == IF Process = -1 THEN "undef" ELSE x[Process]
Y == IF Process = -1 THEN "undef" ELSE y[Process]
```

Error-Trace

Name	Value
<Initial predicate>	State (num = 1)
<a line 45, col 8 to lin	State (num = 2)
<a line 45, col 8 to lin	State (num = 3)
<b line 50, col 8 to lin	State (num = 4)
<a line 45, col 8 to lin	State (num = 5)
Clock	**"{\"0\":1, \"1\":1, \"2\":**
Process	**0**
X	**1**
Y	**0**
pc	(0 :> "b" @@ 1 :> "b" @@
0	"b"
1	"b"
2	"Done"
x	(0 :> 1 @@ 1 :> 1 @@ 2 :
y	(0 :> 0 @@ 1 :> 0 @@ 2 :
<b line 50, col 8 to lin	State (num = 6)
<b line 50, col 8 to lin	State (num = 7)
<Stuttering>	State (num = 8)

```
/\  Clock = "{\"0\":1, \"1\":1, \"2\":2}"
/\  Process = 0
/\  X = 1
/\  Y = 0
/\  pc = (0 :> "b" @@ 1 :> "b" @@ 2 :> "Done")
/\  x = (0 :> 1 @@ 1 :> 1 @@ 2 :> 1)
/\  y = (0 :> 0 @@ 1 :> 0 @@ 2 :> 1)
```

Figure 5: Trace Explorer showing the trace of a violated liveness property. Four trace expressions (bold font) have been evaluated. The definitions of three trace expressions are visible in the second from the top control. The expanded state shows the values for variables declared in the spec (`pc`, `x`, `y`) and via trace expressions.

3.1.1 Toolbox to Back-end

The Toolbox provides a framework to spawn back-ends that hooks into the Eclipse foundation to provide user-visible progress and cancellation functionality.[7] In this framework, a back-end implements an adapter that sets command-line parameters. Primarily, the parameters include the path to the TLA$^+$ specification to verify. In addition, parameters may include performance-specific settings such as the number of cores the back-end may use.

When model checking, the Toolbox additionally serializes a subset of the Model into a plain-text configuration file (compare section 2.3). This configuration contains the definition of the TLA$^+$ behavior specification, invariants and properties to be checked, definitions for all declared constants, and optionally advanced information such as definition overrides as well as state and action constraints. The configuration itself is applicable to model checking in general and not specific to TLC. The configuration is therefore reusable by other back-ends.

3.1.2 Back-end to Toolbox

The Toolbox consumes progress and final results of back-end processes with its parsing framework. This framework is connected to the Spec Editor, Results part, and Trace Explorer following the Model-View-Presenter (MVP) design pattern.[8]

```
" @@ 3 :> "a" @@ 4 :> "a")                                                     1
@!@!@ENDMSG 2217 @!@!@                                                          2
@!@!@STARTMSG 2217:4 @!@!@                                                      3
5: <next_action line 175, col 3 to line 209, col 2 of module TE>                4
/\ X = 1                                                                        5
/\ Y = 0                                                                        6
/\ Process = 2                                                                  7
/\ Clock = "{\"0\":1, \"1\":2, \"2\":1}"                                        8
/\ x = (0 :> 1 @@ 1 :> 1 @@ 2 :> 1)                                             9
/\ y = (0 :> 0 @@ 1 :> 1 @@ 2 :> 0)                                            10
/\ pc = (0 :> "b" @@ 1 :> "Done" @@ 2 :> "b")                                  11
@!@!@ENDMSG 2217 @!@!@                                                         12
@!@!@STARTMSG 2217:4 @!@!@                                                     13
6: <next_action line 220, col 3 to line 254, col 2 of module TE>               14
/\ X =                                                                         15
```

Listing 1: A chunk of TLC back-end output containing a single complete multi-line statement (line 4 to 11). The statement corresponds to the state (constant 2217) of an error trace (constant 4). The state, represented with valid TLA$^+$ (line 5 to 11), equals the expanded state shown in figure 5.

A back-end specific parser has to incrementally consume a stream of back-end output which consists of variably sized chunks of potentially incomplete print statements. For efficiency reasons — error traces occasionally contain thousands of states — and to simplify the implementation of parsers, parsing is based on Toolbox specific formatting of back-end output.[9] It is split into the following three stages: First, a parser buffers chunks of characters into lines separated by a newline character. In the second stage, a parser collects all lines of a multi-line statement. This is possible because both back-ends wrap lines of a multi-line statement with a start and end line tagged by special tokens. Finally, a parser deserializes a multi-line statement into objects as part of the MVP pattern. To map statements to a severity

[7] `https://www.eclipse.org/articles/Article-Concurrency/jobs-api.html`
[8] `https://en.wikipedia.org/wiki/Model-view-presenter`
[9] TLAPS and TLC have command-line parameters to activate their Toolbox specific output mode.

and object types used by the Toolbox, a token contains a tuple of constant numbers. Listing 1 shows an example of a multi-line statement. If applicable, the multi-line statement itself is formatted to be valid TLA$^+$ such as when an error trace is printed.

In summary, the Toolbox provides low-level, text-based frameworks to integrate new back-ends.[10] These frameworks put minimal requirements on individual back-ends which can be implemented in any programming language. Back-ends need not provide sophisticated interfaces such as IPC or a REST API. Instead existing text-based inputs and outputs can be reused.

The flexibility of the low-level and text-based frameworks comes at a price because evolving text-based inputs and outputs is challenging. The lack of schema validation means that bugs can only be caught during runtime. Supporting independently developed and evolving back-ends is expected to be an obstacle. For the model checker, this problem is alleviated by following a synchronized release schedule which is still error-prone and requires significant testing of both the back-end and the Toolbox (see section 3.2). The maturity of the TLAPS back-end means that its inputs and outputs are stable.

Performance problems related to parsing large outputs appeared in the past which had to be addressed by low-level optimizations of the implementation. The frameworks do not support controlling a back-end during execution which would allow e.g. for graceful termination of back-ends. Note that the model checker exposes a subset of its configuration and output via the Java Management Extension [19]. However, the Toolbox does not use this mechanism yet.

3.1.3 CloudTLC Back-end

For CloudTLC, a special back-end has been implemented directly as part of the Toolbox (see 2.3.1 above). It is built with the frameworks discussed in section 3.1 and a multi-cloud toolkit that provides an abstraction from individual IaaS providers.[11]The sequence diagram in figure 6 depicts the interaction between the building blocks of the CloudTLC back-end:

Deploy The Toolbox first queries the chosen IaaS provider via https for specifically tagged CloudTLC instances. If the query result is empty, the Toolkit requests the IaaS provider to launch the given number of instances. If the query is non-empty, it starts the returned instances and skips the following provisioning phase.

Provision Based on an installation script, the Toolbox configures the stock OS and installs dependencies of the model checker. Authentication credentials, needed by later phases of CloudTLC, are copied from the Toolbox's environment variables. The provisioning phase directly communicates with the instances via ssh.

Launch The Toolbox transfers the specification and the model to the instance and starts the model checker remotely. The model checker continuously streams its output through the toolkit to the local TLC parser. The result is additionally sent to a user-provided email address.
The instance remains on-line until manually terminated, in case the Result email cannot be delivered due to e.g. aggressive SPAM protection on the receiver end.

Terminate The instance will wait for subsequent connections before it terminates itself after a given time frame if email delivery in the previous phase succeeded. With some IaaS providers, termination requires an authenticated call to the provider relying on the authentication credentials transferred as part of the previous provisioning phase.

[10]New TLAPS provers should rather be added at the TLAPS layer.

[11]`https://jclouds.apache.org/`

Due to the fact that the hardware specifications of cloud instances are known at development time, the model checker can be optimally deployed and configured. For example, the instance's operating system and the model checker's runtime parameters are chosen based on the hardware specs. The CloudTLC back-end is implemented with TLC in mind but is open to extensions.

3.2 Testing

The Toolbox development follows a combination of the test-driven and the test-last methodologies. Tests are defined at the UI level and are written by Toolbox developers as well as dedicated test engineers.[12] At the time of writing, a test suite of 178 unit, functional and end-to-end tests validate the main workflows of the Toolbox. The tests include workflows that run the model checker. However, the tests do not cover TLAPS yet. Dedicated test suites exist for the model checker and the proof system. The TLC suite consists of 802 unit, integration, and end-to-end tests. The TLAPS test suite contains 56 tests.

The Toolbox's test execution is fully automated and runs as part of the automated build on all three operating systems mentioned above. Builds are executed by a continuous integration system for each commit in the source code repository. While test results do not get published, the build output of the continuous integration system is publicly available.[13] Based on the CloudTLC back-end in command-line mode (compare sections 2.3.1 and 3.1.3), a test suite consisting of user-provided, real-world specifications continuously checks the performance of the model checker. Overall, the test suites provide a useful safety measure to catch functional as well as performance-related regressions early in the development life-cycle.

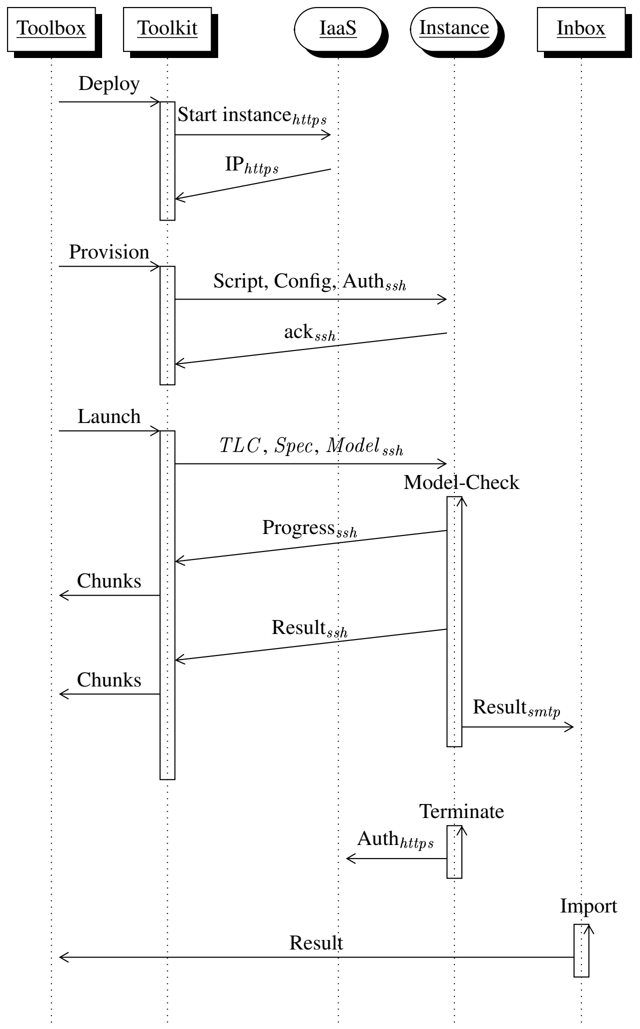


Figure 6: Sequence diagram of CloudTLC. Message labels show the logical payload. Subscripts depict the name of the communication protocol. For brevity reasons, the deploy phase has been simplified and the diagram restricted to show CloudTLC executing with a single instance.

[12]`https://www.eclipse.org/rcptt/`
[13]`https://nightly.tlapl.us/`

4 Conclusion

This paper discussed the high-level, user-visible features of the 1.6 release of the TLA$^+$ Toolbox. It outlined the editing support available in the Toolbox that aids users in writing TLA$^+$ specifications and PlusCal algorithms. The paper introduced the Toolbox's support for combining model checking and deductive reasoning with a proof system. For the model checker, the paper detailed the functionality to analyze error traces. We focused on two features which — to the best of our knowledge — are not found in other formal IDEs: CloudTLC connects the TLA$^+$ Toolbox with cloud computing to satisfy the demand to check larger models and to explore the design space faster. A Profiler helps users understand the evaluation costs of TLA$^+$ expressions thereby aiding them in overriding expressions with more efficient ones. State space statistics at the TLA$^+$ action level allow users to diagnose state space explosion. The paper discussed the architecture and test infrastructure of the Toolbox in hopes of enabling others to add new Toolbox features and to inspire the development of IDEs for more formal languages.

Whether or not the Toolbox can be considered successful in making the TLA$^+$ specification language more accessible is difficult to answer.[14] Its previous release has been downloaded approximately 20k times whereas the standalone model checker has seen one-tenth of this number. The Toolbox has users at major companies such as Amazon and Microsoft and an active community of contributors.

5 Future Work

In the future we intend to add the following features:

TLAPS The main challenge in writing an invariance proof is finding the inductive invariant. Trying to prove an invalid inductive invariant wastes a lot of time. [14] showed that model checking is useful to validate inductive invariant candidates despite the enormous state space attached to it. The trick, that is already implemented in the TLC model checker, is to randomly select and check a subset of all type-correct initial states. With this approach, repeated model checking finds violations with high probability after only a few runs. However, the Toolbox does not yet support this kind of model checking with a special model checking mode that aggregates the results of probabilistic runs.

The idea of taming state space explosion by randomly selecting subsets of all reachable states is additionally useful to validate lower-level proof steps before a user sets out to prove them [15]. The Toolbox should automatically model check proof steps in the background and colorize steps in the Spec Editor if violations are found.

Profiler The Profiler makes it easy to identify TLA$^+$ expressions that are slow to evaluate. Engineers have expressed interest in exploiting TLC module overrides to override inefficient expressions with functions implemented in Java. However, a module override is not trivial to setup which is especially true for users unfamiliar with Java. We will investigate how the Toolbox can better assist users who wish to setup module overrides. The major challenge of which is to assert the equivalence of a module override with the TLA$^+$ expression it is overriding. On a related note, [7] explored the idea of manually translating a TLA$^+$ invariant to Java Pathfinder (compare [4]) to model check module overrides.

[14] The 1.6. Toolbox prompts users to share TLC executions statistics and to identify Toolbox installations.

Trace Exploration The Trace Explorer provides a textural representation of error traces which is ideal to analyze traces with trace expressions. A textural representation on the other hand is not the most appropriate visualization to understand the dynamics of a system. [21] pioneered an error trace animator with which traces can be graphically animated while simultaneously lifting the trace into the problem domain of the system being specified. To not require users to learn a new visualization language, the layout is specified in TLA$^+$. A future Toolbox release will incorporate support to animate error traces as well as include reusable TLA$^+$ modules that provide operators to specify animations at an abstraction layer above basic colors and shapes.

Back-ends The work discussed in [5] gave rise to a symbolic model checker for TLA$^+$, the results of which are encouraging. We wish to integrate this model checker into the Toolbox as described in section 3.1. However, it is an open question how to best combine the advantages of the existing explicit-state and a symbolic model checker. In addition, the symbolic model checker requires users to provide type information for a subset of TLA$^+$ expressions where its type inference fails. How the Toolbox can warn users about missing type information and help users state type annotations is unclear. Type inference, built into the Toolbox, would also allow the Spec Editor to provide a more powerful completion support.

Acknowledgement

The development of the TLA$^+$ Toolbox is a joint effort carried out at the Joint Microsoft Research-INRIA Centre and by Microsoft Research. Among others, Daniel Ricketts and Simon Zambrovski contributed significantly to different aspects of the work presented here. The author would like to thank the anonymous reviewers for their comments and suggestions as well as Leslie Lamport for constructive criticism of the manuscript.

References

[1] Kaustuv C. Chaudhuri, Damien Doligez, Leslie Lamport & Stephan Merz (2008): *A TLA+ Proof System*. *arXiv:0811.1914 [cs]*. Available at http://arxiv.org/abs/0811.1914.

[2] Edmund M. Clarke, William Klieber, Miloš Nováček & Paolo Zuliani (2012): *Model Checking and the State Explosion Problem*. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Bertrand Meyer & Martin Nordio, editors: *Tools for Practical Software Verification*, 7682, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 1–30. Available at http://link.springer.com/10.1007/978-3-642-35746-6_1.

[3] Denis Cousineau, Damien Doligez, Leslie Lamport, Stephan Merz, Daniel Ricketts & Hernán Vanzetto (2012): *TLA+ Proofs*. *arXiv:1208.5933 [cs]*. Available at http://arxiv.org/abs/1208.5933.

[4] Klaus Havelund & Thomas Pressburger (2000): *Model Checking JAVA Programs Using JAVA PathFinder*. *International Journal on Software Tools for Technology Transfer (STTT)* 2(4), pp. 366–381, doi:10.1007/s100090050043. Available at http://link.springer.com/10.1007/s100090050043.

[5] Igor Konnov, Jure Kukovec & Thanh Hai Tran (2018): *BmcMT: Bounded Model Checking of TLA + Specifications with SMT*. Available at https://hal.inria.fr/hal-01899719.

[6] Markus Alexander Kuppe (2014): *Distributed TLC*. Available at http://tla2014.loria.fr/slides/kuppe.pdf. (Accessed 2016-09-06).

[7] Markus Alexander Kuppe (2017): *A Verified and Scalable Hash Table for the TLC Model Checker*. Available at http://www.lemmster.de/talks/MSc_MarkusAKuppe_1497363471.pdf.

[8] Markus Alexander Kuppe (2019): *Visualizing TLA+ Toolbox Error Traces with ShiViz*. Available at `https://github.com/tlaplus/tlaplus/issues/267#issuecomment-481951259`.

[9] Leslie Lamport (1994): *The Temporal Logic of Actions*. ACM Transactions on Programming Languages and Systems 16(3), pp. 872–923, doi:10.1145/177492.177726. Available at `http://portal.acm.org/citation.cfm?doid=177492.177726`.

[10] Leslie Lamport (2003): *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston.

[11] Leslie Lamport (2009): *The PlusCal Algorithm Language*. In Martin Leucker & Carroll Morgan, editors: Theoretical Aspects of Computing - ICTAC 2009, 5684, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 36–60. Available at `http://link.springer.com/10.1007/978-3-642-03466-4_2`.

[12] Leslie Lamport (2012): *How to Write a 21st Century Proof*. Journal of Fixed Point Theory and Applications 11(1), pp. 43–63, doi:10.1007/s11784-012-0071-6. Available at `http://link.springer.com/10.1007/s11784-012-0071-6`.

[13] Leslie Lamport (2012): *How to Write a 21st Century Proof*. Journal of Fixed Point Theory and Applications 11(1), pp. 43–63, doi:10.1007/s11784-012-0071-6. Available at `http://link.springer.com/10.1007/s11784-012-0071-6`.

[14] Leslie Lamport (2018): *Using TLC to Check Inductive Invariance*. Available at `http://lamport.azurewebsites.net/tla/inductive-invariant.pdf`. (Accessed 2018-08-16).

[15] Leslie Lamport (2019): *Proving Safety Properties*. Available at `https://lamport.azurewebsites.net/tla/proving-safety.pdf`.

[16] Leslie Lamport & Lawrence C. Paulson (1999): *Should Your Specification Language Be Typed*. ACM Transactions on Programming Languages and Systems 21(3), pp. 502–526, doi:10.1145/319301.319317. Available at `http://portal.acm.org/citation.cfm?doid=319301.319317`.

[17] D. Marples & P. Kriens (Dec./2001): *The Open Services Gateway Initiative: An Introductory Overview*. IEEE Communications Magazine 39(12), pp. 110–114, doi:10.1109/35.968820. Available at `http://ieeexplore.ieee.org/document/968820/`.

[18] Jeff McAffer, Jean-Michel Lemieux & Chris Aniszczyk (2010): *Eclipse Rich Client Platform*, 2nd ed edition. The Eclipse Series, Addison-Wesley, Upper Saddle River, NJ. OCLC: ocn262433527.

[19] J. Steven Perry (2002): *Java Management Extensions*, 1st ed edition. O'Reilly, Beijing ; Cambridge [Mass.].

[20] Amir Pnueli (1977): *The Temporal Logic of Programs*. IEEE, pp. 46–57, doi:10.1109/SFCS.1977.32. Available at `http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4567924`.

[21] William Schultz (2018): *An Animation Module for TLA+*. Available at `https://easychair.org/smart-slide/slide/8V76#`.

[22] P. Shannon (2003): *Cytoscape: A Software Environment for Integrated Models of Biomolecular Interaction Networks*. Genome Research 13(11), pp. 2498–2504, doi:10.1101/gr.1239303. Available at `http://www.genome.org/cgi/doi/10.1101/gr.1239303`.

[23] Yuan Yu, Panagiotis Manolios & Leslie Lamport (1999): *Model Checking TLA+ Specifications*. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, Laurence Pierre & Thomas Kropf, editors: Correct Hardware Design and Verification Methods, 1703, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 54–66. Available at `http://link.springer.com/10.1007/3-540-48153-2_6`.

Simulation under arbitrary temporal logic constraints

Julien Brunel David Chemouil
ONERA DTIS and Université fédérale de Toulouse, France

Alcino Cunha Nuno Macedo
INESC TEC and Universidade do Minho, Portugal

Most model checkers provide a useful simulation mode, that allows users to explore the set of possible behaviours by interactively picking at each state which event to execute next. Traditionally this simulation mode can not take into consideration additional temporal logic constraints, such as arbitrary fairness restrictions, substantially reducing its usability for debugging the modelled system behaviour. Similarly, when a specification is false, even if all its counter-examples combined also form a set of behaviours, most model checkers only present one of them to the user, providing little or no mechanism to explore alternatives. In this paper, we present a simple on-the-fly verification technique to allow the user to explore the behaviours that satisfy an arbitrary temporal logic specification, with an interactive process akin to simulation. This technique enables a unified interface for simulating the modelled system and exploring its counter-examples. The technique is formalised in the framework of state/event linear temporal logic and a proof of concept was implemented in an event-based variant of the Electrum framework.

1 Introduction

Model checking is one of the most successful techniques for analysing systems, largely due to the ability to automatically verify whether a temporal logic specification holds in a model of a system. Model validation and debugging is essential when analysing a system, and most model checkers provide a simulation mode where the user can explore alternative system traces by choosing how to proceed with the exploration. With most tools, it is possible to choose one of the possible successor states randomly. Additionally, in order to provide a finer control and speed up the debugging process, many tools also allow the user to interactively pick which event to execute next (if the modelling language has some notion of event/action) and/or one of the next possible states (to support the exploration of non-deterministic events, both features must be provided). These simulation modes are quite intuitive and can even be used by problem domain experts unfamiliar with model checking to help validate the model.

Unfortunately this simulation mode only takes into account the system model, traditionally specified by some sort of transition system or a set of events. However, in some situations it would be extremely helpful to perform such simulation under additional constraints, for example to assess the impact of imposing arbitrary fairness constraints. Such constraints reduce the set of valid behaviours and simulation could help the user validate and better understand their impact (which is not always trivial to infer). Similarly, when model checking a given temporal logic property it could be very useful to explore the set of behaviours that falsify it (its set of counter-examples) with a similar simulation technique. Currently most model checkers display a single counter-example when a property is false. As a consequence, the user often inspects the (lone) counter-example to locate the possible source of the problem, changes the model (or specification) to address it, only for the model checker to reveal a different counter-example to the same property. The ability to explore distinct counter-examples at once could allow the user to identify a more general fix, thus tightening the check / analyse / fix loop and making the overall model checking process more efficient.

In this paper we propose a simulation technique that explores the set of the behaviours that satisfy (or falsify) an arbitrary temporal logic specification. At any point the user can focus on a particular state of

Preliminary Report. Final version to appear in:
F-IDE 2019

© J. Brunel, D. Chemouil, A. Cunha, & N. Macedo
This work is licensed under the
Creative Commons Attribution License.

a trace, see which alternative events enable the same trace prefix to be extended into a complete valid behaviour (another infinite trace satisfying the property), and follow any of those to proceed with the exploration. While traditional simulation is rather easy to implement efficiently for any model resembling a transition system, it is unclear how to do so when additional constraints are imposed. This paper explores the viability of a rather naïve on-the-fly technique: when a state is focused, multiple queries to the model checker are run in the background to determine which events can be further explored, while still preserving the same trace prefix. To tame the complexity in models with many events (or parametrised ones), type categorisation is supported: the user first focuses on a specific type and only then iterates over the different events of that type.

This paper is structured as follows. In the next section we very briefly discuss some alternative techniques to explore the set of behaviours that satisfy (or falsify) a given property. In Section 3 we formalise our proposal in the general setting of event/state linear temporal logic. Section 4 presents a prototype implementation of the proposed technique in the Electrum Analyzer [2], the model checker for the Electrum language [8], an extension of Alloy [7] with linear time temporal logic. The goal of this prototype is mainly to show the viability of the approach, namely in terms of user-experience and efficiency. Section 5 wraps-up the paper and presents some ideas for future work.

2 Related work

Some techniques have been proposed to explore of the set of behaviours that satisfy (or falsify) a given property. The simplest ones just provide iteration over such set, by independently displaying one trace at a time. This can be achieved by changing an explicit model checking engine to resume search after finding one counter-example trace, or, in the case of a SAT-based symbolic bounded model checker, by incrementally adding new clauses that exclude exactly the previous trace, as implemented in the Electrum Analyzer [2] developed by the authors. The problem is that this frequently keeps yielding traces that are just slight variations of each other and, since the full set of behaviours is usually too big to be enumerated, finding interesting variations may prove infeasible. To alleviate this problem, for specific modelling languages it is possible to define reasonable equivalence classes on traces (e.g., traces that follow the same control-flow path are deemed equivalent), and implement iteration by restarting the model checker with a modified property that conjoins the original one with a formula excluding all traces in the class of the previous counter-example [6, 3].

Problem domain expertise, namely some kind of user input, could lead to more effective exploration. While in the above techniques user interaction is limited to just asking for the next trace, in [5], by running multiple queries to the model checker, a proof tree of a CTL property is inferred to "explain" a counter-example trace, with which the user can interact to ask for new counter-examples. Possible interactions include asking for alternative proofs (e.g., in a disjunction node), or guiding the search to explore different parts of the model (e.g., in `EX` ϕ nodes, by choosing the next ϕ-satisfying state). However, this approach requires substantial knowledge of the underlying proof system for CTL and it is not clear how it can be generalised to support LTL and fairness constraints.

3 Formalisation

Most systems incorporate both the notion of states and events. *State/event linear temporal logic* (SE-LTL) was proposed to allow a more concise and intuitive specification in these cases [4]. The semantics of a formula in this logic is defined over a *labelled Kripke structure* (LKS), a tuple $(S, I, P, \mathscr{L}, T, \Sigma, \mathscr{E})$ where S

is a finite set of states, $I \subseteq S$ the set of initial states, P a finite set of atomic propositions, $\mathscr{L} : S \to 2^P$ a state labelling function, $T \subseteq S \times S$ a transition relation, Σ a finite set of events, and $\mathscr{E} : T \to 2^\Sigma \setminus \{\emptyset\}$ a transition labelling function. The transition relation is assumed to be total, so every state has at least one successor. To enable a more efficient exploration, events are categorized with a function $\mathscr{T} : \Sigma \to \Upsilon$ that assigns a type to each event. This categorization is natural in many models, namely those with parametrised events. A *path* $\pi = \langle s_0, a_0, s_1, a_1, \ldots \rangle$ of such a *typed LKS* is an alternating infinite sequence of states and events where $\forall i \cdot (s_i, s_{i+1}) \in T \wedge a_i \in \mathscr{E}(s_i, s_{i+1})$ and $s_0 \in I$.

Given a typed LKS, SE-LTL formulas are defined by the following grammar, where p ranges over P, a over Σ, and t over Υ:

$$\phi ::= p \mid a \mid t \mid \top \mid \neg \phi \mid \phi \wedge \phi \mid \mathtt{X}\,\phi \mid \mathtt{G}\,\phi \mid \mathtt{F}\,\phi \mid \phi\,\mathtt{U}\,\phi$$

Given a path π, the semantics of a formula is the standard one of LTL with the addition that $\pi \models a$ iff a is the first event of π and $\pi \models t$ iff a is the first event of π and $\mathscr{T}(a) = t$. $M \models \phi$ means that ϕ holds in the typed LKS M, that is, for every path π of M we have $\pi \models \phi$. Given a formula ϕ the goal of a model checker is to find a path π such that $\pi \not\models \phi$. We will denote the first such counter-example, if it exists, by $M(\phi)$. Given a path π, $[\pi]_i$ is a formula that exactly characterises the prefix of π up to i, defined as $([s_0] \wedge a_0) \wedge \mathtt{X}([s_1] \wedge a_1) \wedge \ldots \wedge \mathtt{X}^{i-1}([s_{i-1}] \wedge a_{i-1})$, where $\mathtt{X}^i$ is a nesting of i "next" operators and $[s]$ is a formula that fixes the values of the propositions of state s, defined as the conjunction of all propositions appearing in $\mathscr{L}(s)$ and all negated propositions in $P - \mathscr{L}(s)$.

Following [9], our interactive exploration technique is specified by a set of scenario exploration operations. The state of the exploration is a tuple (ϕ, π, i, Φ) where ϕ is the formula being model checked[1], π the current counter-example on display, i the state the user is focused in, and Φ a function mapping each path index to a formula that characterises the set of states and transitions the model checker is allowed to explore at that point. Notation $\Phi \oplus \{i\,..\,j\} \mapsto \psi$ will denote an update on this last function, that maps every index between i and j to ψ, keeping all other indexes intact. When updating a single index i, the notation will be simplified to $\Phi \oplus i \mapsto \psi$.

When first checking a property ϕ this state is initialised as $(\phi, M(\phi), 0, \mathbb{N} \mapsto \top)$. Basic navigation operations can then be used to inspect the counter-example, namely $\rhd\!\rhd(\phi, \pi, i, \Phi) = (\phi, \pi, i+1, \Phi)$ and $\lhd\!\lhd(\phi, \pi, i, \Phi) = (\phi, \pi, i-1, \Phi)$ (for $i > 0$). At any point i it is possible to ask for a new counter-example that differs only in the outcome of the previous event, a useful operation to explore non-determinism. This operation is defined as $\rhd(\phi, \pi, i, \Phi) = (\phi, M(\varphi), i, \Phi \oplus (i \mapsto \Phi(i) \wedge \neg[s_i]) \oplus (\{i+1\,..\} \mapsto \top))$, where φ is $\phi \vee \neg([\pi]_i \wedge \mathtt{X}^i(\Phi(i) \wedge \neg[s_i]))$. By repeatedly applying $\rhd$ all possible outcomes of the previous action will eventually be enumerated (or possible initial states when $i = 0$). Notice how Φ is used to trim a branch of the behaviour tree when this operation is selected, but maintains memory of previously trimmed branches while inspecting a trace with $\rhd\!\rhd$ and $\lhd\!\lhd$.

Similarly, it is possible to ask for a new counter-example that picks a different next event of the same type. This operation is defined as $\blacktriangleright(\phi, \pi, i, \Phi) = (\phi, M(\varphi), i, \Phi \oplus (i \mapsto \Phi(i) \wedge \neg([s_i] \wedge a_i)) \oplus (\{i+1\,..\} \mapsto \top)$, where φ is $\phi \vee \neg([\pi]_i \wedge \mathtt{X}^i(\Phi(i) \wedge [s_i] \wedge \neg a_i \wedge \mathscr{T}(a_i)))$. Notice how Φ keeps track that the branch starting in $[s_i]$ and labeled with a_i has already been explored. To ask for a new counter-example with a specific type t for the next event, operation $\vartriangle_t\ (\phi, \pi, i, \Phi) = (\phi, M(\varphi), i, \Phi \oplus \{i+1\,..\} \mapsto \top)$ can be used, where φ is defined as $\phi \vee \neg([\pi]_i \wedge \mathtt{X}^i([s_i] \wedge t))$.

[1]To simplify, in the remaining of the paper we will present the technique in the context of counter-example exploration in model checking, but it can obviously be also used for exploring the valid behaviours of a system with additional arbitrary constraints specified over it, by just running the model checker on the negation of their conjunction and interpreting the resulting set of counter-examples as witnesses of the system's behaviour.

```
open util/ordering[Key]
sig Key {} sig Room { keys: set Key, var current: one keys } sig Guest { var gkeys: set Key }
one sig Desk { var lastKey: Room → lone Key, var occupant: Room → Guest }
event In[g: Guest, r: Room, k: Key] modifies gkeys, occupant, lastKey {
   no r.(Desk.occupant) and k = nextKey[r.(Desk.lastKey), r.keys]
   gkeys' = gkeys + g→k
   Desk.occupant' = Desk.occupant + r→g
   Desk.lastKey' = Desk.lastKey ++ r→k }
event Out[g: Guest] modifies occupant { ... }
event Entry[g: Guest, r: Room, k: Key] modifies current { ... }
event Reentry[g: Guest, r: Room, k: Key] { ... }
fun nextKey[k: Key, ks: set Key] : set Key { min[nexts[k] & ks] }
fact Init { keys in Room lone → Key and no Guest.gkeys and ... }
assert BadSafety { always { all r: Room, g: Guest, k: Key |
   (Entry[g,r,k] or Reentry[g,r,k]) and some r.(Desk.occupant) ⇒ g in r.(Desk.occupant) } }
check BadSafety for 3 Key, 1 Room, 2 Guest, 10 Time
```

Figure 1: Hotel example in Electrum with events.

4 Implementation

Electrum is an extension of the popular Alloy formal specification language, developed for the analysis of dynamic systems. An Alloy model consists of a set of static signatures and relations (of arbitrary arity). Properties can be specified in an extension of first-order logic: apart from the standard connectives and quantifiers, Alloy supports closures and some derived relational logic connectives, such as composition (`.`) or Cartesian product (→). To make the verification decidable, the user must specify a scope setting the maximum size of all signatures. Counter-examples are depicted graphically with user-customisable themes. In Electrum, signatures and relations can be declared mutable (with keyword `var`) and properties can be specified using linear temporal logic connectives (including past ones) and primed expressions (denoting their value in the next state) in addition to Alloy connectives.

Recently, we added the notion of event to Electrum [1]. Figure 1 presents an example of an Electrum model with events based on a classic Alloy example that specifies a protocol for disposable room key-cards in a hotel. There are 4 events in this model (check-`In`, check-`Out`, `Entry`, and `Reentry`), each specified declaratively with relational logic and primed expressions. The keyword `modifies` is used to fix the frame. The desired safety property is that only guests registered as occupants of a room can indeed enter that room. Unfortunately, that is not the case and the `check BadSafety` command yields a counter-example trace where a guest checks in, enters the room after checking out, a second guest checks in, and the first guest reenters the room afterwards. This is possible because the door lock has not yet been recoded with the new key issued by the front desk. The previous version of the Electrum Analyzer [2] already allowed the user to ask for full alternative counter-example traces, but each one could only be inspected independently (by navigating backward and forward in the states), making it difficult to understand the relationship between the different counter-examples.

The new prototype interface for simulation and counter-example exploration is depicted in Fig. 2, which illustrates precisely the exploration of the above counter-example at $i = 1$. As in the previous version, the user can focus on a particular state of a trace by navigating backward (⧏⧏) and forward (⊳⊳), using the left- and right-arrows in the bottom toolbar. However, two states are now shown side-by-side, allowing the user to better understand what is the effect of an event. In the top toolbar we also depict the trace and which transition is being inspected, and the bottom toolbar in the middle shows the event that triggers the current transition. Following the formalisation in the previous section, the user can choose a

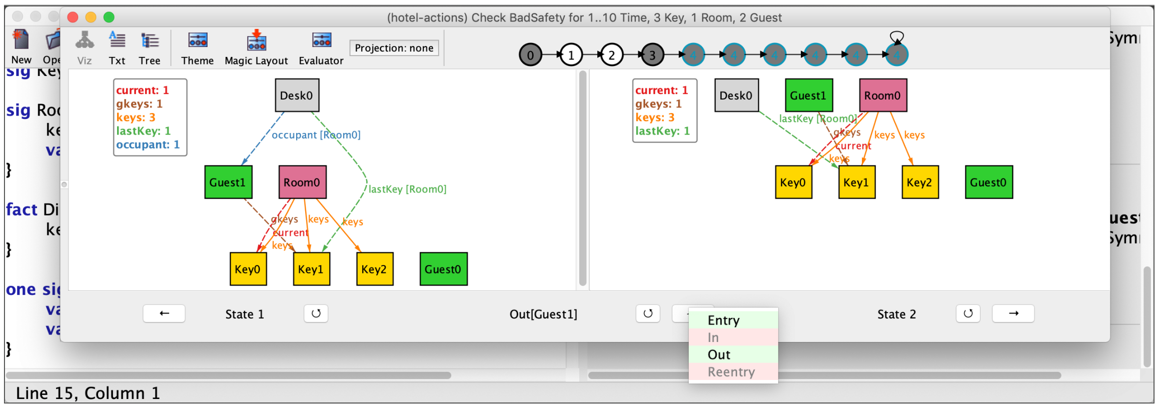


Figure 2: Exploration interface.

different pre- or post-state (the small "reload" buttons under the left- and right-panes, corresponding to operation ▷), an event of the same type with different parameters (the "reload" button next to the event name, corresponding to the ▶ operation), or a different event type to execute (the selection button in the bottom toolbar, implementing the △ operation). When the user focuses on a state, operation △ is dry run on-the-fly to determine which event types are enabled, so that when the event selection button is pressed only the enabled events can be selected (shown with a green background, as opposed to red for the disabled ones). In Fig. 2 we can see that after check-in the only options are for the first guest to check out or enter the room. Unlike in the previous version of the Analyzer, it is now easy to understand that, for the given scope, the check-in of the second guest must necessarily be followed by an entry or reentry of the first guest, and there are no other possibilities to breach safety.

To assess the efficiency of the proposed technique, we measured the required time to determine which △ operations are enabled in the different states of the first counter-example returned by the Analyzer. Table 1 shows the results of this preliminary evaluation for different scopes. The first column (C) shows the configuration (number of guests and a list with the number of keys per room), the second the time (in seconds) to compute the first counter-example (T), and then, for each state i, the total time (in seconds) to compute which event types are enabled (T_i), and the set of enabled events (a_i), with the subscript of each event type identifying the guest involved, and also highlighting the event chosen to be executed in bold. The evaluation was performed with the bounded model checking engine of Electrum (with the Glucose SAT solver), with maximum trace length of 10, in a commodity laptop with a 2.3 GHz Intel Core i5 and 16 GB of RAM. As can be seen, only for $i = 0$ in the last configuration did the solving of all △ events take more than 2s, and in most cases it is in the order of a few hundred ms. Since a user typically needs some time to understand a state after focusing, this delay is almost always unnoticed. Also, times tend to decrease as the user advances in the trace: this is to be expected, since a bigger prefix of the trace is fixed, resulting in a smaller search space for the verification engine.

5 Conclusion

This paper presented a simple technique that allows the user to explore the behaviours that satisfy (or falsify) an arbitrary temporal logic specification, with an interactive process akin to simulation. A prototype was implemented in the Electrum Analyzer, and a preliminary evaluation showed its viability in terms of efficiency. In the future we intend to further improve efficiency by checking which events are

C	T	T_0	a_0	T_1	a_1	T_2	a_2	T_3	a_3	T_4	a_4	T_5	a_5	···
2[3]	0.07	0.33	$\mathbf{I_1}$	0.20	$\mathbf{O_1}$E	0.18	$\mathbf{I_0}$E	0.22	$\mathbf{E_1}$	0.09	$\mathbf{R_1}$OE	0.09	$\mathbf{R_1}$OE	···
2[1,3]	0.06	0.49	$\mathbf{I_1}$	0.30	$\mathbf{E_1}$O	0.27	$\mathbf{O_1}$R	0.23	$\mathbf{I_0}$R	0.26	$\mathbf{R_1}$	0.11	$\mathbf{R_1}$OE	···
3[2,3]	0.11	0.75	$\mathbf{I_2}$	0.34	$\mathbf{O_2}$IE	0.39	$\mathbf{I_1}$E	0.35	$\mathbf{E_2}$I	0.06	$\mathbf{R_2}$IOE	0.07	$\mathbf{R_2}$IOE	···
3[1,1,4]	0.58	1.24	$\mathbf{I_2}$	0.77	$\mathbf{O_2}$E	0.62	$\mathbf{I_1}$E	0.53	$\mathbf{E_2}$O	0.23	$\mathbf{R_2}$OE	0.20	$\mathbf{R_2}$OE	···
4[1,1,6]	1.74	2.30	$\mathbf{I_3}$	1.41	$\mathbf{O_3}$E	1.10	$\mathbf{I_2}$E	0.94	$\mathbf{E_3}$O	0.39	$\mathbf{R_3}$OE	0.33	$\mathbf{R_3}$OE	···

Table 1: Performance of the event type enumeration (times in seconds).

enabled in parallel. To show the generality of the technique we intend to apply it to other model checkers, namely develop a counter-example exploration tool for SMV. Finally, we also plan to conduct a more detailed evaluation, focusing not only on efficiency, but also on its effectiveness, namely in helping the user identify truly different counter-examples.

Acknowledgements

This work is financed by the ERDF - European Regional Development Fund - through the Operational Programme for Competitiveness and Internationalisation - COMPETE 2020 - and by National Funds through the Portuguese funding agency, FCT - Fundação para a Ciência e a Tecnologia, within project POCI-01-0145-FEDER-016826, and the French Research Agency project FORMEDICIS ANR-16-CE25-0007. The third author was also supported by the FCT sabbatical grant with reference SFRH/BSAB/143106/2018.

References

[1] Julien Brunel, David Chemouil, Alcino Cunha, Thomas Hujsa, Nuno Macedo & Jeanne Tawa (2018): *Proposition of an Action Layer for Electrum.* In: *Proceedings of the 6th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ)*, *LNCS* 10817, Springer, pp. 397–402.

[2] Julien Brunel, David Chemouil, Alcino Cunha & Nuno Macedo (2018): *The Electrum Analyzer: Model checking relational first-order temporal specifications.* In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, ACM, pp. 884–887.

[3] Kalou Cabrera Castillos, Hélène Waeselynck & Virginie Wiels (2015): *Show Me New Counterexamples: A Path-Based Approach.* In: *Proceedings of the 8th International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, pp. 1–10.

[4] Sagar Chaki, Edmund M. Clarke, Joël Ouaknine, Natasha Sharygina & Nishant Sinha (2004): *State/Event-Based Software Model Checking.* In: *Proceedings of the 4th International Conference on Integrated Formal Methods (iFM)*, *LNCS* 2999, Springer, pp. 128–147.

[5] Marsha Chechik & Arie Gurfinkel (2007): *A framework for counterexample generation and exploration.* *International Journal on Software Tools for Technology Transfer* 9(5–6), pp. 429–445.

[6] Alma L. Juarez Dominguez & Nancy A. Day (2013): *Generating multiple diverse counterexamples for an EFSM.* Technical Report CS-2013-06, University of Waterloo.

[7] Daniel Jackson (2012): *Software Abstractions: Logic, Language, and Analysis*, 2nd edition. MIT.

[8] Nuno Macedo, Julien Brunel, David Chemouil, Alcino Cunha & Denis Kuperberg (2016): *Lightweight specification and analysis of dynamic systems with rich configurations.* In: *Proceedings of 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, ACM, pp. 373–383.

[9] Nuno Macedo, Alcino Cunha & Tiago Guimarães (2015): *Exploring Scenario Exploration.* In: *Proceedings of the 18th International Conference on Fundamental Approaches to Software Engineering (FASE)*, *LNCS* 9033, Springer, pp. 301–315.

Tool Support for Validation of Formal System Models: Interactive Visualization and Requirements Traceability

Eduard Kamburjan
Department of Computer Science
Technische Universität Darmstadt, Germany
kamburjan@cs.tu-darmstadt.de

Jonas Stromberg
Department of Computer Science
Technische Universität Darmstadt, Germany
jonas.stromberg@stud.tu-darmstadt.de

Development processes in various engineering disciplines are incorporating formal models to ensure safety properties of critical systems. The use of these formal models requires to reason about their adequacy, i.e., to validate that a model mirrors the structure of the system sufficiently that properties established for the model indeed carry over to the real system. Model validation itself is non-formal, as adequacy is not a formal (i.e., mathematical) property. Instead it must be carried out by the modeler to justify the modeling to the certification agency or other stakeholders. In this paper we argue that model validation can be seen as a special form of requirements engineering, and that interactive visualization and concepts from requirements traceability can help to advance tool support for formal modeling by lowering the cognitive burden needed for validation. We present the VisualisierbaR tool, which supports the formal modeling of railway operations and describe how it uses interactive visualization and requirements traceability concepts to validate a formal model.

1 Introduction

The importance of formal methods for safety-critical systems has long been recognized in many engineering disciplines and is demanded or recommended by certification authorities in, e.g., railway engineering [5] and avionic [29] industries. Recently, with the increasing integration of computational parts into devices, digital twins [28] and co-simulation [13] are used to develop new products and prototype changes. One important class of formal methods in this area is formal system modeling.

Under formal system modeling we understand the development of a formal system model of a real system or of a design of a planned system (short: target system) that mirrors the structure and behavior of the target system sufficiently to prototype [17] and/or evaluate changes[1] [13]. Digital twins are a variant of this, which are integrated into the target system. Nonetheless, digital twins are based on a subsystem whose structure they must mirror as close as possible and face the same challenges for validation. Formal system modeling requires model validation to ensure that properties established for the formal model hold for the target system: While verification ensures that the model *behaves correctly*, validation ensures that the *correct thing* was modeled. Model validation itself is not formal (in the sense that it is not a mathematical property), it is an informal process to argue for the adequacy of the model and bridges between the intention of the developer and the realized model. Validation is required to convince safety assessors ,such as certification agencies, that formal proofs have value in the certification process and other stakeholders that the prototypes developed in this model save development time for the target system.

Our main observation in modeling projects with industry partners is that certain stages of formal modeling can be seen as a specific form of requirements engineering.

[1]We contrast formal system modeling with Model-Driven-Development approaches, where the model becomes the final system through refinement.

Submitted to:
F-IDE 2019

© E. Kamburjan & J. Stromberg
This work is licensed under the
Creative Commons Attribution License.

- *Requirements elicitation is model scoping.* Both these processes turn the *implicit* knowledge and assumptions of the user about the domain into an *explicit* representation. More importantly, they also decide on the aspects of the domain that are not needed for a specific model/project.
- *Requirements traceability is model validation.* Both tasks relate parts of the formal system model to the target system, which are the two main artifacts from requirements engineering view. Instead of tracing a requirement to the point where it is realized, one traces an aspect to the point where it is modeled. In reverse, instead of tracing backwards what requirement a part of the implementation is realizing, one traces what aspect a model-part realizes.

However, formal modeling poses challenges that prevent the straightforward adaption of, e.g., Software Engineering practices. The boundary between modeling and programming is not clear [4], but concerning validating programs and formal system models of the described kind the main difference is that model validation of formal system models requires to validate a *white-box* model (in contrast to a black-box model when testing a program) and, in particular, raises the following points:

Cognitive Burden. Significant cognitive burden is required to judge formal system models, as formal modeling languages are not adopted by all industries and there is little training material available. Even in industries which use formal modeling, keeping the cognitive burden low is a desired aspect of adopters of formal methods in industry [24] and the cognitive burden of validation is higher than when designing, e.g., use cases and user stories.

Validating Structure. It is conceptually different to validate the *structure* of a white-box model, than to validate the *behavior* of a black-box model [1]. As formal modeling aims to mirror the structure precisely enough that changes in the model have the same causal effect as their counterpart in the target system, merely describing (by, e.g., test cases) the input/output does not suffice. This thwarts the application of behavior-centric approaches such as Behavior-Driven Development (BDD) [25].

Another experience we make in our work with domain experts is that merely *visualizing* a formal system model is not enough when using it to prototype new ideas [17]. Interaction with the visualization allows even quicker feedback cycles with the domain experts, as it allows them to test a specific situation for validation with little overhead to induce it into the model. This ties in with the above point of lowering the cognitive burden to simplify validation.

These observations raise the question how, and what, techniques for requirements traceability and interactive visualization can be applied to formal modeling. In particular, we are interested in integrating such techniques into an IDE that helps not only with model verification, but also with model validation.

We illustrate with the `VisualisierbaR` tool for formal modeling of railway operations how requirements traceability can be integrated into a formal methods toolkit and describe future research directions for formal modeling languages and toolkits.

Our main contribution is to develop tool support for model validation by intergrating requirements traceability and interactive visualization into an IDE, as well as a tool and a case study illustrating this idea. This work is structured as follows: Sec. 2 describes model validation in railway operations, Sec. 3 gives an overview over a formal model in this domain, Sec. 4 descirbes the implementation of the `VisualisierbaR` tool, Sec. 5 describes the validation features , Sec. 6 gives two case studies from automatic train operations and rule prototyping and Sec. 7 concludes with related and future work.

2 Validation of Railway Operation Models

We describe our approach using the `VisualisierbaR` tool developed for the `FormbaR` model [18] for German railway operations. This section describes the specification of railway operations and the challenges of validating in this domain, while the approach itself is easily generalized to other domains.

2.1 Specification of Railway Operations

Railway operations for German railways are not described by a single document, but by (1) legal regulations, the *"Eisenbahn-Bau- und Betriebsordnung" (Law for Operating and Building Railways)* [9], (2) public rulebooks managed by Deutsche Bahn (DB), in particular Ril. 408 [6] and 819 [7], (3) internal rulebooks for operations, (4) requirements specification for technical elements, (5) training documentation and (6) internal announcements. `FormbaR` only considers the operations of DB, but other railway companies are also bound to the same legal regulations (and to Ril. 408 when using DB infrastructure), in addition to their own internal rulebooks.

In this environment, procedures are not described algorithmically in one place, but are described in a distributed manner. This makes it hard to pinpoint the exact point where the procedure is defined. E.g., the procedure to depart a train is partially described by Ril. 408 [6], partially by the requirements of the specific station interlocking in a station, partially by internal announcements and possibly by local exemptions (*"Lokale Zusätze"*). Implicitly, building regulations are also referenced, as certain minimal distances are assumed to hold.

These procedures are subject to constant change and completely new procedures for automatic train operations (ATO) and ETCS level 3 are in development. When scoping `FormbaR`, it was decided not to model certain rules, because they are not relevant to train operations itself but, e.g., specify interactions with the passengers [18].

2.2 Validating Railway Operation Models

Validation of models of new procedures requires to track each part of the model to the document that specifies it — legal regulations, rulebooks and technical documents are requirements and model validation entails documenting that the requirements are met. This is especially critical if these models are planned to be used for certification.

However, contrary to engineering projects, the form of the requirements is already fixed in a form that is optimized towards other uses – rulebooks are a form, which is difficult to process and which may not be changed during development. In terms of characteristics for software requirements [14], they are neither unambiguous, nor complete or modifiable[2]. Furthermore, while there is a public specification in DOORS format available for the new European Train Control System (ETCS) modes of operations, the other rulebooks are written in plain natural language structured by sections and paragraphs. The use of technical documents that were not intended to be used as requirements for formal system models is not specific to railway engineering but is common in other fields, as formal models are mostly developed after the target system is finished. Similarly using the requirements of the original system can be problematic, as the formal model then expresses what the system is *supposed to do*, in contrast to what it really *does*. Nonetheless this can be of use, e.g., to analyze the design before implementation.

Not the complete model is directly related to requirements: some parts model basic infrastructure. E.g., `FormbaR` contains code for the physical behavior of the train, which is not explicit in any rulebook. For validation it is important to carefully distinguish between basic infrastructure and other model parts, since an error in the basic infrastructure is a mistake of the modeler, while an error in the other parts may hint towards a problem with the target system.

[2]Arguably, they are also neither consistent nor structured by importance.

2.3 Validating `FormbaR`

We use three techniques to validate `FormbaR`: simulation, interactive visualization and traceability.

Simulation. Simulation runs the model on predefined infrastructures and scenarios and checks that the behavior is the one expected by the domain expert, e.g., that after a fault on the infrastructure the train has the expected delay. This roughly corresponds to acceptance testing for software, but does not scale for bigger scenarios, e.g., because propagation of faults is not easily specifiable or predictable. Visualization scales better, as it is easier for a human to assess the visualized situation than to assess (or specify) the expected behavior as a trace.

Interactive Visualization Simulation is only able to detect errors in simple scenarios. A visualization tool shows the state of the whole infrastructure, e.g., the position of the train or the state of the signals. Interactive visualization is not merely a representation of the behavior of the system. The user interacts with the model via the visualization and introduces faults or gives orders to the train.

Interaction extends the use of visualization for validation. First, it is easier for the domain expert to assess the adequacy of the model if larger parts of the model can be inspected easily. Second, by interacting with the model he can explore the behavior of the model for questions arising during the validation. E.g., to check whether a certain combination of faults has been modeled correctly, when the interactions of faults is scattered in the model.

Requirements Traceability Simulation and interactive visualization treat the model as a black box and merely ensure that the behavior of the model corresponds to the expectations of the domain expert in a number of situations. To ensure that the *internal structure* of the model mirrors the internal structure of the domain we annotate the model and the visualization with links to the text files containing the specification.

Requirement links trace a requirement either forward (answering "*where is this requirement realized?*") or backward (answering "*what requirement does this code realize?*"). Similarly, annotations are two-directional. A section of a rulebook links to the code in the model that implements it (i.e., is a forward trace link) and the code links to the rulebook it implements (i.e., is a backwards trace link). Links between model and specification are not enough, as the code may still implement a procedure that is a described in several places. Visualization allows us to output messages that also contain links to the specification, to connect these representation without explicitly invoking the model. These links serve two purposes: First, they enable us to track in the visualization whether the procedure is executed correctly (i.e., according to specification). Second, they ensure that the visualization, which is an additional abstraction layer/artifact (additionally to rulebooks and formal model) is integrated into the validation of the formal model.

3 ABS and the `FormbaR` Model

In this section we give a short overview over the Abstract Behavioral Specification (ABS) language [15] and the `FormbaR` [18] model of railway operations. For brevity's sake, we only introduce ABS and `FormbaR` as far as needed to explain `VisualisierbaR`; an introduction to ABS can be found in [15], an extended description of `FormbaR` in [18].

3.1 Abstract Behavioral Specification

ABS is a modeling language, developed for the modeling, simulation and analysis of distributed systems. ABS models are executable, yet it is not a programming language in this context: its foremost use is to mirror the structure of the target system, not its computational results. Its conceptual closeness to programming languages, however, allows us to demonstrate the use of requirements traces more succinctly. Most constructs of ABS are standard and its syntax is based on Java, with additional statements for concurrency. We introduce the data, communication and time models of ABS.

Data and Communication Model. ABS models data and behavior in two sublanguages. Data, and operations on the data, is modeled in a functional sublanguage based on abstract data types (ADTs). As an example, the following defines an ADT modeling the state of a (logical) signal:

```
data State = GO | HALT | SLOW | INVALID;
```

Behavior and communication is modeled in an object-oriented language on top of the functional sublanguage. The object model uses classes and interfaces and is based on Java, but all fields are object-private. Additionally, traits may be used to add methods to a class. The following class models a Zs10 auxiliary signal (end of speed limitation).

```
[Concept:"Zs10"] class Zs10(Edge track, String name) {
  uses NoSig adds NoBack adds Nameable;
  List<Trans> trigFront(Train t, Edge e){
     Information info = NoInfo;
     if( e == track )
       info = AreaEnd(-1, False, null);
     return list[Pass(info)];
  }
}
```

The class `Zs10` has two fields (`track` and `name`) and uses three traits (`NoSig`, `NoBack`, `Nameable`). It has one additional method, that transmits `AreaEnd` if the train passes the Zs10 auxiliary signal from the direction where it is visible. The types `Trans` for transmissions and `Information` for transmitted information are ADTs.

Simulation, Time and Model API. `FormbaR` uses Timed ABS [3], which extends ABS with explicit operations on time. The statement **await** `duration(x,y)` suspends the current process for at least `x` to `y` simulation time steps. At runtime, the shortest possible time is chosen. In `FormbaR`, a simulation time step corresponds to one second.

ABS can be compiled into, among other languages, Erlang and then be executed. The compiled executable contains a runtime environment in Erlang, that implements the above concurrency model and keeps track of the symbolic time — the global symbolic clock is only advanced if every object is waiting for time to pass. The clock is then advanced by the minimal time that unlocks some object.

3.2 The `FormbaR` model of Railway Operations

The `FormbaR` model is centered around the notion of *points of information flow*, which are the basic infrastructure of the formal model and are not specified by the rulebooks.

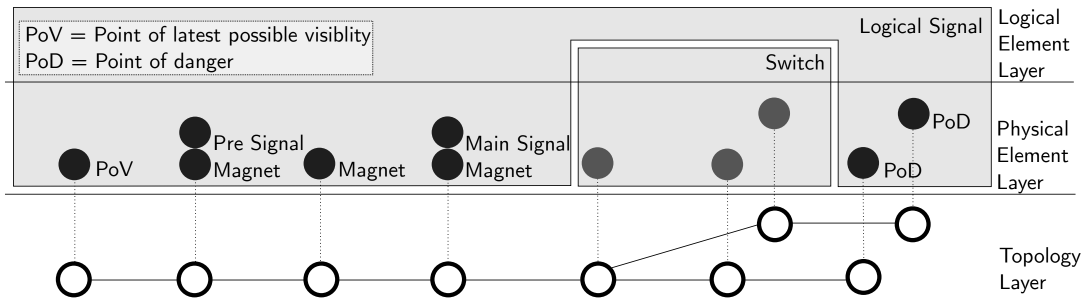


Figure 1: The lower three layers of a station entry, with a logical entry signal and a switch. (From [18])

Definition 1. *A* Point of Information Flow *(PIF) is an object at a fixed position on a track, where one of the following applies:*

- *It is an infrastructure element transmitting information to trains (e.g., balises)*
- *It is in some critical distance to an infrastructure element (e.g., the point where the presignal is seen at the latest)*
- *It is an infrastructure element, which receives information from trains*

PIFs allow one to discretize the infrastructure from an *operational* perspective, as the physical behavior of the train can be interpolated between two PIFs. `FormbaR` is also able to handle state changes of trains between two PIFs, e.g., because of orders or if the train comes to a halt before a signal. However, for the most time during simulation, the train behaviors must only be adjusted at PIFs and simulation is thus less time-consuming.

3.3 Infrastructure

The infrastructure model is based on a graph, where the nodes form the base for a four-layer model of the infrastructure. At each node of the graph, PIFs may exist and the edge has the length of the track in between two nodes.

This topological graph forms layer 1 and contains all information about physical distances. Layer 2 is a set of physical elements assigned to a node, e.g., presignals, main signals, etc. Furthermore, layer 2 is the view of the train driver on the infrastructure, who has to react to these elements. Layer 3 consists of *logical elements*. A logical element is a set of physical elements, which share state or interface to the interlocking system. This layer is the view of the train dispatcher on the infrastructure, as it is not possible to, e.g., change the state of the main signal without changing the state of the presignal. A physical element may be assigned to multiple logical elements (e.g., a presignal may belong to multiple logical signals) or none (e.g., a buffer stop).

Fig. 1 shows the entry to a train station. The black elements constitute one logical signal, the entry signal of the station: The point of visibility, where the presignal is seen at the latest, the presignal itself, the main signal, three magnets of the automatic train protection system PZB and two point of danger which are covered by the signal (e.g., axle counters). `VisualisierbaR` has basic CAD features to create and manipulate infrastructure.

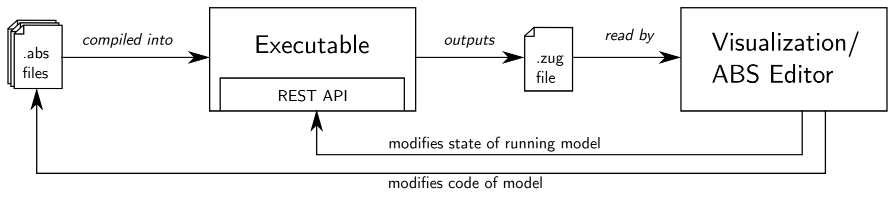


Figure 2: Overview over the components in the implementation

3.4 Communication of Stations and Trains

Stations manage a set of logical elements and communicate only with their logical elements and adjacent stations. Trains only communicate with the lowest layer of the infrastructure, the graph. The nodes relay all the transmissions from the physical elements on them to the train. Trains and stations communicate directly only via orders and only in case of faults, not during normal operations.

4 Implementation and Interaction

In the following sections we describe `VisualisierbaR`, an IDE that implements the principles of the previous section and illustrates prototyping of railway operation procedures. The interface has three components: an ABS IDE for the model, a PDF viewer for the rulebooks and a visualization. `VisualisierbaR` is designed for a multi-monitor working place, due to the space requirements of the visualization of the simulation. First, we describe the architecture of the implementation and the possibilities to interact with the model in `VisualisierbaR`.

4.1 Implementation

To start the simulation, the following workflow is implemented: The ABS compiler first generates Erlang code and then compiles Erlang to an executable file. The executable outputs a `.zug` file. This file contains a list of all `FormbaR` events that are needed for the visualization and acts as the interface between visualization and the model. It also allows to replay an execution without having the ABS code. The format is a list of events, e.g., the following is a main signal with the internal Erlang identity `TrackElements.HauptSignalImpl:<0.581.0>` changing its state to “Go”(Fahrt) at $459/8s$.

```
CH;TrackElements.HauptSignalImpl:<0.581.0>;FAHRT;459/8
```

Additionally, the executable contains a web server running a RESTful API [30] to query the object state, call methods from the outside and to limit the clock. By limiting the clock, it is possible to start the executable, read (and visualize) the output up to a certain time step and then interact with the model by calling methods via the RESTful API. Afterwards, one may resume the execution for some fixed time span by increasing the limit of the clock. `VisualisierbaR` requires that the ABS project consisting from `*.abs` files and a scenario are selected. It automatically compiles the model and starts the executable. We use two kinds of interactions from `VisualisierbaR`, which are illustrated by the two cycles in Fig. 2.

4.2 Interaction with Running Model

Interaction with the running model is the inner cycle in Fig. 2. As described, the web server allows us to interact with an already running model by invoking exposed methods. In the visualization, each physical element displays the interactions. An interaction is a method which is exposed by an `[HTTPCallable]` annotation in the interface of the class.

The RESTful API is used to read the list of exposed methods and allows to easily add methods as new interactions (at compile time). When the simulation is halted, these methods may be called to change the current state and alter the following steps in the simulation. In principle, the model must not be halted for the interactions, however `FormbaR` does not adjust simulation and wall time, to interact on a precise point in time, one thus needs to simulate up to this point, interact, and continue simulation. These interactions are available for trains and physical elements, which may however propagate the interaction to their currently responsible train station, resp. logical element.

4.3 Interaction with ABS Code

Interaction with the ABS code is the outer cycle in Fig. 2. `VisualisierbaR` contains an IDE for ABS, which allows to run the simulation, visualize it and then directly modify the ABS model. This allows visual *debugging* of railway operations, where certain situations can be modeled as the infrastructure and then directly checked whether the new (or modified) procedure behaves as intended. After modifying the code, the model is recompiled and re-executed.

To support this interaction, it is necessary to provide a way to link the visualization with two parts of the ABS code: First, the infrastructure that is currently active and second, the part of the procedure that is executed. This connection falls under contextualization, which does not only provide the context of rulebooks for the ABS model, but also the context of the ABS model for the visualization. When modifying the infrastructure, the complete initialization block of the scenario if generated anew.

5 Using `VisualisierbaR` for Validation

In this section we describe the use of `VisualisierbaR` for validation. First, we describe the visualization.

5.1 Visualization

Modes. `VisualisierbaR` can be started in three modes. If `VisualisierbaR` is started in *Visualize/Edit* mode, then the code of the ABS model and the visualization of the simulation are shown. The scenario can be edited, the simulation can be rerun and the contextual documents can be displayed. If `VisualisierbaR` is started in *Interactive* mode, then the visualization of the simulation is shown. The simulation, however, is not run yet. Instead the visualization offers the opportunity to either interact with the halted simulation (e.g., to inject faults by breaking signals or to give orders to a train) or decide to continue for a certain time frame. The ABS model and contextual documents can be displayed, but the scenario can not be edited. In these two modes, the root directory of the ABS model and the chosen scenario have to be selected. Finally, the *Replay* mode allows to visualize `.zug` files without ABS model.

A detail of the visualization of railway operations is shown in Fig. 3. This window offers, beyond visualization itself, the ability to interact with the model and can be used as an editor, that offers standard computer aided design (CAD) features: adding, editing, copying and deleting nodes, edges and physical elements. It is also possible to manage logical elements. If the ABS model is changed, the model is recompiled and the visualization shows the rerun scenario.

Window A.2 shows the details of a train or element, if one is selected. For physical elements, properties such as position and its logical element are displayed. For trains, additionally to the properties, the *v*-*t* and other graphs are shown. In interactive mode, A.2 also contains the possible interactions with the selected element or train. Optionally, a window with the list of `FormbaR`-events can be opened. There are 3 possibilities to advance the simulation:

- by manually selecting a point in time or event,
- by traversing the list of events automatically at a fixed rate events/second,
- by traversing the list of events automatically at a fixed speed. In this case the position of trains between two nodes is interpolated.

In any case it is possible to go back in time and review a part of the simulation. It is however only possible to interact with the current (i.e., newest) state.

Interaction is realized through a RESTful API [30] embedded in the ABS executable that allows to call methods from the outside and to limit the clock. The executable outputs `*.zug` files, which are read by the visualization. Reading these files allows one to replay an execution without needing the ABS code, which simplifies sharing.

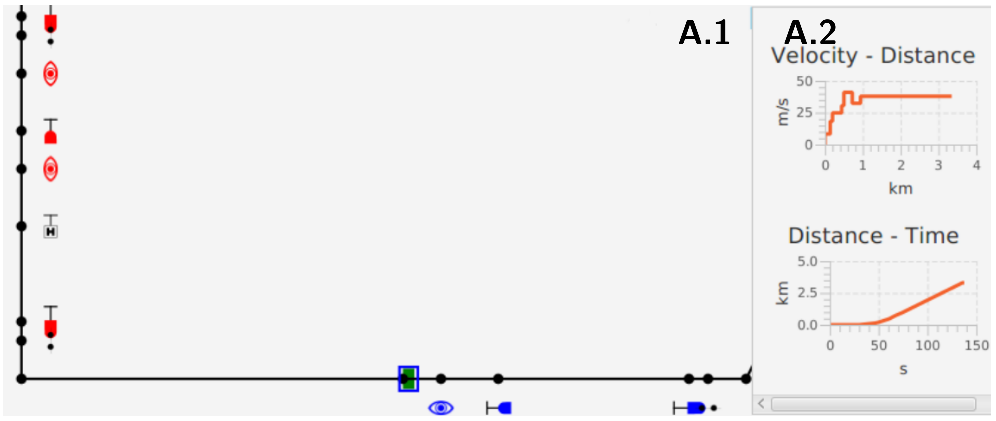


Figure 3: Window A: Visualization. The green train is about to enter the station on the bottom right, the entry signal shows “Go”. Window A.2 shows the *v*-*t* diagram and other information about the train.

5.2 Requirements Traces

The links between the components are illustrated in Fig. 4:

Documents to ABS By selecting a part of the rulebook that is linked from the code, the linking code, i.e. the annotated element, can be highlighted.

ABS to Visualization Objects are highlighted in the visualization, if their object creation site (their **new** expression) is selected in the ABS editor.

Visualization to Documents The visualization allows us to show explaining text for the simulation. These messages may contain the annotations to the rulebooks.

Visualization to ABS The object creation sites of elements selected in the visualization are highlighted in the editor.

ABS to Documents The ABS code allows us to directly link to the rulebooks from the code via annotations. The relevant part of the rulebook is then highlighted.

The first two links implement forward tracing, the others implement backward tracing. The links between ABS and documents support *n-m* relations – if a part of the document is modeled in several points of the code, a window allows to select one. The following sections illustrate the trace links in more detail.

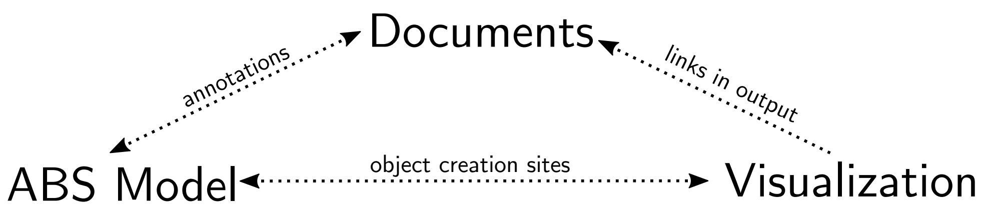


Figure 4: Structure of trace links in `VisualisierbaR`.

Visualization. The visualization provides context in two ways: (1) when selecting an element, window A highlights the statement responsible for the creation of this element. This link is used to trace an element in the visualization to a point in the model. (2) Additionally, special `MSG`-events in the `*.zug` files are supported: These events are shown as pop-ups and visualize non-visible state changes (e.g., message exchange between train and station). An `MSG`-event may contain annotations to link to contextual documents. This link traces a point in the execution of a model to a rulebook/requirement.

ABS Model. The ABS model is shown in window B in Fig. 5. ABS is a modeling language with a Java-style syntax and is presented similar to mainstream programming languages in the IDE.

In window B a file browser shows the different code files (B.1), while the main part (B.2) allows one to view and manipulate a single ABS file. B.2 offers standard IDE features like syntax highlighting or jumping to definitions.

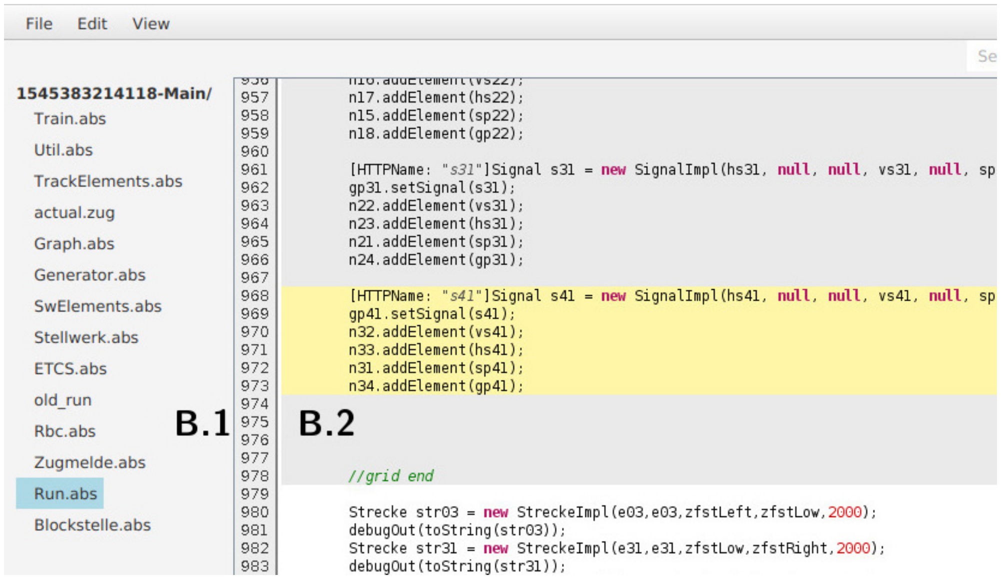
Specific to `VisualisierbaR` are two features that provide contextualization:

I In the scenario setup in file `Run.abs`, each created element can be clicked on and is then highlighted in the visualization (window A). This link is used to trace a part of the model to the visualization.

II Each class and method can be annotated with `[Document:Y]`, where `Y` is a rulebook identifier (e.g., “Ril. 408.0615”) the name of a concept or a keyword, (e.g., “Main Signal”). A click on such annotations highlights the document part in the document window (window C) marked with this identifier (in case of `[Document:Y]` or a window that lists all document parts responsible for the keyword (in case of `[Concept:X]`). This link is used to trace a part of the model to the rulebook/requirement. If a concept is linked to multiple parts of the rulebooks, the user can select one of them. The mapping between annotated concept and rulebook sections are manually managed in a `.csv` file that allows *n-m* relations and is a variant of a requirement matrix between code and rulebooks.

The IDE offers a way to modify the code and recompile. After recompilation, window B.2 is split into two panes, where the left shows the current code and the right the code of the model before compilation to simplify tracking of changes.

Documents. Window C is a PDF viewer which highlights parts of the document if referenced from the other components. It provides context in two ways. When a part of the document is clicked on and this part of the document is referenced from the model, a list of all annotation referring to it is shown. This list then highlights the annotation in window B.2. This link is used to trace a requirement to the model.



```
        n16.addElement(vs22);
        n17.addElement(hs22);
        n15.addElement(sp22);
        n18.addElement(gp22);

        [HTTPName: "s31"]Signal s31 = new SignalImpl(hs31, null, null, vs31, null, sp
        gp31.setSignal(s31);
        n22.addElement(vs31);
        n23.addElement(hs31);
        n21.addElement(sp31);
        n24.addElement(gp31);

        [HTTPName: "s41"]Signal s41 = new SignalImpl(hs41, null, null, vs41, null, sp
        gp41.setSignal(s41);
        n32.addElement(vs41);
        n33.addElement(hs41);
        n31.addElement(sp41);
        n34.addElement(gp41);




        //grid end

        Strecke str03 = new StreckeImpl(e03,e03,zfstLeft,zfstLow,2000);
        debugOut(toString(str03));
        Strecke str31 = new StreckeImpl(e31,e31,zfstLow,zfstRight,2000);
        debugOut(toString(str31));
```

Figure 5: Excerpt of Window B: ABS editor. The different backgrounds visualize infrastructure selected in window A.

Fig. 6 shows the PDF viewer. The bars on the left mark the referenced parts of the documents. The window on the right is displayed when a part of the document is clicked on and lists all references in the ABS code to it. A list of all references can be shown in the ABS editor.

6 Validation Case Studies for ATO and Prototyping

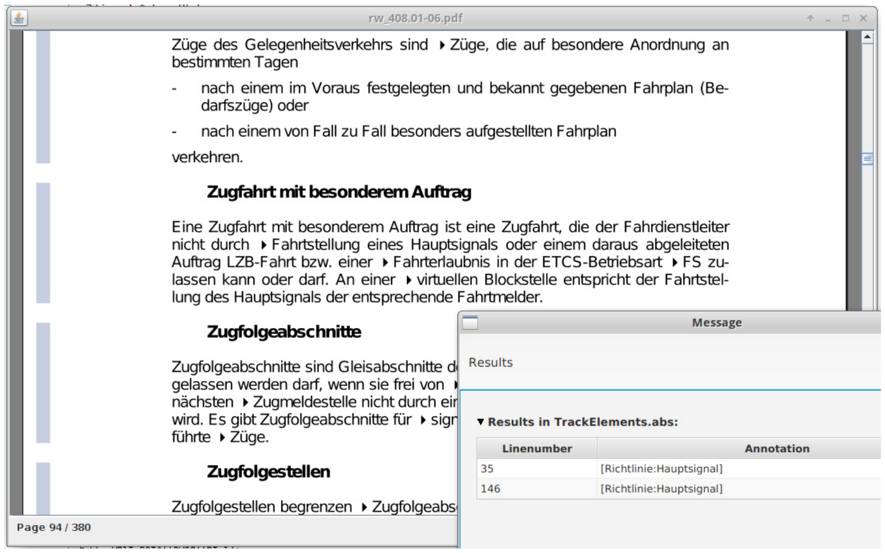
We give two examples how `VisualisierbaR` can be used in the workflow of rulebook authors. The first example is from the on-going development of new procedures for autonomous train operations [31], where `VisualisierbaR` was applied to check that the new rules correctly interact with the old rules for non-autonomous train operations. The second example models the change of a rule. Followine real world changes [27], we model the effects on delays, depending on whether the first train after a fault has occurred drives on sight or not.

6.1 Validating ATO Procedures

We give an example where `VisualisierbaR` is used in the current development of a system to handle faults during autonomous train operations (ATO) with grade of autonomy (GoA) 4 [31] to analyze how the additional checks needed for ATO interact with the operational rules. In this case, the requirements are the developed procedures for ATO GoA 4 and their correct interactions with the original rules.

The investigated scenario was an obstacle in front of a signal, where the autonomous train adheres to the rules specific to GoA 4 (detecting the obstacle and waiting for it to disappear) as well as to the rules for general operations (responding to the signal). The model has to realize both rulebooks.

The decisions necessary for ATO are annotated with links to the ATO documentation and the already existing model for general operations with links to the rulebooks. To connect the model with the visualization, we added messages and used the following scenario: A train is driving towards a signal and an



Züge des Gelegenheitsverkehrs sind ▸Züge, die auf besondere Anordnung an bestimmten Tagen

- nach einem im Voraus festgelegten und bekannt gegebenen Fahrplan (Bedarfszüge) oder
- nach einem von Fall zu Fall besonders aufgestellten Fahrplan

verkehren.

Zugfahrt mit besonderem Auftrag

Eine Zugfahrt mit besonderem Auftrag ist eine Zugfahrt, die der Fahrdienstleiter nicht durch ▸Fahrtstellung eines Hauptsignals oder einem daraus abgeleiteten Auftrag LZB-Fahrt bzw. einer ▸Fahrterlaubnis in der ETCS-Betriebsart ▸FS zulassen kann oder darf. An einer ▸virtuellen Blockstelle entspricht der Fahrtstellung des Hauptsignals der entsprechende Fahrtmelder.

Zugfolgeabschnitte

Zugfolgeabschnitte sind Gleisabschnitte d
gelassen werden darf, wenn sie frei von ▸
nächsten ▸Zugmeldestelle nicht durch ei
wird. Es gibt Zugfolgeabschnitte für ▸sig
führte ▸Züge.

Zugfolgestellen

Zugfolgestellen begrenzen ▸Zugfolgeabs

▾ **Results in TrackElements.abs:**

Linenumber	Annotation
35	[Richtlinie:Hauptsignal]
146	[Richtlinie:Hauptsignal]

Figure 6: Window C: PDF viewer

obstacle, e.g., a cow, is directly before the signal. The signal signals “Halt”. ATO GoA 4 adds the rule that a train halts before any detected obstacle.

Trace links are able to enhance validation through simulation by tracing certain execution steps back to the original procedures. Simulating the scenario shows that the train detects the cow and halts until the cow leaves the track, even if the signal switches to “Go”. Similarly, if the cow leaves and the signal is still signaling “Halt”, the train waits. At each point, the simulation displays the decisions of the ATO algorithms, e.g., if the obstacle is not detected anymore it is displayed why the system decided to halt.

This allows us to check that rules for ATO GoA 4 do not override rules for normal train driving or otherwise interfere with them. From a development process view, the simulation itself is a behavioral test that links its output with the requirements and the annotations are links for requirements traces.

6.2 Prototyping Rule Changes

To reason about the effects of a proposed rule change, `VisualisierbaR` was first used to model a variant of the infrastructure in the west branch of the Frankfurt City Tunnel. This branch has a length of 4.7km and is the main part of the Rhein-Main S-Bahn – eight lines pass through it, with intervals below five minutes. Its high usage makes it representative of how rules affect operations in networks with high occupancy rate and short distances between signals. We only model one direction (from Hauptbahnhof to Südbahnhof) without the branch-off point Schlachthof, which is sufficient for the analyzed rule change.

On the infrastructure two trains with a 5 minutes interval are simulated, both with a maximal velocity of 60km/h. We model the following scenario: the main signal on the track between Ostendstraße and Lokalbahnhof[3] has a fault that is local to this signal (e.g., a broken bulb). To sustain operations, the train dispatcher gives an order to depart (equivalent to a Zs1 auxiliary signal) nonetheless.

Old rule. The train dispatcher must not order to drive on sight, thus the first train can still drive the full 60km/h. In this case the second train, which departs at $t = 300s$, arrives in the final station at $t = 1027s$.

[3]This block was chosen because it has the shortest sight distance and requires the slowest speeds when driving on sight.

New rule. Now, the train must first drive on sight, which is walking speed in tunnels (6 km/h). In this case the second train (which is not effected by this and may drive 60km/h) arrives at the final station at $t = 1487s$.

The delay, over 7 minutes[4] is specific to this infrastructure and time table, yet gives an estimate which helps the developers to assess the impact of a rule change. Another example to examine rule changes with `FormbaR` is discussed in [18]. `VisualisierbaR` is an improvement over the previous ad-hoc visualization, as it allows to assess the relevant information faster by showing the *v-t* graph.

This application of `VisualisierbaR` was presented to the rulebook authors of DB Netz responsible for this rule, who deemed the visualization and the trace links as helpful.

7 Conclusion and Future Research

We presented `VisualisierbaR` and have shown how it can be integrated into the processes for developing railway operation procedures. It illustrates how model validation can be supported by integrating requirement traces and how these traces increase the usefulness of tests and visualization. It extends our previous work on modeling these procedures by giving an interface that does not require the user to learn ABS to use the model, but gives him the possibility for deeper manipulation with ABS if necessary. `VisualisierbaR` extends the use cases of formal tools in railway engineering from support for implementation and planning [8, 12] to the development of new procedures by using an ABS model to prototype ATO procedures. Beyond railway engineering, we addressed the challenge to use technical documents as requirements for validation, which are not designed to be used as requirements and are not modifiable by the modeler.

`VisualisierbaR` is available under `formbar.raillab.de/visbar` with limited annotations, as most rulebooks and the rules for the above ATO case study are not public. A video demonstrating the usage of `VisualisierbaR` is available under `https://figshare.com/s/71f1c2e7252bfd032f57`.

It is often observed that formal models offer a benefit for the designer, even without analyzing formal properties, as it forces to clarify all ambiguities. Thus, formal modeling languages must not only be easy to analyze, but also easy to validate and easy to integrate into existing development processes. Yet, validation of formal system models and its place in development processes remains a challenging domain. For future work, we are not only interested in the integration of validation of formal models into a development process, but the development process of formal models and digital twins itself. In particular, we are interested in the following:

- Requirement trace generation for formal modeling.
- Integration of conceptual modeling [26] into formal model validation by connecting requirements and formal model with a domain ontology[5].

The overarching questions are (1) how to design formal modeling languages (and IDEs for them) which are not only easily usable and analyzable, but also easy to validate and (2) how to use traceability in verified formal models for certification. We propose that automatic generation of traces would not only vastly simplify validation, but also be a step towards a wider acceptance of formal proofs for certification.

[4]This is longer than the delay caused purely by waiting (3 minutes) for the first train to arrive, but still realistic. The additional 4 minutes are caused by a non-optimal train dispatching in our model. However, the duties of the train dispatcher to document the situation and give written orders in case of faults accounts for this.

[5]Conceptual modeling faces similar problems with validation, but is more abstract in the information it captures and relies more on implicit knowledge, than specifications, designs or a concrete existing system.

Related Work Luteberget et al. [22] use traces to link errors raised during verification to the responsible part of the model and the original document. These traces roughly resemble the annotations in messages generated by `VisualisierbaR` during simulation, but are not used to validate the model itself. Ferrari et al. [10] investigated the requirements of railway engineering projects from a natural language processing perspective. Concerning the connection of conceptual and formal modeling, Kharlamov et al. [20] propose to use ontologies to develop digital twins, but not for validation.

Fischer and Dghaym [11] use acceptance tests to validate a formal model of Hybrid ETCS L3 segments. Contrary to requirement traces and interactive visualization their approach requires *fully formalized* test cases of observable behavior of the model. This approach is not only subsumed by simulation – as discussed, it also does not lower the cognitive burden of validation, as these test cases are a *formal behavioral model themselves*.

Integration of multiple aspects is common for programming languages in mainstream IDEs, but development environments based on formal methods focus mostly only on the formal model and its verification, e.g., by an interface to the proof system. E.g., the B-OVADO [12] tool for the PERF [2] approach, offers a toolbox for data validation tasks that integrates B as a language to specify data. The `Sphinx` tool [23], which integrates verification and modeling tools for model-based engineering of hybrid systems, is the only approach that uses formal methods for coordinating multiple components for development. It also provides a way to connect to documentation in a special UML profile and is specific to differential dynamic logic. Ladenberger [21] also investigated interaction for validation of B-models.

Future Work Beyond further research in the connection to requirements engineering sketched above, we plan (1) to enable statistical analyses, such as expected lost units [19] after a rule change, in a representative network and (2) to integrate our verification approach [16] to use it for certifications. We also plan to investigate how, analogous to Domain Specific Languages, Domain Specific IDEs, can be used to integrate formal methods into other domains.

Acknowledgments This work is supported by the `FormbaR` project, part of AG Signalling/DB RailLab. We thank Heike Villioth-Ebert, Armin Krieger, Matthias Kopitzki and Bilal Üyümez for their feedback.

References

[1] Y. Barlas. Formal aspects of model validity and validation in system dynamics. *System Dynamics Review - SYST DYNAM REV*, 12, 09 1996.

[2] N. Benaissa, D. Bonvoisin, A. Feliachi, and J. Ordioni. The PERF approach for formal verification. In T. Lecomte, R. Pinger, and A. Romanovsky, editors, *RSSRail 2016 proc.*, pages 203–214, Cham, 2016. Springer International Publishing.

[3] J. Bjørk, F. S. de Boer, E. B. Johnsen, R. Schlatte, and S. L. Tapia Tarifa. User-defined schedulers for real-time concurrent objects. *ISSE*, 9(1):29–43, 2013.

[4] M. Broy, K. Havelund, R. Kumar, and B. Steffen. Towards a unified view of modeling and programming (track introduction). In T. Margaria and B. Steffen, editors, *ISoLA*, pages 3–21. Springer, 2018.

[5] CENELEC. DIN EN 50128:2011, Railway applications – Communication, Signalling and Processing Signals, 2011.

[6] DB Netz AG, Frankfurt, Germany. Richtlinie 408, Fahrdienstvorschrift, 2017.

[7] DB Netz AG, Frankfurt, Germany. Richtlinie 819, LST-Anlagen planen, 2017.

[8] S. Dillmann and R. Hähnle. Automated planning of ETCS tracks. In *RSSRail*, volume 11495 of *Lecture Notes in Computer Science*, pages 79–90. Springer, 2019.

[9] Eisenbahnbundesamt (Federal Railway Authority). Eisenbahn-bau- und betriebsordnung, 2017. April 2017: https://www.gesetze-im-internet.de/ebo/index.html.

[10] A. Ferrari, G. Gori, B. Rosadini, I. Trotta, S. Bacherini, A. Fantechi, and S. Gnesi. Detecting requirements defects with NLP patterns: an industrial experience in the railway domain. *Empirical Software Engineering*, 23(6):3684–3733, 2018.

[11] T. Fischer and D. Dghaym. Formal model validation through acceptance tests. In *RSSRail 2019*, volume 11495 of *Lecture Notes in Computer Science*, pages 159–169. Springer, 2019.

[12] M. Fredj, S. Leger, A. Feliachi, and J. Ordioni. OVADO - enhancing data validation for safety-critical railway systems. In A. Fantechi, T. Lecomte, and A. B. Romanovsky, editors, *RSSRail 2017 proc.*, volume 10598 of *LNCS*, pages 87–98. Springer, 2017.

[13] C. Gomes, C. Thule, D. Broman, P. G. Larsen, and H. Vangheluwe. Co-simulation: State of the art. *CoRR*, abs/1702.00686, 2017.

[14] IEEE. IEEE guide for software requirements specifications. *IEEE Std 830-1998*, 1998.

[15] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In *FMCO*, volume 6957 of *LNCS*. Springer, 2010.

[16] E. Kamburjan and R. Hähnle. Deductive verification of railway operations. In *RSSRail 2017*, volume 10598 of *Lecture Notes in Computer Science*, pages 131–147. Springer, 2017.

[17] E. Kamburjan and R. Hähnle. Prototyping formal system models with active objects. In *Interaction and Concurrency Experience*, volume 279 of *EPTCS*, pages 52–67. Open Publishing Association, 2018.

[18] E. Kamburjan, R. Hähnle, and S. Schön. Formal modeling and analysis of railway operations with active objects. *Science of Computer Programming*, 166:167 – 193, 2018.

[19] F. R. Kämmerer. Entwicklung eines Kennzahlensystems für Effektivität des Bahnbetriebs bei Abweichungen vom Regelbetrieb. Master's thesis, Technische Universität Darmstadt, 2017.

[20] E. Kharlamov, F. Martin-Recuerda, B. Perry, D. Cameron, R. Fjellheim, and A. Waaler. Towards semantically enhanced digital twins. In *2018 IEEE International Conference on Big Data*, pages 4189–4193, 2018.

[21] L. Ladenberger. *Rapid Creation of Interactive Formal Prototypes for Validating Safety-Critical Systems*. PhD thesis, University of Düsseldorf, Germany, 2017.

[22] B. Luteberget, J. J. Camilleri, C. Johansen, and G. Schneider. Participatory verification of railway infrastructure by representing regulations in RailCNL. In *SEFM*, volume 10469 of *LNCS*. Springer, 2017.

[23] S. Mitsch, G. O. Passmore, and A. Platzer. Collaborative verification-driven engineering of hybrid systems. *Mathematics in Computer Science*, 8(1):71–97, 2014.

[24] C. Newcombe. Why Amazon chose TLA+. In Y. Ait Ameur and K.-D. Schewe, editors, *Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 25–39, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[25] D. North. Introducing BDD, 2006. http://dannorth.net/introducing-bdd/.

[26] A. Olivé. *Conceptual Modeling of Information Systems*. Springer-Verlag, Berlin, Heidelberg, 2007.

[27] J. Pachl. Das Ersatzsignal – ein deutscher Sonderweg? *Deine Bahn*, 3, 2018. In German.

[28] R. Rosen, G. von Wichert, G. Lo, and K. D. Bettenhausen. About the importance of autonomy and digital twins for the future of manufacturing. *IFAC-PapersOnLine*, 48(3):567 – 572, 2015.

[29] RTCA Inc, EUROCAE. DO-178C, 2012.

[30] R. Schlatte, E. B. Johnsen, J. Mauro, S. L. Tapia Tarifa, and I. C. Yu. *Release the Beasts: When Formal Methods Meet Real World Data*, pages 107–121. Springer International Publishing, Cham, 2018.

[31] B. Üyümez. Modellierung des Steuerungsprozesses der Rückfallebenen als Grundlage für die Automatisierung. *Eisenbahntechnische Rundschau*, 4 2018. In German.

Author Index

Ahrendt, Wolfgang	1
Bessai, Jan	2
Brunel, Julien	66
Chemouil, David	66
Cousineau, Denis	8
Cunha, Alcino	66
Inoue, Hiroaki	8
Kamburjan, Eduard	72
Kuppe, Markus Alexander	51
Macedo, Nuno	66
Masci, Paolo	36
Mentré, David	8
Mosses, Peter	30
Munoz, Cesar	36
Roidl, Moritz	2
Stromberg, Jonas	72
Tuong, Frédéric	14
Vasileva, Anna	2
Wolff, Burkhart	14