F-IDE 2016 Preliminary Proceedings

3rd workshop on Formal Integrated Development Environments

Satellite event of FM 2016
Limassol, Cyprus

Preface

F-IDE 2016 is the third Formal Integrated Development Environment Workshop (F-IDE 2016) held in Limassol, Cyprus, on 8 November, 2016 as a satellite workshop of the FM conference.

High levels of safety, security and also privacy standards require the use of formal methods to specify and develop compliant software (sub)systems. Any standard comes with an assessment process, which requires a complete documentation of the application in order to ease the justification of design choices and the review of code and proofs. An F-IDE dedicated to such developments should comply with several requirements. The first one is to associate a logical theory with a programming language, in a way that facilitates the tightly coupled handling of specification properties and program constructs. The second one is to offer a language/environment simple enough to be usable by most developers, even if they are not fully acquainted with higher-order logics or set theory, in particular by making development of proofs as easy as possible. The third one is to offer automated management of application documentation. It may also be expected that developments done with such an F-IDE are reusable and modular. Moreover, tools for testing and static analysis may be embedded in this F-IDE, to help address most steps of the assessment process. The workshop is a forum of exchange on different features related to F-IDEs.

We solicited several kinds of contributions: research papers providing new concepts and results, position papers and research perspectives, experience reports, tool presentations. The current edition is a one-day workshop where eight communications are given, offering a large variety of approaches, techniques and tools. Some of the presentations took the form of a tool demonstration. Each submission was reviewed by three reviewers.

We have the honor to welcome Professor Kim G. Larsen from Aalborg University and he will give a keynote entitled *Verification, Optimization, Performance Analysis and Synthesis of Cyber-Physical Systems*.

We would like to thank the PC members for doing such a great job in writing high-quality reviews and participating in the electronic PC discussion.

We would like to thank all authors who submitted their work to F-IDE 2016. We are grateful to the FM Organisation Committee, which has accepted to host our workshop. The logistics of our job as Program Chairs were facilitated by the EasyChair system and we thank the editors of *Electronic Proceedings in Theoretical Computer Science* who accepted to publish the papers.

Catherine Dubois
Paolo Masci
Dominique Méry
F-IDE 2016 Program Chairs

Verification, Optimization, Performance Analysis and Synthesis of Cyber-Physical Systems

Kim G. Larsen
Department of Computer Science, Aalborg University, Denmark
kgl@cs.aau.dk

Timed automata and games, priced timed automata and energy automata have emerged as useful formalisms for modeling real-time and energy-aware systems as found in several embedded and cyber-physical systems. In this talk we will survey how the various component of the UPPAAL tool-suite over a 20 year period has been developed to support various type of analysis of these formalisms.

This includes the classical usage of UPPAAL as an efficient model checker of hard real time constraints of timed automata models, but also the branch UPPAAL CORA which have been extensively used to find optimal solutions to time-constrained scheduling problems. More ambitiously, UPPAAL TIGA allows for automatic synthesis of strategies and subsequent executable control programs for safety and reachability objectives. Most recently the branch UPPAAL SMC offers a highly scalable statistical model checking engine supporting performance analysis of stochastic hybrid automata, and the branch UPPAAL STRATEGO which supports synthesis (using machine learning) of near-optimal strategies for stochastic priced timed games. The keynote will review the various branches of UPPAAL and highlight their concerted applications to a selection of real-time and cyber-physical examples.

To appear in EPTCS.

The KeYmaera X Proof IDE

Concepts on Usability in Hybrid Systems Theorem Proving

Stefan Mitsch
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA
smitsch@cs.cmu.edu

André Platzer
Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, USA
aplatzer@cs.cmu.edu

Hybrid systems verification is quite important for developing correct controllers for physical systems, but is also challenging. Verification engineers, thus, need to be empowered with ways of guiding hybrid systems verification while receiving as much help from automation as possible. Due to undecidability, verification tools need sufficient means for intervening during the verification and need to allow verification engineers to provide system design insights.

This paper presents the design ideas behind the user interface for the hybrid systems theorem prover KeYmaera X. We discuss how they make it easier to prove hybrid systems as well as help learn how to conduct proofs in the first place. Unsurprisingly, the most difficult user interface challenges come from the desire to integrate automation and human guidance. We also share thoughts how the success of such a user interface design could be evaluated and anecdotal observations about it.

1 Introduction

Cyber-physical systems such as cars, aircraft, and robots combine computation and physics, and provide exceedingly interesting and important verification challenges. KeYmaera X [11] is a theorem prover for *hybrid systems*, i. e., systems with interacting discrete and continuous dynamics, which arise in virtually all mathematical models of cyber-physical systems.[1] It implements *differential dynamic logic* (d$\mathscr{L}$ [23, 24, 27]) for *hybrid programs*, a program notation for hybrid systems. Differential dynamic logic provides compositional techniques for proving properties about hybrid systems. Despite the substantial advances in automation, user input is often quite important, since hybrid systems verification is not semidecidable [23]. Human insight is needed most notably for finding invariants for loop induction and finding differential invariants for unsolvable differential equations [24]. But even some of the perfectly decidable questions in hybrid systems verification are intractable in practice, such as the final step of checking the validity of formulas in real arithmetic[2] in a d$\mathscr{L}$ proof.

To overcome those verification challenges, KeYmaera X combines automation and interaction capabilities to enable users to verify their applications even if they are still out of reach for state-of-the-art automation techniques. The central question in usability, thus, is how interaction and automation can jointly solve verification challenges. For isolated strategic aspects of the proofs, such user guidance is easily separated, for example when using system insights to provide loop invariants and differential invariants. Other aspects of human insights are more invasive, such as picking and transforming relevant

[1] KeYmaera X is available at http://keymaeraX.org/

[2] Decision procedures are doubly exponential in the number of quantifier alternations [9], and practical implementations doubly exponential in the number of variables.

EPTCS ??, 20??, pp. 1–15, doi:10.4204/EPTCS.??.??

formulas to make intractable arithmetic tractable. Users have to link the logical level of proving (e. g., the conjecture, available proof rules and axioms) to the abstract interaction level (e. g., visualization of formulas in a sequent) and decide about concrete interaction steps (e. g., where to click, what to type) [3]. Hence, going back and forth between automated proof tactics and user guidance poses several challenges:

- Users need to be provided with a way of understanding the proof state produced by automated tactics. What are the open proof goals? How are these goals related to the proof of the conclusion? And why did the automated tactic stop making progress?
- Users need to be provided with efficient ways of understanding the options for making progress. What tactics are available and where can they be applied? What input is needed? And what interactions provide genuinely new insights into a proof that the automation would not have tried?
- Users need efficient tools for executing proof steps. How to provide gradual progress for novices? How to let experienced users operate with minimal input? How to reuse proof steps across similar goals? And how to generalize a specific tactic script into a proof search procedure for similar problems?

Users may additionally benefit from picking an appropriate interaction paradigms, such as *proof as programming*, *proof by pointing*, or *proof by structure editing* [3].

This paper discusses how KeYmaera X addresses these challenges through its web-based user interface. The complexity of larger hybrid systems verification challenges require that users be granted significant control over how a proof is conducted and what heuristics are applied for proof search. To this end, KeYmaera X separates user interaction and proof search from the actual proof steps in the prover kernel to ensure soundness while providing reasoning flexibility [11]. All proof steps follow from a small set of axioms by uniform substitution [26, 27]. This paper further introduces an evaluation concept to determine how effectively alternative user interaction concepts implemented in KeYmaera X address these challenges, as well as whether or not the combination of these concepts compare favorably to our previous hybrid systems theorem prover KeYmaera [28]. The experiments are yet to be conducted; our reports on the effectiveness of the user interaction concept remain anecdotal based on feedback from external users and students.

2 Preliminaries: Differential Dynamic Logic

This section recalls the syntax and semantics of differential dynamic logic by example of the motion of a person on an escalator. This example serves for illustrating prover interaction throughout the paper.

Syntax and semantics by example. Suppose a person is standing on an escalator, which moves upwards with non-negative speed $v\geq 0$, so the person's vertical position x follows the differential equation $x' = v$. If the person is not at the bottom-most step ($?x>1$), she may step down one step ($x := x-1$) or may just continue moving upwards; since she may or may not step down, even if allowed, we use a non-deterministic choice ($\cup$). We want to prove that the person never falls off the bottom end of the escalator ($x\geq 0$) when stepping down and moving upwards are repeated arbitrarily often (modeled with the repetition operator *).

$$\underbrace{x \geq 2 \wedge v \geq 0}_{\text{initial conditions}} \rightarrow [\underbrace{\big((?x > 1; x := x-1) \cup x' = v\big)^*}_{\text{hybrid program}}] \underbrace{x \geq 0}_{\text{safety condition}} \tag{1}$$

The formula (1) captures this example as a safety property in $\mathrm{d}\mathscr{L}$. Suppose, the person is initially at some position $x \geq 2$ when the escalator turns on. From any state satisfying the initial conditions

$x \geq 2 \land v \geq 0$, all runs of the hybrid program $((?x > 1; x := x - 1) \cup x' = v)^*$ reach only states that satisfy the safety condition $x \geq 0$. More detailed examples on modeling hybrid systems are in [29].

Manual proof in dℒ. Formulas in dℒ, such as the simple example (1), can be proved with the dℒ proof calculus. Proofs in dℒ are sequent proofs: a sequent has the shape $\Gamma \vdash \Delta$, where we assume all formulas Γ in the antecedent (to the left of the turnstile $\vdash$) to show any of the formulas Δ in the succedent (right of the turnstile). The sequent notation works from the desired conclusion at the bottom toward the resulting subgoals at the top. While sometimes surprising for novices, this notation emphasizes how the truth of the conclusions follows from the truth of their respective premises top-down. This notation also highlights the current subquestion at the very top of the deduction. Steps in the sequent proof are visualized through a horizontal line, which separates the conclusion at the bottom from the premises at the top. The name of the deduction step is annotated to the left of the horizontal line. For example, the proof rule $[\cup]$ says that to conclude safety of a non-deterministic choice of programs $\alpha \cup \beta$ (below the bar) it suffices to prove safety of both α and β individually (above bar).

$$[\cup]\frac{\phi \vdash [\alpha]\psi \land [\beta]\psi}{\phi \vdash [\alpha \cup \beta]\psi}$$

Manual proof example. Starting from the formula (1) at the very bottom of the deduction, we develop a safety proof as follows. As first step, the rule $\rightarrow$ R moves the assumptions from the left-hand side of an implication into the antecedent. Next $\land$L splits the conjunction $x \geq 2 \land v \geq 0$ into individual facts (i. e., we get to assume both, $x \geq 2$ and $v \geq 0$ individually). Then, we use loop induction with the invariant $x > 0$. We have to show three cases: the loop invariant must hold initially (base case $x \geq 2 \rightarrow x > 0$), it must be strong enough to entail safety (use case $x > 0 \rightarrow x \geq 0$), and it must be preserved by the loop body (induction step $[(?x > 1; x := x - 1) \cup x' = v]x > 0$).

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{*}{x>0, v\geq 0, x>1 \vdash x-1>0}\text{QE}}{x>0, v\geq 0, x>1 \vdash [x:=x-1]x>0}[:=]}{x>0, v\geq 0 \vdash [?x>1][x:=x-1]x>0}[?],\rightarrow\text{R} \quad \dfrac{*}{x_0>0, v\geq 0 \vdash \forall t\geq 0 \forall 0\leq s\leq t\ (x = x_0 + vs \rightarrow x>0)}\text{QE}}{x>0, v\geq 0 \vdash [?x>1; x:=x-1]x>0}[;] \quad \dfrac{}{x>0, v\geq 0 \vdash [x'=v]x>0}\text{ODE}}{x>0, v\geq 0 \vdash [?x>1; x:=x-1]x>0 \land [x'=v]x>0}\land\text{R}}{\dots}[\cup]$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{(base case)}\ *}{x\geq 2, v\geq 0 \vdash x>0}\text{QE} \quad \dfrac{\text{(use case)}\ *}{x>0 \vdash x\geq 0}\text{QE} \quad \dfrac{}{x>0, v\geq 0 \vdash [(?x>1; x:=x-1)\cup x'=v)]x>0}\text{(induction step)}}{x\geq 2, v\geq 0 \vdash [((?x>1; x:=x-1)\cup x'=v)^*]x\geq 0}\text{loop}}{x\geq 2 \land v\geq 0 \vdash [((?x>1; x:=x-1)\cup x'=v)^*]x\geq 0}\land\text{L}}{\vdash x\geq 2 \land v\geq 0 \rightarrow [((?x>1; x:=x-1)\cup x'=v)^*]x\geq 0}\rightarrow\text{R}$$

Here, the base case and use case can be shown easily using quantifier elimination QE. The induction step proceeds using the proof rule $[\cup]$ for non-deterministic choice that we saw above, followed by $\land$R to split the induction step proof into two branches. On the first branch, we first turn the sequential composition (;) into nested boxes ($[?x > 1][x := x - 1]x > 0$), and then use the test condition ($x > 1$) as an additional assumption using rule $[?]$ followed by $\rightarrow$ R. We show safety of the assignment $x := x - 1$ using quantifier elimination QE to close the proof. On the second branch, we show safety of the differential equation $x' = v$ using the proof rule ODE followed by QE. □

3 KeYmaera X Proof Automation

As a basis for understanding how KeYmaera X searches for proofs and where and why it asks for user guidance, this section gives a high-level explanation of KeYmaera X tactics.

KeYmaera X automates the tedious task of proving steps that follow unambiguously from the structure of the conjecture. It further provides (heuristic) tactics to generate and explore invariant candidates for loop induction and differential equations. One might imagine KeYmaera X to try to solve differential equations and use the solution to guide a differential invariant proof, before it resorts to more involved differential invariant proofs. KeYmaera X provides proof tactics for propositional reasoning, reasoning about hybrid programs, and closing (arithmetic) proof goals, which are combined into a fully automated proof search tactic. For example, with a loop invariant candidate annotated in the KeYmaera X input file, the running example in this paper proves fully automated.

Propositional reasoning prop and program unfolding unfold of hybrid programs follows along propositional sequent rules and the axioms of $\mathrm{d}\mathscr{L}$. These tactics successively match on the shape of a formula to transform it into simpler parts, before the tactics descend into the resulting parts. Program unfolding focuses on the decidable fragment of reasoning about hybrid programs: it stops and asks for user guidance when it encounters loops or ODEs. The following proof snippet applies unfold to just the induction step of the escalator proof.

Induction step by unfold. The tactic unfold applies hybrid program axioms and splits conjunctions in the succedent into proof branches, but stops when it encounters loops or ODEs. The resulting two subgoals correspond to the two (logically unfolded) paths through our running example: we have to show safety of the discrete assignment $x := x-1$ as well as of the differential equation $x' = v$.

$$\text{unfold}\ \frac{x{>}0, v{\geq}0, x{>}1 \vdash x-1 \geq 0 \qquad x{>}0, v{\geq}0 \vdash [x'=v]x{>}0}{x{>}0, v{\geq}0 \vdash [?x{>}1; x := x-1 \cup x'=v]x{>}0}$$

□

KeYmaera X ships with proof search tactic auto, which combines propositional reasoning with program unfolding, loop invariant exploration, certain automated proof techniques for differential equations, and proof closing by quantifier elimination. Even though the auto tactic finds proofs for important classes of hybrid systems automatically, it still may stop exploration and ask for user guidance in complicated cases (e. g., when none of the explored differential invariants helps closing the proof).

4 KeYmaera X User Interaction

When the automated tactics shipped with KeYmaera X fail to find a proof (due to a wrong model, missing loop or differential invariants, or intractable arithmetic), user interaction is needed to improve the model and make progress with the proof. This section introduces the KeYmaera X user interaction concepts and their implementation in a graphical web-based user interface. The user interface of KeYmaera X is based on these principles and hypotheses:

Familiarity Prover user interfaces benefit from a familiar look&feel that resembles how proofs are conducted in theoretical developments and that are compatible with the way that proof rules are presented.

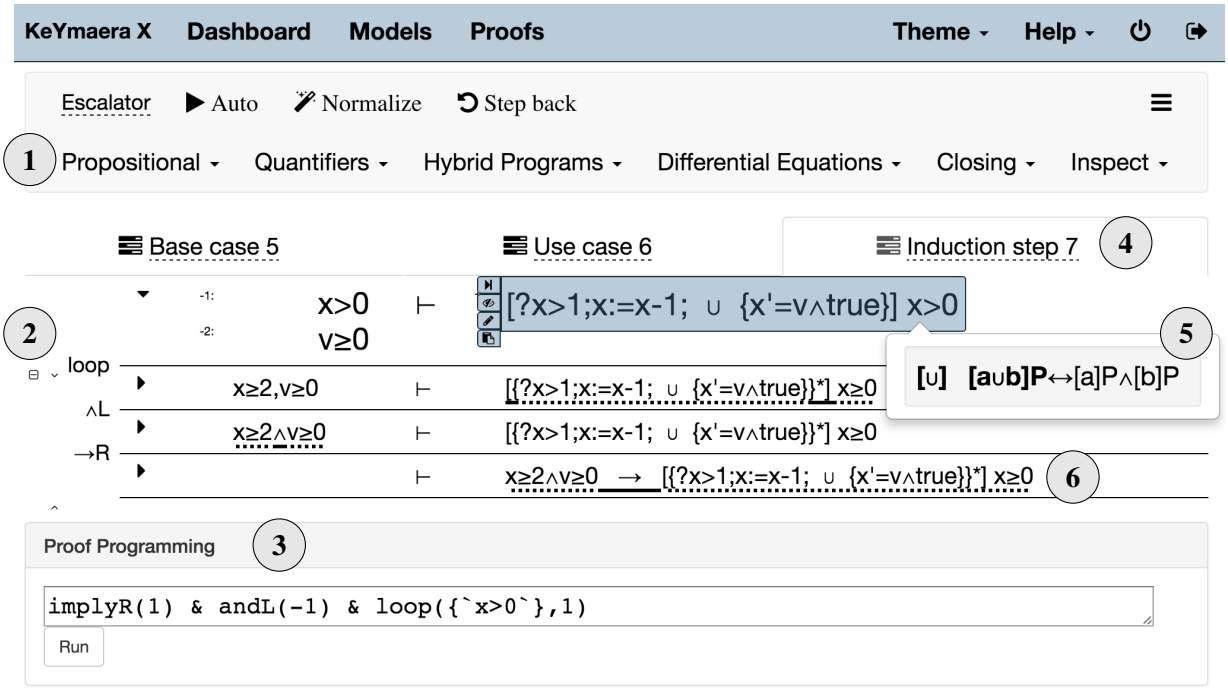


Figure 1: Screenshot of KeYmaera X with annotated user interface elements: ① proof-by-search; ② sequent view; ③ proof programming and tactic extraction; ④ proof branch; ⑤ tactic suggestion; ⑥ tactic step highlighting.

Traceability Sophisticated verification challenges, especially in hybrid systems, need a way of mixing automation with user guidance in a way that the user can trace and understand the respective remaining questions.

Tutoring Interactive proof-by-pointing at formulas and terms are an efficient way of learning how to conduct proofs and help internalizing reasoning principles by observation. Tactic recording is an efficient way of learning how to write tactics by observing to which interactive proof steps they correspond.

Flexibility Humans reason in more flexible ways than automation procedures. User interfaces should allow proof steps at any part of a proof as well as free-style transformations of formulas, and they should embrace multiple reasoning styles, such as explicit proofs, proof-by-pointing, and proof-by-search.

Experimentation A strict separation of prover core and prover interface not only helps soundness, but also enables more agile experimentation with new styles of conducting proofs and of interacting with provers.

Figure 1 shows a screenshot of the KeYmaera X user interface with user interface elements annotated by their interaction purpose. We discuss these design choices in more detail in the following paragraphs.

4.1 Familiarity

The KeYmaera X prover kernel implements Hilbert-style proofs by uniform substitution from a small set of locally sound axioms [27] together with a first-order sequent calculus [23]. The user interface of KeYmaera X presents proofs in sequent form as in Figure 2, which enables users to equivalently read the logical transformations as either sequent proof rule uses [23] or axiom uses [27]. The rendering is consistent with the notation used in Section 2 and in the *Foundations of Cyber-Physical Systems* course [25], which should make it easy to switch between proof development on paper and proving in KeYmaera X. Proof suggestions for tactics are rendered by their primary nature either as axioms (e. g., the d$\mathscr{L}$ axiom $[\cup]$, see Figure 2b), proof rules (e. g., the propositional proof rule $\wedge$L, see Figure 2c), or proof rules with input (e. g., the sequent proof rule loop, which requires input, combines multiple axioms in a tactic to implement a proof rule, and creates multiple subgoals, see Figure 2d).

The hypothesis is that the ability to work from a small number of reasoning principles, which is crucial for a small prover core, helps human understanding as well. The experience with the *Foundations of Cyber-Physical Systems* course [25], in which KeYmaera and KeYmaera X have been used, suggests that equivalence axioms indeed make it easier for students to understand reasoning principles than explicit sequent proof rules [23], which obscure inherent dualities for novices. For example, the equivalence $[a\cup b]P \leftrightarrow [a]P\wedge[b]P$ characterizes nondeterministic choices under all circumstances. Its conjunction $\wedge$ and the rules for $\wedge$ make it apparent that such nondeterministic choices will branch in the succedent but not in the antecedent:

$$[\cup],\wedge\mathrm{R}\frac{\Gamma\vdash[a]P,\Delta\quad\Gamma\vdash[b]P,\Delta}{\Gamma\vdash[a\cup b]P,\Delta}\qquad[\cup],\wedge\mathrm{L}\frac{\Gamma,[a]P,[b]P\vdash\Delta}{\Gamma,[a\cup b]P\vdash\Delta}$$

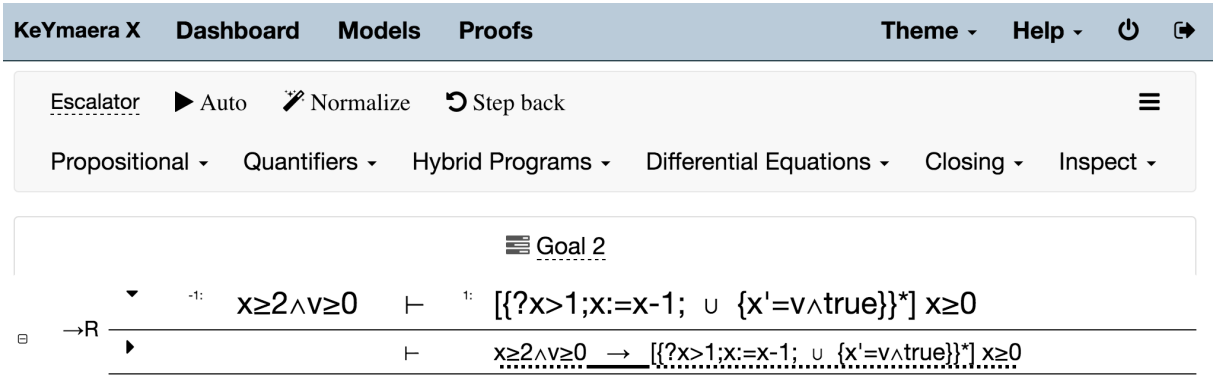
Of course, it is exactly the same reasoning principle either way, but understanding the direct sequent proof rules still requires two logical principles at once compared to the single axiom.

4.2 Traceability

When switching from automated mode to user guidance, it is important to visualize just enough contextual information about the open proof goals to understand where the present subgoals came from and how they relate to the proof of the ultimate conclusion. KeYmaera X shows local views of proofs that illustrate the way how the current question came about and how it is related to the proof of the conclusion. The idea is that this enables traceability and gives local justifications while limiting information to the presently relevant part of the proof.

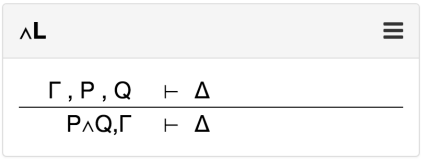
Visualizing the proof state. Visualizing the entire proof tree that unfolds from the original conjecture as tree root is not a viable option, since proofs in d$\mathscr{L}$ unfold into many branches and therefore easily exceed the screen width when rendered in a single tree. Even simple models, such as the escalator example with only four branches (induction base case, induction use case, and stepping down plus moving upwards in the induction step) become hard to navigate and keep track of when both horizontal and vertical scrolling is needed. KeYmaera X, therefore, renders a proof tree in sequent deduction paths from the tree root representing the original conjecture at the bottom of the screen to a leave representing an open goal at the top, see Figure 1.

The sequent deduction paths are arranged in tabs; branching occurs when a deduction step has more than one premise, so that premises are spread over multiple tabs and interconnected with links between the tabs.

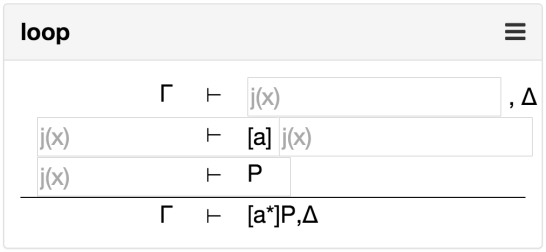


(a) Sequent proof with step → R; the position where → R was applied in the original formula (below the horizontal sequent line) is highlighted with a dotted line, the specific operator with a solid line.

(b) Axiom [∪]



(c) Sequent rule ∧L



(d) Sequent rule loop

Figure 2: Sequent proof, axioms, and proof rules rendered with standard notation.

Proof navigation. Users can focus solely on the open goal by collapsing the entire deduction path (⊟), or they can keep some part of the proof structure visible, e. g., by collapsing only between branching points in the proof, see Figure 1. Triangles left of the sequent path identify groups: when uncollapsed, down/up arrows (≍) indicate the group borders; when collapsed (⟨⟩), the steps in a group are abbreviated into "…". Additionally, sequents can be expanded over multiple lines (▼, one formula per line) or collapsed into a single line (▶).

When proof automation hands over to the user, the topmost line in a sequent deduction path represents an open goal. This means that either the goal cannot be proved at all because the model is wrong, or it is not yet proved because user guidance is needed. In the former case, the counterexample tool allows users to find concrete values that make all formulas on the left of the sequent true but violate all formulas on the right. In the latter case, users are interested in what to do next (see Section 4.3).

4.3 Tutoring

Suggesting possible proof steps. KeYmaera X analyzes the shape of formulas to suggest proof steps on demand. A tactic comes with a description of the shape of its conclusion (must match the current open goal so that the tactic is applicable), a description of the premises that remain to be proved after applying the tactic, and a description of required user input (such as invariants for loops). Such meta-information allows tactic suggestion in two different flavors, as depicted in Figure 3:

- When users know where to continue with the proof (i. e., exactly on which formula or term or part thereof), KeYmaera X displays a dialog with important applicable tactics and their required

input, which resembles the user interaction of KeY [1, 2] and its hybrid systems descendant KeYmaera [28].

- When users know what to do next, KeYmaera X searches for formulas or terms where the desired tactic is applicable.

For example, when pointing to a loop, KeYmaera X suggests the induction tactic loop as well as loop unrolling $[^*]$ as shown in Figure 3. The tactic loop requires a loop invariant $j(x)$ as input and produces three branches that remain to be proved to conclude safety of the program with the loop. Loop unrolling $[^*]$ turns a formula of the shape in boldface (left-hand side $[\alpha^*]P$ of the equivalence) into a formula of the right-hand side $P \wedge [\alpha][\alpha^*]P$ of the equivalence.

For reasons of flexibility, KeYmaera X supports contextual reasoning to apply axioms deeply nested inside a formula, not only on the top-level operator as in Figure 3. As a consequence, users may have many options to work on a single formula. In the induction step of the escalator example

$$\vdash x{>}0 \rightarrow [(?x{>}1;x:=x-1)\cup x'=v]x{>}0$$

we used $\rightarrow$ R top-level to get $x{>}0 \vdash [(?x{>}1;x:=x-1)\cup x'=v]x{>}0$ as a next goal. Alternatively, the non-deterministic choice tactic $[\cup]$ on the right-hand side of the implication would result in a conjunction $\vdash x{>}0 \rightarrow [?x{>}1;x:=x-1]x{>}0 \wedge [x'=v]x{>}0$.

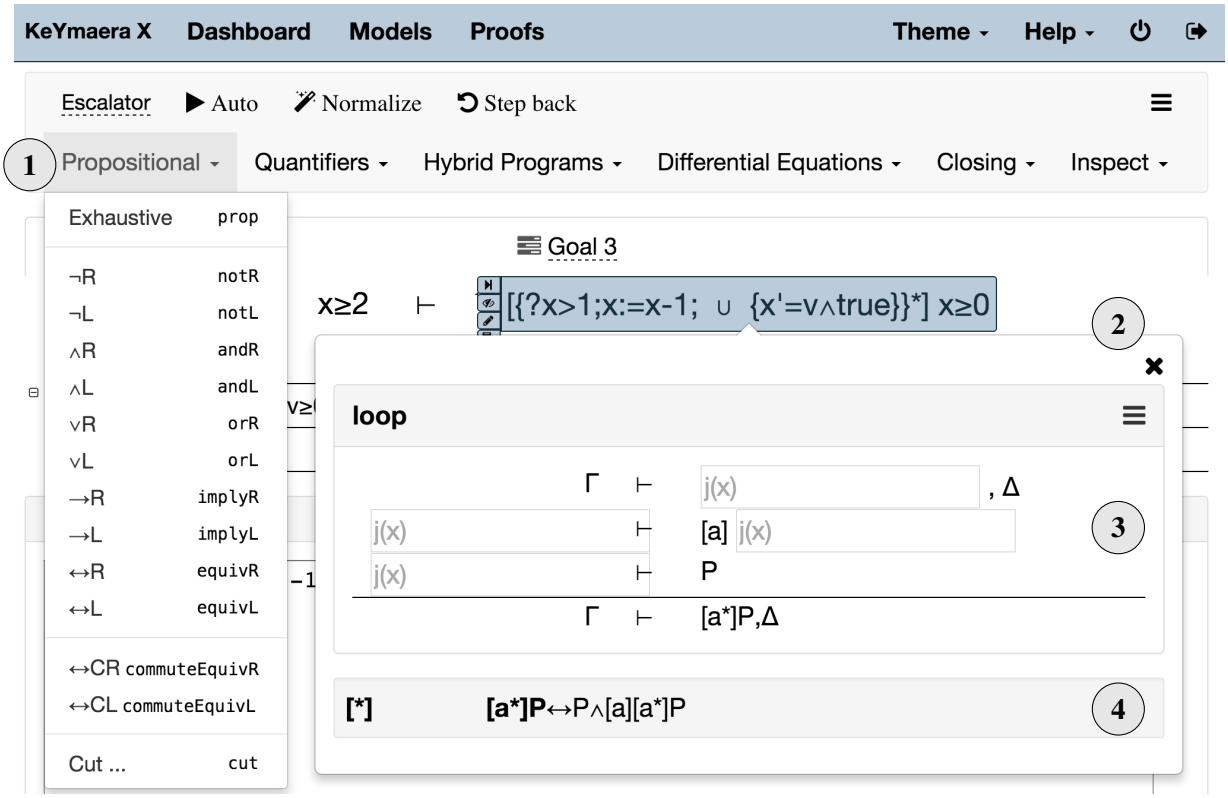


Figure 3: Two ways of asking KeYmaera X for help: ① Know what to do, but not sure where $\rightsquigarrow$ search tactics in the toolbox menu instruct KeYmaera X to apply a specific tactic to the first applicable formula. ② Know where, but not sure what to do $\rightsquigarrow$ tactic suggestion in the context menu of a formula: ③ tactic with input; ④ apply an axiom.

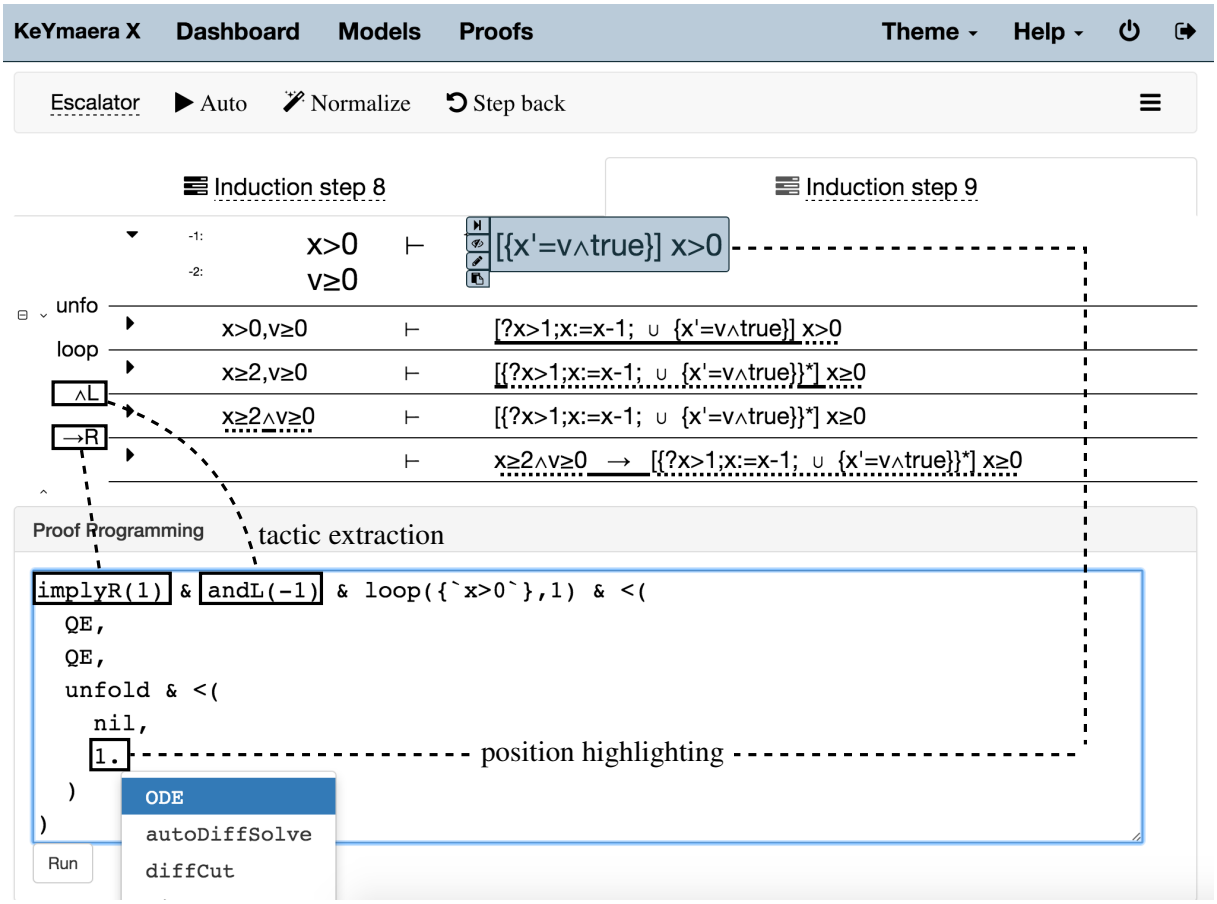


Figure 4: Tactic editor below sequent view. The tactic is extracted automatically from the point-and-click interaction: the proof closed both the induction base case and the induction use case by quantifier elimination QE; the sequent view shows two open goals in the induction step, which result from using unfold (splits the choice, handles the test and the assignment, but stops at the differential equation). The tactic editor displays tactic suggestions akin to the tactic popover in the sequent view. Here, it display suggestions for the formula at position "1", which is also highlighted in the sequent view.

For novice users, it is often easiest to focus on the top-level operator and work on formulas outside-in, i. e., apply $\rightarrow$ R first and then $[\cup]$ next. But it quickly becomes more convenient to apply proof rules in any order anywhere in the middle of the formulas to follow whatever line of thought the user may have in mind. Such proofs in arbitrary order also often reduce the branching or repetition of proof steps in different branches.

Tactic extraction. In order to conduct proofs effectively, interactive theorem provers typically ship with extensive tactic libraries (e. g., Coq [20], Isabelle [22]), accompanied with library documentation and examples to facilitate learning. Still, extensive tactic libraries incur a steep learning curve. We conjecture that observing how tactics evolve while doing a proof (similar to observing an expert) can reduce the steep learning curve associated with extensive tactic libraries. KeYmaera X automatically generates tactics that correspond to the point-and-click interaction that the user performed and displays these tactics below the graphical sequent view, see Figure 4.

The graphical sequent view and the tactic editor operate on the same proof, so that users can switch between both interaction concepts as they see fit. As a rule of thumb, the sequent view is designed for conducting specific steps at specific positions (e. g., use a specific equality $x = y$ to rewrite x into y in some other term), while the tactic editor is intended for describing proof search (e. g., repeat some step exhaustively $\wedge$L('L)* or follow a high-level proof strategy such as unfold & ODE & QE).

4.4 Flexibility

To allow users to reason in flexible ways and reduce manual proof effort, KeYmaera X supports proof steps by pointing at formula parts and lets users transform and abbreviate formulas as well as execute tactic scripts. As underlying technique for proof-by-pointing and flexible reasoning anywhere in formulas, KeYmaera X provides contextual reasoning CQ and CE [27], so that questions about contextual equality or equivalence in a context $C\{\dots\}$ reduce to reasoning about arguments as follows.

$$\text{CQ}\,\frac{\vdash f() = g()}{\vdash C\{p(f())\} \leftrightarrow C\{p(g())\}} \qquad \text{CE}\,\frac{\vdash P \leftrightarrow Q}{\vdash C\{P\} \leftrightarrow C\{Q\}}$$

When contextual reasoning is combined with uniform substitution US and axiom lookup, axioms can be applied inside formulas, which enables proof by pointing at the desired formula part. Often, the next proof step follows unambiguously from just the shape of the pointed formula part by indexing [27], so that the proof advances without further user input.

$$\text{cut}\,\frac{\vdash y > 2 \wedge 7 > 5 \qquad \text{CE}\,\dfrac{\text{US}\,\dfrac{\text{ax}\,[:=]\,\dfrac{*}{\vdash [x := f()]p(x) \leftrightarrow p(f())}}{\vdash [x := 7]x > 5 \leftrightarrow 7 > 5}}{\vdash y > 2 \wedge [x := 7]x > 5 \leftrightarrow y > 2 \wedge 7 > 5}}{\vdash y > 2 \wedge \underbrace{[x := 7]x > 5}}$$

unifies with underlined key in axiom $\underline{[x:=f()]p(x)} \leftrightarrow p(f())$ $\rightsquigarrow$ replace with $p(f())$, which is $7>5$

Note that side branches with contextual reasoning, uniform substitution, and axiom lookup (as in the proof above) close fully automatically. Hence, the user interface displays only the result of applying axioms by pointing, while it hides the minutia of the side deductions from the user as in the following example.

$$\text{useAt}\,\frac{\vdash P \rightarrow ([x' = 2]x \geq 5) \vee Q}{\vdash P \rightarrow (\underbrace{[x' = 2 \cup x := 5]x \geq 5}) \vee Q}$$

$[\beta]B \rightarrow (\underline{[\alpha \cup \beta]}B \leftrightarrow [\alpha]B)$

4.5 Experimentation

The prover kernel and the prover interface are strictly separated, to the extent that the prover kernel only knows about making single deduction steps and combining them, but is completely oblivious of organizing these steps in a tree structure. For soundness, it suffices to know that any step between the original conjecture and the current subgoals must have been done in the prover kernel. Intermediate steps are only necessary to repeat a proof from the original conjecture, and can therefore be tracked outside the prover kernel. As a result, proof organization features (such as undoing proof steps, pruning the proof tree) can be implemented conveniently in the user interface without affecting soundness.

The separation between the prover core and the tactics becomes especially useful when adding compiled tactics to the server-side implementation. Complementing the interpreted tactics from the web-based user interface, server-side tactics can base on a rich implementation language (Scala) to try proof

steps that are only sound in the usual cases and just rely on the prover core to catch when they happen to be applied in one of the unsound corner cases.

5 Evaluation Concept

Compared to its predecessor KeYmaera [28], the clean-slate implementation KeYmaera X introduces significant changes in user interaction. Although the user interaction changes are based on informal feedback from KeYmaera users and our own observations on how students used KeYmaera, it still remains to be checked by an experimental user study which style of user interaction is more effective. Additionally, KeYmaera X introduces new interaction concepts (e. g., tactic programming and tactic recording), which seem promising for experienced users to conduct large proofs, but are not yet backed by evidence that indeed prover interaction is improved compared to only mouse-based interaction. Similar user-related research questions were addressed in a recent controlled user experiment [13], which followed methods from empirical software engineering [32] to compare two very different user interfaces of the KeY theorem prover [1, 2]. The obtained results are encouraging pointers towards controlled user experiments being an appropriate method for testing theorem prover interaction.

Such controlled experiments, however, require a large number of participants with varying experience and deliberately exposing them to different user interaction concepts. In order to avoid adverse effects from pre-assigned interaction styles, we propose gathering data from normal operation on how often users rely on certain interaction patterns (e. g., clicking vs. automated tactic), how often they use a certain functionality during a proof, and how successful they were, even if that might lead to selection bias. Learnability of theorem provers can be measured based on cognitive dimensions [8], such as visibility (how easy are relevant steps accessed), juxtaposability (different notations side-by-side), viscosity (resistance to change), premature commitment (to an order how to do steps), error-proneness (how likely are errors), and consistency (similar information presented in similar ways). Theorem provers, especially in education and also in industrial applications, should have high visibility and juxtaposability, high consistency, and low viscosity [12]. The following metrics should be easily recordable from KeYmaera X without changing user interaction and give insights into cognitive dimensions.

Interaction concept Number of proof steps by clicking/tactic programming/automated proof search. Supporting evidence: number of proof steps at top-level/inside formulas; for clicking: number of steps executed by pointing/from tactic suggestion. Related to visibility and premature commitment.

Functionality Number of undo operations, length of pruned deduction paths with distance to next branching point above and below, number of find counterexample operations. Number of interactions in pruned proofs. Related to viscosity and error-proneness.

Time Proof duration (including estimates of user time and number of reproof attempts)

Trends Compare trends as students gain experience and examples become more complex

We believe this data should enable a study that tests the following hypotheses.

Interaction preference Novice users prefer automated proof search over clicking over tactic programming. Novice users prefer working top-level over working inside formulas. Experienced users balance interaction.

Functionality Novice users either undo short paths or entire proofs and end up with more open branches. Experienced users undo branching and exercise branching control techniques.

Trends Automated usage drops with increased example complexity and student experience, while use of tactic programming increases; the focus on applying steps top-level drops while applying steps inside formulas increases with experience.

Influence Students that engage more direct control over proofs also for simpler examples are more productive in conducting proofs of complex cases than students who relied on full automation earlier on.

The quantitative insights from these metrics could be augmented with qualitative evaluation techniques, as demonstrated being effective for theorem provers, e. g., with focus groups [7] (users subjectively express their perception of or opinion on the user interface), with questionnaires [6], or by co-operative evaluation [15] (users verbalize their interactions while using the theorem prover).

6 Related Work

The verification landscape spans a wide variety of approaches from automated theorem provers and reachability analysis tools to interactive theorem provers.

Automated theorem provers (e. g., Vampire [16]) and reachability analysis tools (e. g., SpaceEx [10]) strive for fully automatic verification without user interaction, but their scope is inherently limited in cases that are not semidecidable, such as hybrid systems.

Auto-active verifiers (e. g., Dafny [17, 18], AutoProof [30]) put the model or code first and hide the verification engine, but support user guidance through annotations in the code. The basic idea is to make verification an integral part of (software) development that should be performed in the background by an IDE, much like background compilation. The downside of such an approach is that the verification steps are hidden entirely from the user, which can make it hard to resolve proofs with additional annotations when the verifier is stuck because the proof state and the verifier's working principles are opaque. The KeY interactive verification debugger [14] combines code annotations with interactive verification in KeY [1, 2] to supplement manual proofs when automated proving fails at some goals.

Interactive theorem provers, such as Coq [20] and Isabelle [22], primarily interact with users through tactic scripts, such as structured proofs in Isabelle/Isar [21]. Their user interfaces (e. g., CoqIDE [20], ProofGeneral [5], jEdit [31]) focus on text editing support for writing tactics and let users inspect the proof state and open goals by placing the cursor in the tactic script. Navigation with cursors introduced a limited form of proof by pointing [4] to fold or unfold equations. KeYmaera X advances proof-by-pointing to transform all or parts of a formula following the shape of an axiom or fact. PeaCoq[3] aims to make the proof state more accessible to users by providing a proof tree.

KeYmaera X combines concepts from automated, auto-active, and interactive theorem proving: it comes with fully automated proof search tactics for hybrid systems of limited scope (i. e., when a loop invariant can be found, a symbolic solution of a differential equation can be found to serve as an oracle for differential invariants, and the resulting arithmetic is tractable); it supports annotations for loop invariants and differential invariants, since both serve as model documentation as well as proof guidance; and finally, it allows users to conduct and finish proofs themselves with tactics and a graphical user interface.

[3] `http://goto.ucsd.edu/peacoq/`

7 Conclusion

The user interface of KeYmaera X is based on the principles of familiarity, traceability, tutoring, flexibility, and experimentation. It supports several different user interaction styles to make progress in proofs. The web-based user interface applies proof steps when clicking on formulas in the sequent view, and it batch-runs proof steps by automated search tactics as well as through tactic programming.

Future work includes evaluation in controlled user experiments, as described in the evaluation concept. On the basis of user studies, we expect to gain insights into how to best teach hybrid systems theorem proving, tactic programming, tactic generalization, and proof search. We are working on improved proof exploration, e.g., with timeouts (e. g., allow QE 5s to close; if it does not close within the budgeted time, try something else). Failed proof attempts or expired timeouts need a robust approach to making proofs portable and repeatable.

Acknowledgments The authors thank Nathan Fulton, Brandon Bohrer, Jan-David Quesel, Ran Ji, and Marcus Völp for their support in implementing KeYmaera X, Sarah Loos, Jean-Baptiste Jeannin, João Martins, Khalil Ghorbal, and Sarah Grebing for feedback and discussions on the user interface, as well as the anonymous reviewers for feedback on the manuscript.

References

[1] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager & Peter H. Schmitt (2005): *The KeY tool*. *Software and System Modeling* 4(1), pp. 32–54, doi:10.1007/s10270-004-0058-x.

[2] Wolfgang Ahrendt, Bernhard Beckert, Daniel Bruns, Richard Bubel, Christoph Gladisch, Sarah Grebing, Reiner Hähnle, Martin Hentschel, Mihai Herda, Vladimir Klebanov, Wojciech Mostowski, Christoph Scheben, Peter H. Schmitt & Mattias Ulbrich (2014): *The KeY Platform for Verification and Analysis of Java Programs*. In Dimitra Giannakopoulou & Daniel Kroening, editors: *Verified Software: Theories, Tools and Experiments - 6th International Conference, VSTTE 2014, Vienna, Austria, July 17-18, 2014, Revised Selected Papers*, *LNCS* 8471, Springer, pp. 55–71, doi:10.1007/978-3-319-12154-3_4.

[3] J. Stuart Aitken, Philip D. Gray, Thomas F. Melham & Muffy Thomas (1998): *Interactive Theorem Proving: An Empirical Study of User Activity*. *J. Symb. Comput.* 25(2), pp. 263–284, doi:10.1006/jsco.1997.0175.

[4] David Aspinall & Christoph Lüth (2004): *Proof General meets IsaWin: Combining Text-Based And Graphical User Interfaces*. *Electr. Notes Theor. Comput. Sci.* 103, pp. 3–26, doi:10.1016/j.entcs.2004.09.011.

[5] David Aspinall, Christoph Lüth & Daniel Winterstein (2007): *A Framework for Interactive Proof*. In Manuel Kauers, Manfred Kerber, Robert Miner & Wolfgang Windsteiger, editors: *Towards Mechanized Math. Assistants, 14th Symp., Calculemus, 6th Int. Conf., MKM, Hagenberg, Austria, June 27-30, 2007, Proc.*, *LNCS* 4573, Springer, pp. 161–175, doi:10.1007/978-3-540-73086-6_15.

[6] Bernhard Beckert & Sarah Grebing (2012): *Evaluating the Usability of Interactive Verification Systems*. In Vladimir Klebanov, Bernhard Beckert, Armin Biere & Geoff Sutcliffe, editors: *Proceedings of the 1st International Workshop on Comparative Empirical Evaluation of Reasoning Systems, Manchester, United Kingdom, June 30, 2012*, *CEUR Workshop Proceedings* 873, CEUR-WS.org, pp. 3–17.

[7] Bernhard Beckert, Sarah Grebing & Florian Böhl (2014): *A Usability Evaluation of Interactive Theorem Provers Using Focus Groups*. In Carlos Canal & Akram Idani, editors: *Software Engineering and Formal Methods - SEFM 2014 Collocated Workshops: HOFM, SAFOME, OpenCert, MoKMaSD, WS-FMDS, Grenoble, France, September 1-2, 2014, Revised Selected Papers*, *LNCS* 8938, Springer, pp. 3–19, doi:10.1007/978-3-319-15201-1_1.

[8] Alan F. Blackwell, Carol Britton, Anna Louise Cox, Thomas R. G. Green, Corin A. Gurr, Gada F. Kadoda, Maria Kutar, Martin Loomes, Chrystopher L. Nehaniv, Marian Petre, Chris Roast, Chris Roe, Allan Wong & R. Michael Young (2001): *Cognitive Dimensions of Notations: Design Tools for Cognitive Technology*. In Meurig Beynon, Chrystopher L. Nehaniv & Kerstin Dautenhahn, editors: *Cognitive Technology: Instruments of Mind, 4th International Conference, CT 2001, Warwick, UK, August 6-9, 2001, Proceedings*, *Lecture Notes in Computer Science* 2117, Springer, pp. 325–341, doi:10.1007/3-540-44617-6_31.

[9] James H. Davenport & Joos Heintz (1988): *Real Quantifier Elimination is Doubly Exponential*. *J. Symb. Comput.* 5(1/2), pp. 29–35, doi:10.1016/S0747-7171(88)80004-X.

[10] Goran Frehse, Colas Le Guernic, Alexandre Donzé, Scott Cotton, Rajarshi Ray, Olivier Lebeltel, Rodolfo Ripado, Antoine Girard, Thao Dang & Oded Maler (2011): *SpaceEx: Scalable Verification of Hybrid Systems*. In Ganesh Gopalakrishnan & Shaz Qadeer, editors: *CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proc.*, *LNCS* 6806, Springer, pp. 379–395, doi:10.1007/978-3-642-22110-1_30.

[11] Nathan Fulton, Stefan Mitsch, Jan-David Quesel, Marcus Völp & André Platzer (2015): *KeYmaera X: An Axiomatic Tactical Theorem Prover for Hybrid Systems*. In Amy P. Felty & Aart Middeldorp, editors: *CADE*, *LNCS* 9195, Springer, pp. 527–538, doi:10.1007/978-3-319-21401-6_36.

[12] D. Diaper G. Kadoda, R. G. Stone (1999): *Desirable features of educational theorem provers - a cognitive dimensions viewpoint*. In: *11th Annual Workshop of Psychology of Programming Interest Group*, PPIG, pp. 1–6.

[13] Martin Hentschel, Reiner Hähnle & Richard Bubel (2016): *An empirical evaluation of two user interfaces of an interactive program verifier*. In Lo et al. [19], pp. 403–413, doi:10.1145/2970276.2970303.

[14] Martin Hentschel, Reiner Hähnle & Richard Bubel (2016): *The interactive verification debugger: effective understanding of interactive proof attempts*. In Lo et al. [19], pp. 846–851, doi:10.1145/2970276.

[15] Michael J. Jackson (1997): *Evaluation of a Semi-Automated Theorem Prover (Part II)*.

[16] Laura Kovács & Andrei Voronkov (2013): *First-Order Theorem Proving and Vampire*. In Natasha Sharygina & Helmut Veith, editors: *Computer Aided Verification - 25th Int. Conf., CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proc.*, *LNCS* 8044, Springer, pp. 1–35, doi:10.1007/978-3-642-39799-8_1.

[17] K. Rustan M. Leino (2013): *Developing verified programs with Dafny*. In David Notkin, Betty H. C. Cheng & Klaus Pohl, editors: *35th Int. Conf. on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, IEEE Computer Soc., pp. 1488–1490, doi:10.1109/ICSE.2013.6606754.

[18] K. Rustan M. Leino & Valentin Wüstholz (2014): *The Dafny Integrated Development Environment*. In Catherine Dubois, Dimitra Giannakopoulou & Dominique Méry, editors: *Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014.*, *EPTCS* 149, pp. 3–15, doi:10.4204/EPTCS.149.2.

[19] David Lo, Sven Apel & Sarfraz Khurshid, editors (2016): *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. ACM, doi:10.1145/2970276.

[20] The Coq development team (2015): *The Coq proof assistant reference manual*. LogiCal Project. Available at `http://coq.inria.fr`. Version 8.5.

[21] Tobias Nipkow (2002): *Structured Proofs in Isar/HOL*. In Herman Geuvers & Freek Wiedijk, editors: *Types for Proofs and Programs, 2nd Int. Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002, Selected Papers*, *LNCS* 2646, Springer, pp. 259–278, doi:10.1007/3-540-39185-1_15.

[22] Tobias Nipkow, Lawrence C. Paulson & Markus Wenzel (2002): *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. *LNCS* 2283, Springer.

[23] André Platzer (2008): *Differential Dynamic Logic for Hybrid Systems*. *J. Autom. Reas.* 41(2), pp. 143–189, doi:10.1007/s10817-008-9103-8.

[24] André Platzer (2012): *Logics of Dynamical Systems*. In: *LICS*, IEEE, pp. 13–24, doi:10.1109/LICS.2012.13.

[25] André Platzer (2013): *Teaching CPS Foundations With Contracts*. In: *CPS-Ed*, pp. 7–10.

[26] André Platzer (2015): *A Uniform Substitution Calculus for Differential Dynamic Logic*. In Amy P. Felty & Aart Middeldorp, editors: *CADE*, *LNCS* 9195, Springer, pp. 467–481, doi:10.1007/978-3-319-21401-6_32.

[27] André Platzer (2016): *A Complete Uniform Substitution Calculus for Differential Dynamic Logic*. *J. Autom. Reas.*, doi:10.1007/s10817-016-9385-1.

[28] André Platzer & Jan-David Quesel (2008): *KeYmaera: A Hybrid Theorem Prover for Hybrid Systems (System Description)*. In Alessandro Armando, Peter Baumgartner & Gilles Dowek, editors: *Automated Reasoning, 4th International Joint Conference, IJCAR 2008, Sydney, Australia, August 12-15, 2008, Proc.*, *LNCS* 5195, Springer, pp. 171–178, doi:10.1007/978-3-540-71070-7_15.

[29] Jan-David Quesel, Stefan Mitsch, Sarah M. Loos, Nikos Arechiga & André Platzer (2016): *How to model and prove hybrid systems with KeYmaera: a tutorial on safety*. *STTT* 18(1), pp. 67–91, doi:10.1007/s10009-015-0367-0.

[30] Julian Tschannen, Carlo A. Furia, Martin Nordio & Nadia Polikarpova (2015): *AutoProof: Auto-Active Functional Verification of Object-Oriented Programs*. In Christel Baier & Cesare Tinelli, editors: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, London, UK, April 11-18, 2015. Proceedings*, *LNCS* 9035, Springer, pp. 566–580, doi:10.1007/978-3-662-46681-0.

[31] Makarius Wenzel (2012): *Isabelle/jEdit - A Prover IDE within the PIDE Framework*. In Johan Jeuring, John A. Campbell, Jacques Carette, Gabriel Dos Reis, Petr Sojka, Makarius Wenzel & Volker Sorge, editors: *Intelligent Computer Mathematics - 11th International Conference, AISC 2012, 19th Symp., Calculemus 2012, 5th Int. Workshop, DML 2012, 11th Int. Conf., MKM 2012, Systems and Projects, Held as Part of CICM 2012, Bremen, Germany, July 8-13, 2012. Proc.*, *LNCS* 7362, Springer, pp. 468–471, doi:10.1007/978-3-642-31374-5.

[32] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson & Björn Regnell (2012): *Experimentation in Software Engineering*. Springer, doi:10.1007/978-3-642-29044-2.

Interfacing Automatic Proof Agents in Atelier B: Introducing "iapa"*

Lilian Burdy
ClearSy System Engineering, Aix-en-Provence, France
lilian.burdy@clearsy.com

David Déharbe†
david.deharbe@clearsy.com
ClearSy System Engineering, Aix-en-Provence, France

Étienne Prun
ClearSy System Engineering, Aix-en-Provence, France
etienne.prun@clearsy.com

The application of automatic theorem provers to discharge proof obligations is necessary to apply formal methods in an efficient manner. Tools supporting formal methods, such as Atelier B, generate proof obligations fully automatically. Consequently, such proof obligations are often cluttered with information that is irrelevant to establish their validity.

We present iapa, an "Interface to Automatic Proof Agents", a new tool that is being integrated to Atelier B, through which the user will access proof obligations, apply operations to simplify these proof obligations, and then dispatch the resulting, simplified, proof obligations to a portfolio of automatic theorem provers.

1 Introduction

Historically, the B Method[1] was introduced in the late 80's to design provably safe software. Promoted and supported by RATP, the B method and Atelier B, the tool implementing it, have been successfully applied to the industry of transportation leading to a worldwide implementation of the B technology for safety critical software, mainly as automatic train controllers for subways.

The development of such controllers corresponds to "big" industrial project. To give an idea of the size of such development, a train controller is composed of different software components communicating together. Taking from a real example, the size of the critical parts of the controller is around 500.000 lines of B which give after translation 300.000 lines of Ada and 160.000 generated proof obligations. The proof, already mainly automated, of those proof obligations is a substantial part of the development cost for such project. Limiting these costs by using more efficient provers, or by using more efficiently provers is a real concern in industry.

Atelier B comes with two provers developed with the tool in the 90's. Recently the ProB model checker [9] has been added as a prover than can be called during interactive proof session. The Bware project[4] [8] aims to provide a mechanized framework to apply automated theorem provers, such as first order provers and SMT (Satisfiability Modulo Theories) solvers on proof obligations coming from the development of industrial applications using the B method. This approach is produces proof obligations in the format of Why3 [3], which is used then responsible for calling different automatic theorem provers with the adequate input, and for interpreting their output and for synthesizing a verification result.

*This work is partly supported by the Bware (ANR-12-INSE-0010, http://bware.lri.fr/) project of the French national research organization (ANR).

†On leave from Universidade Federal do Rio Grande do Norte.

SMT provers are routinely used by any tool that have to deal with a logic-based verification task. Notably, they have already been added as a plugin in Rodin[7], another IDE for Event-B, a formal method closely related to the B method. Concerning Bware, promising first results[10] [5] have already been published.

We present here the integration of such automated provers in Atelier B considering the specificities of proof obligations produced by industrial software developments, described in section 2. The basic elements of this integration are the Why3-based bridge to automated theorem provers developed in the Bware project and a new approach for efficiently selecting the relevant parts of a proof obligation. The principles of this latter aspect are presented in section 3. The implementation of this functionality in a graphical user interface is then presented in section 4.

2 Proof obligations

Verifying program by proving verification condition (called proof obligations here) leads to deal with huge lemmas. This problem is not specific to the B Method, indeed the same observation is done, for example, in [6] for verification conditions issued from C program verification.

Concerning the B Method, the figure 1 shows a proof obligation template. One can see that hypotheses are collected in many clauses of many components. As an example, let us consider the declaration of constants: as the B Method is a modular software development method, constants are usually declared in some specific components. In those components, constants are declared with properties. When a function needs to use a constant, the component is seen and with the needed constant come all the properties of all the constants of the component. So all the proof obligations will have as hypothesis the properties of all the constants of the seen component even if only few of them are relevant for the proof.

The incremental approach to refinement is another source of growth in the size of the proof obligations. Indeed, in each new refinement, all the proof obligations will include the contexts from all the components that come before in the refinement chain.

In the real project described previously, the average number of hypotheses for a lemma is around 2000 formulas. Some proof obligations can contain more that 4000 hypotheses. Of course, not all hypotheses are necessary in the proof of the goal. To use efficiently automated provers on such lemma, we argue that it is necessary to filter relevant hypotheses. This is the motivation for the development of an “Interface to Automatic Proof Agents”, giving the users of Atelier B the means to build scripts constructing a mini-lemma from a generated proof obligation, and to submit such mini-lemmas to the provers.

3 Core functionality in iapa

Atelier B already contains an interface to discharge proof obligations: the interactive prover. Indeed, this is the part of the interface where users spend most of their time. Some functionalities in iapa are similar to those found in the interactive prover and will be familiar to the users.

The iapa tool is invoked within Atelier B on a given component, once the proof obligations of that component have been generated (as is the case with the interactive prover). In Atelier B, all the proof obligations of a component are available in a single file, and can be either in a legacy format called the “theory language” or in a XML-based format. Only the latter contains typing annotations and it is this format that has been chosen in the Bware initiative as the basis for interfacing with the Why3 platform. Thus, iapa reads the proof obligations of a given component from the XML-based file.

$A_1 \wedge$ *"Machine parameter constraints"*

$B_1 \wedge \ldots \wedge B_{n-1} \wedge$ *"Properties of constants of previous refinements"*

$B_n \wedge$ *"Properties of refinement constants"*

$B_s \wedge$ *"Properties of constants of components seen"*

$B_{i_1} \wedge B_{i_2} \wedge$ *"Properties of constants of components included"*

$[X_{i_1}, x_{i_1} : N_{i_1}, n_{i_1}](I_{i_1} \wedge L_{i_1} \wedge J_{i_1}) \wedge$ *"Invariants and assertions"*
$[X_{i_2}, x_{i_2} : N_{i_2}, n_{i_2}](I_{i_2} \wedge J_{i_2}) \wedge$ *"of included components"*

$I_s \wedge J_s \wedge$ *"Invariants and assertions of components seen"*

$(I_1 \wedge J_1) \wedge \ldots \wedge (I_n \wedge J_n) \wedge$ *"Invariants and assertions of the vertical development"*

Q_1 *"Precondition of the abstract operation"*

$\Rightarrow$

$Q_n \wedge$ *"Precondition of the refinement operation"*

"Refinement operation applied to the negation of the specified operation"
"applied to the negation of the invariant"
$[[u_1 : u_1'] V_n] \neg [V_{n-1}] \neg (I_n \wedge u_1 u_1')$

Figure 1: Theoretical operation refinement proof obligation

In the iapa interface, proof obligations are grouped according to their origin in the corresponding B component (assertions, initialization, operations). Navigating through these proof obligations is a first core functionality available in iapa, and its principles mimick those of the interactive prover. Regarding this aspect, one important difference with respect to the interactive prover is that well-definedness proof obligations are presented together with those proof obligations instead of in a separate project.

Formally, a proof obligation is a pair (Γ, φ), where Γ is a set of hypotheses and φ is the goal. The main goal of iapa is to assist the user in selecting the relevant information in the current proof obligation. Formally, this consists in building a new proof obligation (Γ', φ) such that $\Gamma' \subseteq \Gamma$. In iapa, the new, simplified, proof obligation is termed the *lemma*. The interface also contains means to submit lemmas to a portfolio of automatic theorem provers.

One notable requirement of iapa is that the steps leading to the construction of a lemma for a given proof obligation can be applied automatically to other proof obligations. This is achieved by means of two kinds of entities: *contexts* and *lexicons*, that the user has to manipulate and combine in order to build a lemma.

A context $\gamma \subseteq \Gamma$ is a set of hypotheses that originate from the proof obligation. When a proof obligation is loaded in iapa, a number of contexts are pre-defined:

- all contains all the hypotheses;

- local contains all the hypotheses that are local in the proof obligation;
- global contains all but the local hypotheses;
- a number of contexts that correspond to the different sections in a B component (properties, invariants, etc.);
- B definitions contains hypotheses on pre-defined sets such as the range of implementable integers.

A lexicon λ is a set of free identifiers of the original proof obligations. Assuming fv returns the set of free identifiers in a formula, then $\lambda \subseteq \bigcup\{\mathsf{fv}(\psi) \mid \psi \in \Gamma \cup \{\varphi\}\}$. Initially, there is a single pre-defined lexicon, named goal, and containing $\mathsf{fv}(\varphi)$ (the free identifiers in the goal).

At any time, the state of iapa contains the following elements:

- (Γ, φ) the current proof obligation;
- $\mathscr{C}$: a set of contexts ($\forall x \in \mathscr{C}, x \subseteq \Gamma$);
- $\mathscr{L}$: a set of lexicons ($\forall x \in \mathscr{L}, x \subseteq \bigcup\{\mathsf{fv}(\psi) \mid \psi \in \Gamma \cup \{\varphi\}\}$);
- c: a current context ($c \in \mathscr{C}$);
- l: a current lexicon ($l \in \mathscr{L}$);
- S: a set of selected hypotheses ($S \subseteq \Gamma$).

The pre-defined values for $\mathscr{C}$ and $\mathscr{L}$ are as described previously, those of c, l and S are local, goal, and $\emptyset$, respectively. Then the commands on contexts and lexicons currently implemented in iapa are the following:

- ah add the hypotheses in the current context to the set of selected hypotheses;
- dh removes the hypotheses in the current context from the set of selected hypotheses;
- chctx(c) sets the current context to c;
- chctx(l) sets the current lexicon to l;
- mklex creates a new lexicon with the free identifiers of the current context;
- mklex(i1,..., in) creates a new lexicon with the given identifiers;
- mkctx(Some) creates a new context containing the hypotheses in the current context that have at least one free identifier in the current lexicon;
- mkctx(All) creates a new context containing the hypotheses in the current context such that their free identifiers includes the current lexicon;
- mkctx(h1,...,hn) creates a new context containing the given hypotheses.

The condition and effect of the execution of these commands are summarized in table 1, and some illustrative scripts are presented in table 2. The following section presents the iapa interface, including how the user can play such commands.

command	effect	condition
ah	$S := S \cup c$	
dh	$S := S \setminus c$	
chctx(c)	$c :=$ c	$\mathsf{c} \in \mathscr{C}$
chctx(l)	$l :=$ l	$\mathsf{l} \in \mathscr{L}$
mklex	$\mathscr{L} := \mathscr{L} \cup \{\mathsf{fv}(c)\}$	$\mathsf{fv}(c) \neq \emptyset$
mklex(i1,...,in)	$\mathscr{L} := \mathscr{L} \cup \{\{\mathsf{i1}, \cdots, \mathsf{in}\}\}$	$\mathsf{i1} \in l \cdots \mathsf{in} \in l$
mkctx(Some)	$\mathscr{C} := \mathscr{C} \cup \{\{h \mid h \in c \wedge \mathsf{fv}(h) \cap l \neq \emptyset\}\}$	$\{h \mid h \in c \wedge \mathsf{fv}(h) \cap l \neq \emptyset\} \neq \emptyset$
mkctx(All)	$\mathscr{C} := \mathscr{C} \cup \{\{h \mid h \in c \wedge l \subseteq \mathsf{fv}(h)\}\}$	$\{h \mid h \in c \wedge l \subseteq \mathsf{fv}(h)\} \neq \emptyset$
mkctx(h1,...,hn)	$\mathscr{C} := \mathscr{C} \cup \{\{\mathsf{h1}, \cdots, \mathsf{hn}\}\}$	$\mathsf{h1} \in c \cdots \mathsf{hn} \in c$

Table 1: Formalization of iapa commands.

script	description
ah	builds lemma containing only local hypotheses
chctx(all) & ah	builds lemma identical to proof obligation
mklex & chctx(all) & mkctx(Some) & ah	builds lemma with hypotheses containing an identifier in the local hypotheses

Table 2: Example iapa scripts (& being used as command separator).

4 The iapa tool

The core functionality is presented in a graphical user interface that has to be launched from Atelier B's main window on a given component. The functionalities are offered both by textual and point-and-click means. Figure 2 contains a screenshot of the initial contents of the window. At that point, two views are populated: *Provers* and *Proof obligations*. The latter is in Atelier B's database. The former is obtained by querying the automatic provers currently installed. Since, at the moment, the access to these provers is realized through Why3, this information is found automatically using Why3 and its auto-configuration facilities. Besides the menu and the tool bar found at the top of the window, the *Command* section contains a widget containing a command-line interface to iapa. Here, the user has already typed the command ne, which, when executed, will open the next proof obligation.

Opening a proof obligation results in filling the *Goal*, *Context manager* and *Lexicon manager* sections, and enables actions corresponding to the core iapa functionalities. Figure 3 shows the contents of the iapa window after the user has managed to complete a proof after having selected some hypotheses in the context (using here one of the scripts presented in table 2 and started the provers on the resulting lemma with the pr command. The steps realized by the user are saved and displayed in the *Script* section. Also the *Messages* section is dedicated to the display of feed-back information.

5 Conclusions and future work

This paper presents an on-going work to reduce the cost of discharging proof obligations when applying formal methods in an industrial environment. This work is embodied in iapa, a prototype for an extension of Atelier B aiming at both integrating additional proof engines and offer hypotheses selection facilities

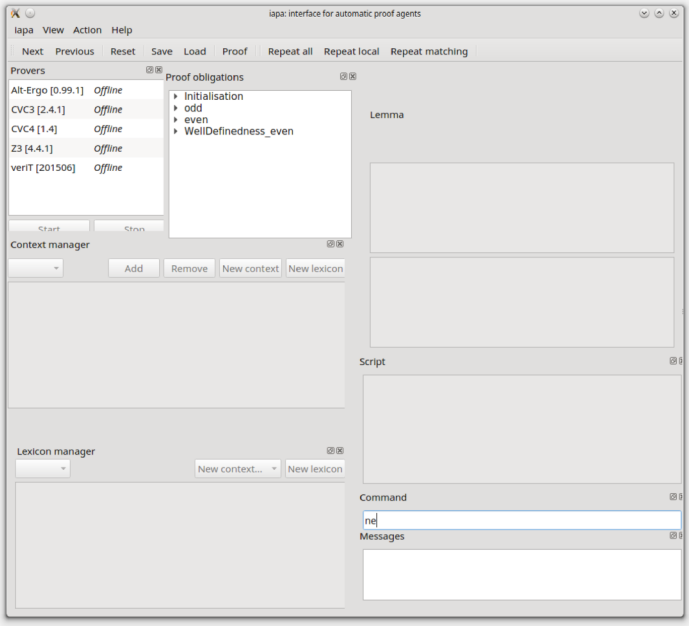


Figure 2: Initial contents of the iapa window.

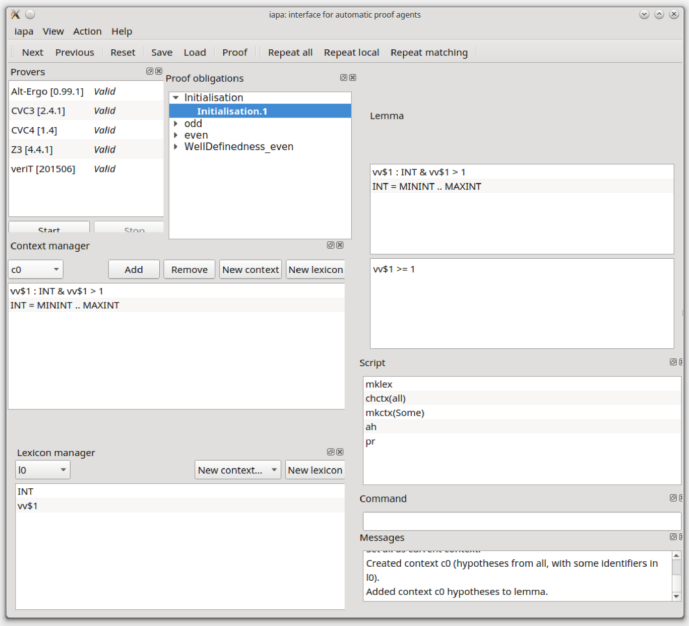


Figure 3: Contents of the iapa window during a session.

to the user.

The effectiveness of the approach will be assessed through a systematic evaluation on a representative set of industrial projects. The results of this evaluation will decide whether iapa is eventually deployed together with the distributions of Atelier B. We also plan to improve the usability of iapa, by adding hypotheses selection criteria based on formula patterns, and also taking user feed-back into account.

Certification is another important aspect of tooling in an industrial setting for safety-critical systems. The historical provers in Atelier B have been certified; but certifying new tools is costly. We forecast some solutions to address this issue in iapa. First, since some automated theorem provers are proof-producing, we envision using the proofs thus produced to build proofs that can be played by the certified provers. Second, since redundancy is also a mean to achieve desired safety levels, a second tool chain could be developed. It would bypass Why3 and generate proof obligations directly in the input language of the automatic provers. An approach similar to [7], targetting the SMT-LIB format [2] is a good candidate.

References

[1] Jean-Raymond Abrial (2005): *The B-book - assigning programs to meanings*. Cambridge University Press.

[2] Clark Barrett, Pascal Fontaine & Cesare Tinelli (2015): *The SMT-LIB Standard: Version 2.5*. Technical Report, Department of Computer Science, The University of Iowa. Available at `www.SMT-LIB.org`.

[3] François Bobot, Jean-Christophe Filliâtre, Claude Marché & Andrei Paskevich (2011): *Why3: Shepherd your herd of provers*. In: *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pp. 53–64.

[4] (2012): *The BWare Project*. Available at `http://bware.lri.fr/`.

[5] Sylvain Conchon & Mohamed Iguernelala (2014): *Tuning the Alt-Ergo SMT Solver for B Proof Obligations*. In Yamine Ait Ameur & Klaus-Dieter Schewe, editors: *Abstract State Machines, Alloy, B, TLA, VDM, and Z: 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 294–297, doi:10.1007/978-3-662-43652-3_27.

[6] Jean-François Couchot & T. Hubert (2007): *A Graph-based Strategy for the Selection of Hypotheses*. In: *FTP'07, Int. Workshop on First-Order Theorem Proving*, Liverpool, UK.

[7] David Déharbe, Pascal Fontaine, Yoann Guyot & Laurent Voisin (2012): *SMT Solvers for Rodin*. In: *Proceedings of the Third International Conference on Abstract State Machines, Alloy, B, VDM, and Z*, ABZ'12, Springer-Verlag, Berlin, Heidelberg, pp. 194–207, doi:10.1007/978-3-642-30885-7_14.

[8] D. Delahaye, C. Dubois, C. Marché & D. Mentré (2014): *The BWare Project: Building a Proof Platform for the Automated Verification of B Proof Obligations*. In: *Abstract State Machines, Alloy, B, VDM, and Z (ABZ)*, pp. –, doi:10.1007/978-3-662-43652-3_26.

[9] Michael Leuschel & Michael Butler (2003): *ProB: A Model Checker for B*. In Keijiro Araki, Stefania Gnesi & Dino Mandrioli, editors: *FME 2003: Formal Methods: International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003. Proceedings*, Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 855–874, doi:10.1007/978-3-540-45236-2_46.

[10] David Mentré, Claude Marché, Jean-Christophe Filliâtre & Masashi Asuka (2012): *Discharging Proof Obligations from Atelier B using Multiple Automated Provers*. In Steve Reeves & Elvinia Riccobene, editors: *ABZ - 3rd International Conference on Abstract State Machines, Alloy, B and Z*, *Lecture Notes in Computer Science* 7316, Springer, Pisa, Italy, pp. 238–251, doi:10.1007/978-3-642-30885-7_17. Available at `https://hal.inria.fr/hal-00681781`.

Evaluation of formal IDEs for human-machine interface design and analysis: the case of CIRCUS and PVSio-web

Camille Fayollas[1] Célia Martinie[1] Philippe Palanque[1] Paolo Masci[2]
Michael D. Harrison[2,3] José C. Campos[2] Saulo Rodrigues e Silva[2]

[1]ICS-IRIT, University of Toulouse, Toulouse, France
{fayollas,martinie,palanque}@irit.fr

[2]HASLab/INESC TEC and Universidade do Minho, Braga, Portugal
{paolo.masci,jose.c.campos,saulo.r.silva}@inesctec.pt

[3]Newcastle University, Newcastle upon Tyne, United Kingdom
michael.harrison@newcastle.ac.uk

Critical human-machine interfaces are present in many systems including avionics systems and medical devices. Use error is a concern in these systems both in terms of hardware panels and input devices, and the software that drives the interfaces. Guaranteeing safe usability, in terms of buttons, knobs and displays is now a key element in the overall safety of the system. New integrated development environments (IDEs) based on formal methods technologies have been developed by the research community to support the design and analysis of high-confidence human-machine interfaces. To date, little work has focused on the comparison of these particular types of formal IDEs. This paper compares and evaluates two state-of-the-art toolkits: CIRCUS, a model-based development and analysis tool based on Petri net extensions, and PVSio-web, a prototyping toolkit based on the PVS theorem proving system.

1 Introduction

Use error is a major concern in critical interactive systems. Consider, for example, a pilot calibrating flight instruments before take-off. When calibrating the barometer used to measure the aircraft's altitude, a consistency check should be performed automatically by the cockpit software to help guard against use errors, such as mistyping a value or selecting the wrong units.

New IDEs based on formal methods have been developed by the research community to support the design and analysis of high-confidence human-machine interfaces. Each IDE supports different types of analysis, ranging from functional correctness (e.g., absence of deadlocks and coding errors such as division by zero) to compliance with usability and safety requirements (e.g., assessing the response to user tasks, or the visibility of critical device modes). Choosing the right tool is important to ensure efficiency of the analysis and that the analysis addresses the appropriate safety concerns relating to use. To date, little work has been done to compare and evaluate different formal IDEs for human-machine interface design and analysis, and little or no guidance is available for developers to understand which IDE can be used most effectively for which kind of analysis. This paper describes a first step towards addressing this gap.

Contribution. We compare and evaluate two state-of-the-art formal verification technologies for the analysis of human-machine interfaces: CIRCUS [9], a model-based development and analysis tool that uses Petri net extensions; and PVSio-web [19], a prototyping toolkit based on the PVS theorem prover.

EPTCS ??, 20??, pp. 1–19, doi:10.4204/EPTCS.??.??

The aim of this work is to provide guidance to developers to understand which tool can be used most effectively for which kind of analysis of interactive systems. Both tools have their foundations in existing formal technologies, but are focused towards particular issues relating to the user interface. The capabilities of the two tools are demonstrated in the paper through a common case study based on a critical subsystem in the cockpit of a large civil aircraft. A taxonomy is developed as a result of the comparison that can be used to describe the characteristics of other similar tools.

Organisation. The remainder of the paper is organised as follows. Section 2 illustrates typical features of formal IDEs for the design and analysis of human-machine interfaces, and presents a detailed description of CIRCUS and PVSio-web. Section 3 introduces the common example for comparison of the selected tools, as well as the developed models. Section 4 presents the metrics for comparing the IDEs, and then uses the metrics as a basis for the comparison. Section 5 concludes the paper and presents future directions in which the tools may evolve.

2 The formal modelling and analysis of user interfaces

Formal tools for the modelling and analysis of human-machine interfaces are designed to support multi-disciplinary teams of designers from different engineering disciplines, including human factors engineering (to establish usability requirements, run user studies and interpret compliance), formal methods (to verify compliance of a system design with design requirements), and software engineering (to develop prototypes and software code, e.g., using model-based development methods). Although several tools provide graphical model editors and automated functions for modelling and analysis of interactive elements of a system, different tools are usually complementary, as they support different levels of description, and different types of analysis, ranging from micro-level aspects of human-machine interaction, e.g., aspect and behaviour of user interface widgets, to the analysis of the wider socio-technical system within which the interactive system is used.

In the present work, we compare two state-of-the-art formal tools developed by two different research teams: CIRCUS [9], a toolkit for model-based development of interactive systems; and PVSio-web [19], a toolkit for model-based development of user interface software. Both build on tools that have been developed more generally for model based design and software engineering, extending them with features that are particularly useful when considering the human-machine interface or the wider socio-technical system.

Other formal tools that could be used (and in some cases have been used) for the analysis of human-machine interfaces exist. They offer functionalities that complement those of CIRCUS and PVSio-web. The evaluation of these other tools is not within the scope of this paper, although a brief overview can be found in Section 2.3.

2.1 CIRCUS

CIRCUS, which stands for Computer-aided-design of Interactive, Resilient, Critical and Usable Systems, is an IDE for the formal verification of the system's behaviour as well as the analysis of compatibility between the user's task and the system's behaviour. CIRCUS includes three tools:

- **HAMSTERS** (Human-centred Assessment and Modelling to Support Task Engineering for Resilient Systems) is a tool for editing and simulating task models. The tool can be used to ensure consistency, coherence, and conformity between assumed or prescribed user tasks and the sequence of actions necessary to operate interactive systems [3]. The notation used in the tool makes it possible to structure

users' goals and sub-goals into hierarchical task trees. Qualitative temporal relationships among tasks are described by operators. Various notational elements support modelling of specialised task types, explicit representations of data and knowledge, device descriptions, genotypes and phenotypes of errors, and collaborative tasks.

- **PetShop** (Petri Net workshop) is a tool for creating, editing, simulating and analysing system models using the ICO (Interactive Cooperative Objects) notation [23, 17]. The ICO notation allows developers to specify the behaviour of interactive systems. The notation uses Petri Nets for describing dynamic behaviours, and uses object-oriented concepts (including dynamic instantiation, classification, encapsulation, inheritance and client/server relationships) to describe structural or static aspects of the system.

- **SWAN** (Synergistic Workshop for Articulating Notations) is a tool for the co-execution of PetShop models and HAMSTERS models [3]. The tool allows developers to establish correspondences between system behaviours and tasks, and perform automated system testing by means of co-execution [5].

2.2 PVSio-web

PVSio-web is a toolkit for model-based development of user interface software. The toolkit is based on and extends an established theorem prover, PVS [24], providing a graphical environment for constructing, visualising and analysing formal models of user interface software. PVSio-web has three main components:

- **Prototype Builder and Simulator**. This tool allows developers to create device prototypes based on formal models, and run them within a Web browser. The visual aspect of the prototype uses an interactive picture of the device. Developers create programmable areas over the picture to identify input widgets (e.g., buttons) and output widgets (e.g., displays, LEDs). The tool automatically translates user actions over input widgets (e.g., button presses) into PVS expressions that can be evaluated within PVSio [22], the native PVS component for animating executable PVS models. The Simulator tool executes PVSio in the background, and the effects of the execution are automatically rendered using the output widgets of the prototype to closely resemble the visual appearance of the real system in the corresponding states.

- **Emucharts Editor**. This tool facilitates the creation of formal models using visual diagrams known as Emucharts. These diagrams are based on Statecharts [14]. The tool allows developers to define the following design elements: states, representing the different modes of the system; state variables, representing the characteristics of the system state; and transitions, representing events that change the system state. The tool incorporates a model generator that translates the Emucharts diagram into executable PVS models automatically. The model generator also supports the generation of other different formal modelling languages for interactive systems, including VDM [18], MAL [6], and PIM [4], as well as executable code (MISRA-C).

- **The PVS back-end**. This includes the PVS theorem prover and the PVSio environment for model animation. The back-end is used for formal analysis of usability-related properties of the human-machine interface model, such as consistency of response to user actions and reversibility of user actions.

2.3 Other tools

MathWorks Simulink [20] is a commercial tool for model-based design and analysis of dynamic systems. It provides a graphical model editor based on Statecharts, and functions for rapid generation of realistic prototypes. SCR [13] is a toolset for formal analysis of system requirements and specifications. Using SCR, it is possible to specify the behaviour of a system formally, use visual front-ends to demonstrate the system behaviour based on the specifications, and use a set of formal methods tools for the analysis of system properties. SCADE and IBM's Rational Statemate are two commercial tool sets for model-based development of interactive systems. The tool sets provide, among other features, rapid prototyping, co-simulation, and automated testing. Formal verification is supported by these tools, but is limited to the analysis of coding errors such as division-by-zero. Use-related requirements and tasks can be analysed only using simulation and testing. IVY [7] is a workbench for formal modelling and verification of interactive systems. The tool provides developers with standard property templates that capture usability concerns in human-machine interfaces. The core of the IVY verification engine is the NuSMV model checker. A graphical environment isolates developers from the details of the underlying verification tool, thereby lowering the knowledge barriers for using the tool. The particular tools that are of interest in the design and analysis of interactive systems enable the analysis of user activities, with a focus on what users do in terms of what they perceive about the systems and the actions they perform. Furthermore an important requirement for such tools is that the means of analysis and their results should be accessible to team members without a background in formal techniques, or even software development techniques.

3 Case study and IDE showcase

The case study for comparing the selected tools is based on a subsystem of the Flight Control Unit (FCU) of the Airbus A380. It is an interactive hardware panel with several different buttons, knobs, and displays. The FCU has two main components: the Electronic Flight Information System Control Panel (EFIS CP), for configuring the piloting and navigation displays; and the Auto Flight System Control Panel (AFS CP), for setting the autopilot state and parameters.

In future cockpits, the interactive hardware elements of the FCU panel might be replaced by an interactive graphical application rendered on displays. This graphical software (hereafter, referred to as FCU Software) will provide the same functionalities as the corresponding hardware elements. This graphical software will be displayed on one of the screens in the cockpit. Pilots will interact with the FCU Software via the Keyboard and Cursor Control Unit (KCCU) that integrates keyboard and track-ball (see Figure 1).

The present paper illustrates how CIRCUS and PVSio-web can be used to create models and prototypes of the FCU Software. Developers can explore design options and analyse requirements for these future generation FCUs using these formal IDEs for model-based development of human-machine interfaces. To keep the example simple, we focus further and analyse the EFIS CP. This component includes most of the fundamental interactive elements of the FCU.

3.1 Description of the system and its use

A close up view of the EFIS CP is shown in the rightmost picture of Figure 1. The left panel of the EFIS CP window is dedicated to the configuration of the barometer settings (upper part) and of the Primary Flight Display (lower part). The right panel is dedicated to the configuration of the Navigation Display. The top part provides buttons for displaying navigation information on the cockpit displays.

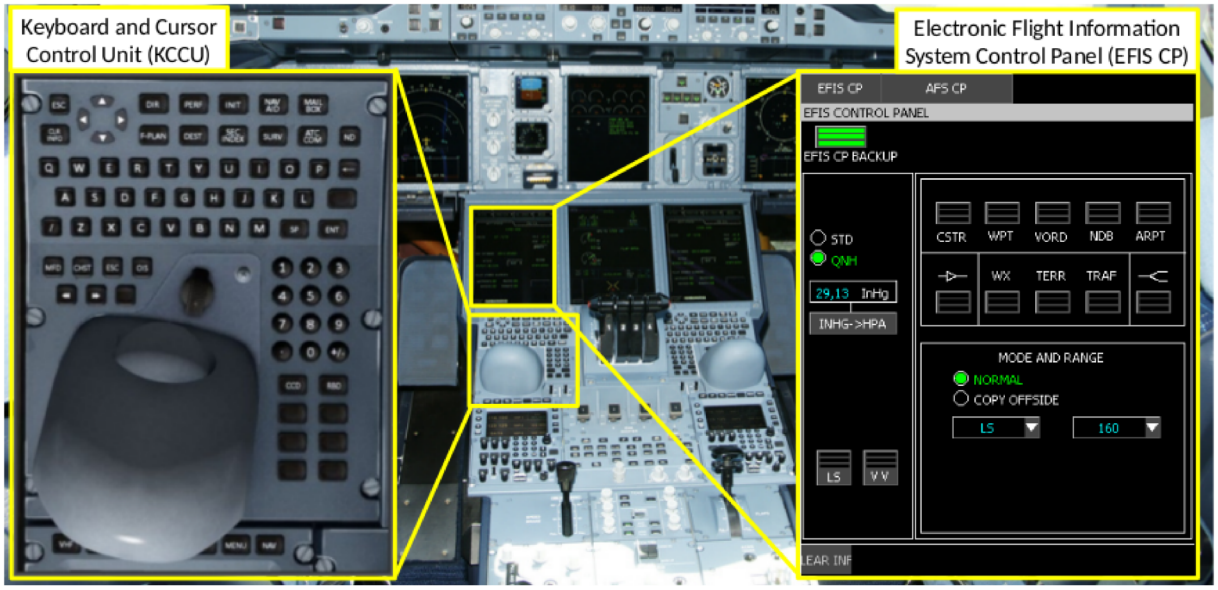


Figure 1: Keyboard and Cursor Control Unit and Flight Control Unit Software.

The ComboBoxes at the bottom allow the pilot to choose display modes and scale.

In this work, we focus more particularly on the configuration of the barometric settings (upper part of the left panel). This panel is composed of several widgets: two CheckButtons enable pilots to select either Standard (STD) or Regional Pressure Settings (QNH) mode. When in QNH mode, a number entry widget (EditBoxNumeric) enables pilots to set the barometric reference. Finally, a button (PushButton) enables pilots to switch the barometer units between inches of mercury (inHg) and hectopascal (hPa). When switching from one unit to the other, a unit conversion is triggered, and the barometer settings value on the display is updated accordingly. When the barometer unit is inHg, the valid range of values is [22, 32.48]. When the unit is hPa, the valid range is [745, 1100]. If the entered value exceeds the valid value range limits, the software automatically adjusts the value to the minimum (when over-shooting the minimum valid value) or the maximum (when overshooting the maximum valid value).

When starting the descent (before landing), pilots may be asked to configure the barometric pressure to the one reported by the airport. The barometric pressure is used by the altimeter as an atmospheric pressure reference in order to process correctly the plane altitude. To change the barometric pressure, pilots select QNH mode, then select the pressure unit (which depends on the airport), and then edit the pressure value in the EditBoxNumeric.

3.2 Modelling and analysis using CIRCUS

A prototype user interface was developed in CIRCUS that captures the functionalities of the FCU Software. The workflow for the modelling and analysis of this interactive system includes six main steps that are briefly detailed below.

Step 1: Task analysis and modelling. This step describes user activities. It identifies goals, tasks, and activities that are intended to be performed by the operator. For example, the task “Perform descent preparation”, is decomposed into several HAMSTERS models represented as subroutines, components and instances of components [10]. Due to space constraints, we only describe an extract of the model in Figure 2.

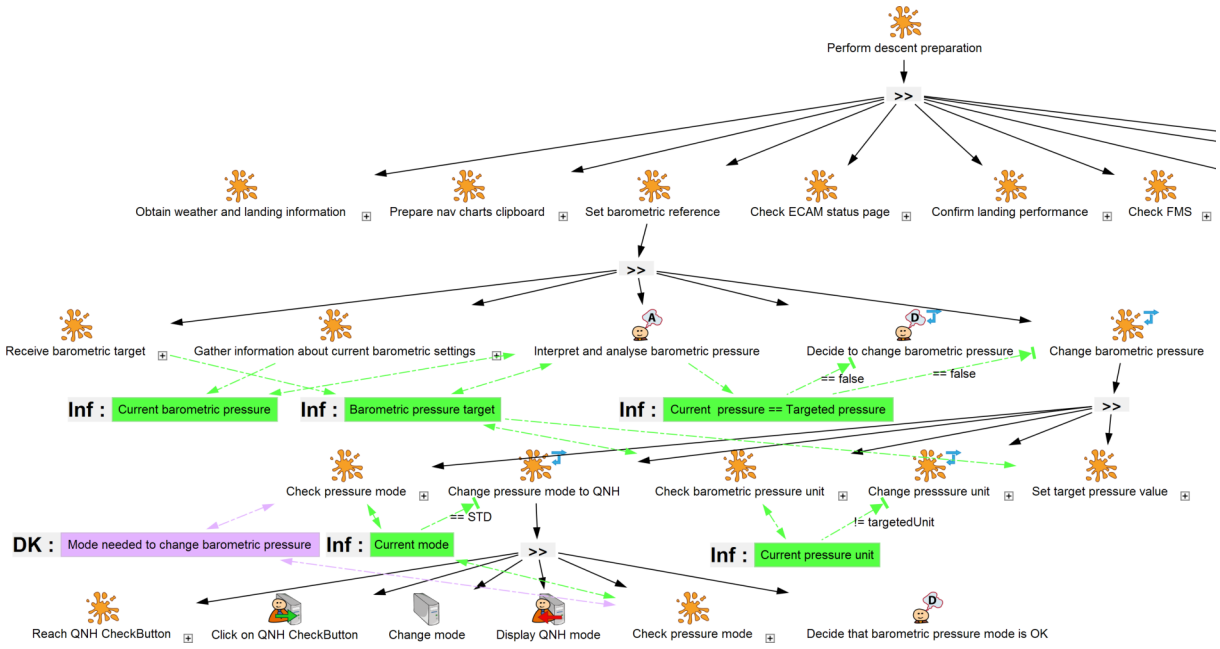


Figure 2: Extract of the task model for the task “Perform descent preparation”.

A simplified version of the task model is described as the abstract task “Perform descent preparation” in the first row of Figure 2. The second row refines this task into several abstract sub-tasks (e.g., “Obtain weather and landing information”). Each one of these abstract tasks corresponds to a step of the operational procedure that is intended to be performed by the flight crew when preparing the descent phase [1].

In the present paper, we focus on the “Set barometric reference” abstract task, refined in the third row. The task is decomposed as follows: the pilot receives the new barometric target (“Receive barometric target” abstract task) and remembers the corresponding information (“Barometric pressure target”). The pilot then needs to gather information about the current barometer settings (“Gather information about current barometric settings”), thus remembering a new piece of information (“Current barometric pressure”). The pilot then needs to compare the two values that have been received in the previous two steps (“Interpret and analyse barometric pressure” cognitive analysis task) creating the “Current pressure == Targeted pressure” information. If the targeted pressure is different from its current value the pilot decides to change the pressure (“Decide to change barometric pressure” cognitive decision task) and change it (“Change barometric pressure” abstract task).

The fourth row of Figure 2 refines the “Change barometric pressure” abstract task as follows: the pilot must first check the pressure mode (“Check pressure mode” abstract task), switch to QNH mode if the current mode is STD (“Change pressure mode to QNH” abstract task), check the unit and change it if needed (“Check barometric pressure unit” and “Change pressure unit” abstract tasks) and finally set the new pressure value (“Set target pressure value” abstract task).

Finally, the fifth row refines the “Change pressure mode to QNH” abstract task. This task is performed by the pilot if the current mode is STD. In this task, the pilot first reaches the QNH checkButton (“Reach QNH CheckButton” abstract task). Then s/he clicks on it (“Click on QNH CheckButton” interactive input task). The system then changes the mode and displays the new state (“Change mode” system task and “Display QNH mode” interactive output task). The pilot can then check that the new pressure

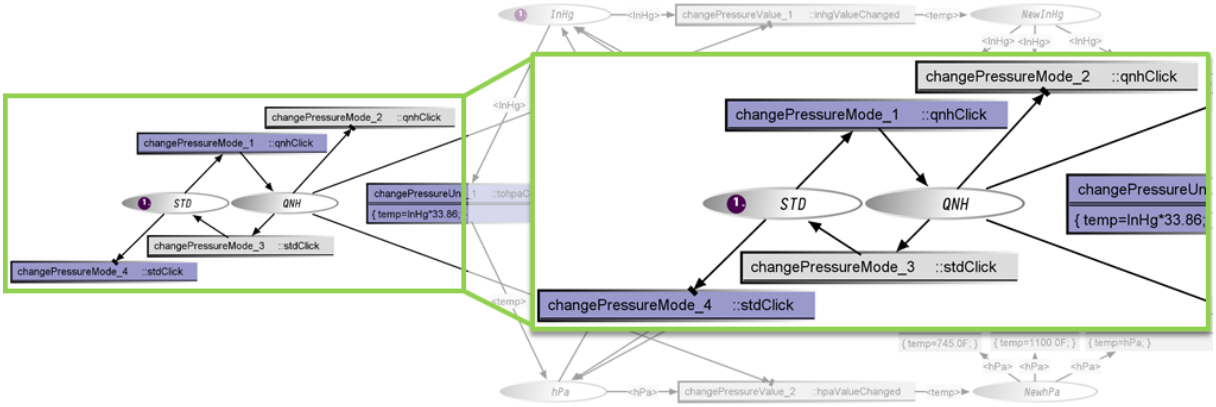


Figure 3: ICO model of the barometer settings behaviour.

mode is the good one ("Check pressure mode" abstract task and "Decide that barometric pressure mode is OK" cognitive decision task). It is important to note that this task model is detailed both in terms of user task refinement (e.g., cognitive task analysis) to allow the analysis of workload and performance (see Step 3 below); and in terms of interactive task refinement (see, for instance, the refinement of the "Change pressure mode to QNH" abstract task which includes the "Click on QNH Checkbutton" interactive input task) to allow the compatibility assessment between the task model and the behavioural model of the system (see Step 6 below).

Step 2: Workload and performance analysis. As presented in Figure 2, the HAMSTERS notation and tool enable human task refinement. It makes it possible to differentiate between cognitive, motor, and perception tasks as well as representing the knowledge and information needed by the user to perform a task. The refinement allows the qualitative analysis of user tasks, workload and performance. For example, the number of cognitive tasks and information that pilots need to remember may be effective indicators for assessing user workload [9]. This kind of analysis can be used to reason about automation design [16].

Step 3: User interface look and feel prototyping. This step aims at developing the user interface look and feel. A result of this step is described in the screen-shot of the EFIS CP presented in Figure 1. The widgets are organised in a style that is compatible with the library defined by the ARINC 661 standard [2].

Step 4: User interface formal modelling. The behaviour of the user interface is specified using ICO models. The behaviour of the barometer settings part of the EFIS CP user interface is represented by the ICO model presented in Figure 3. The left part of this model (that has been enlarged) is dedicated to the pressure mode. As described in Section 3.1, the pressure mode can be in two different mutually exclusive states: STD and QNH. The user can switch from one mode to the other one by clicking either STD or QNH CheckButton (clicking on a CheckButton while already in the corresponding mode is also possible but will have no impact on the pressure mode). This behaviour is defined by the enlarged part of the ICO model presented in Figure 3. The state of the pressure mode is represented by the presence of a token within "QNH" or "STD" places (in Figure 3, place "STD" holds a token meaning that the current pressure mode is STD). Transitions "changePressureMode_1" and "changePressureMode_2" (resp. "changePressureMode_3" and "changePressureMode_4") correspond to the availability of the "qnhClick" (resp. "stdClick") event: when one of these two transitions is enabled, the "qnhClick" event is available (thus enabling the QNH CheckButton). The "changePressureMode_1" transition therefore makes it possible to switch from STD pressure mode to QNH pressure mode as a result of clicking on

the QNH CheckButton. The "changePressureMode_2" transition allows the user to click on the QNH CheckButton while in QNH pressure mode without any impact on the pressure mode. The right part of the ICO model presented in Figure 3 (behind the enlarged part of the model) allows pressure to be changed. While we only present here a part of the ICO model describing the behaviour of the EFIS user interface, it is important to note that the ICO notation has also been used to describe the behaviour of the widgets and the window manager. The ICO models can be validated using the simulation feature.

Step 5: Formal analysis. The PetShop tool provides the means to analyse ICO models by the underlying Petri net model [26] using static analysis techniques as supported by the Petri net theory [25]. The ICO approach is based on high level Petri nets. As a result the analysis approach builds on and extends these static analysis techniques. Analysis results must be carefully taken into account by the analyst as the underlying Petri net model can be quite different from the ICO model. Such analysis has been included in CIRCUS and can be interleaved with the editing and simulation of the model, thus helping to correct it in a style that is similar to that provided by spell checkers in modern text editors [8]). It is thus possible to check well-formedness properties of the ICO model, such as absence of deadlocks, as well as user interface properties, either internal properties (e.g., reinitiability) or external properties (e.g., availability of widgets). Note that it is not possible to express these user interface properties explicitly — the analyst needs to express these properties as structural and behavioural Petri net properties that can be then analysed automatically in PetShop.

The analysis of the enlarged part of the ICO model presented in Figure 3 allows developers to check that, whatever action is taken, the pair of places "STD" and "QNH" will always hold one (and only one) token, exhibiting the mutual exclusion of the two states. Transitions connected to these places correspond to the availability of two events "qnhClick" and "stdClick", and therefore it can be demonstrated that these events will remain available whatever action is triggered. Lastly, there are two transitions in the model that correspond to the event "qnhClick" (transitions "changePressureMode_1" and "changePressureMode_2"). This could potentially lead to non-determinism in the model. However, as "changePressureMode_1" has place "STD" as input place and "changePressureMode_2" has "QNH" place as input place, non-determinism is avoided due to the mutual exclusive marking of these places.

Step 6: Compatibility assessment between task models and user interface models. This step aims at guaranteeing that the task model and the formal model of the user interface behaviour are complete and consistent together (thus helping to guarantee that procedures followed by the operators are correctly supported by the system). A correspondence editing panel is used to establish the matching between interactive input tasks (from the task model) with system inputs (from the system model) and between system outputs (from the system model) with interactive output tasks (from the task model). The co-execution part of the SWAN tool provides support for validation as it makes it possible to find inconsistencies between the two models, e.g., sequences of user actions allowed by the system model and forbidden by the task model, or sequences of user actions that should be available but are not because of inadequate system design. The SWAN tool also provides support for automated scenario-based testing of an interactive application [5]

3.3 Modelling and analysis using PVSio-web

The focus of the PVSio-web analysis is the interaction logic of the EFIS data entry software. Here, we describe the modelling and analysis workflow supported by the tool, and highlight the main characteristics of the developed models (the full description is included as an example application in the PVSio-web tool distribution [19]).

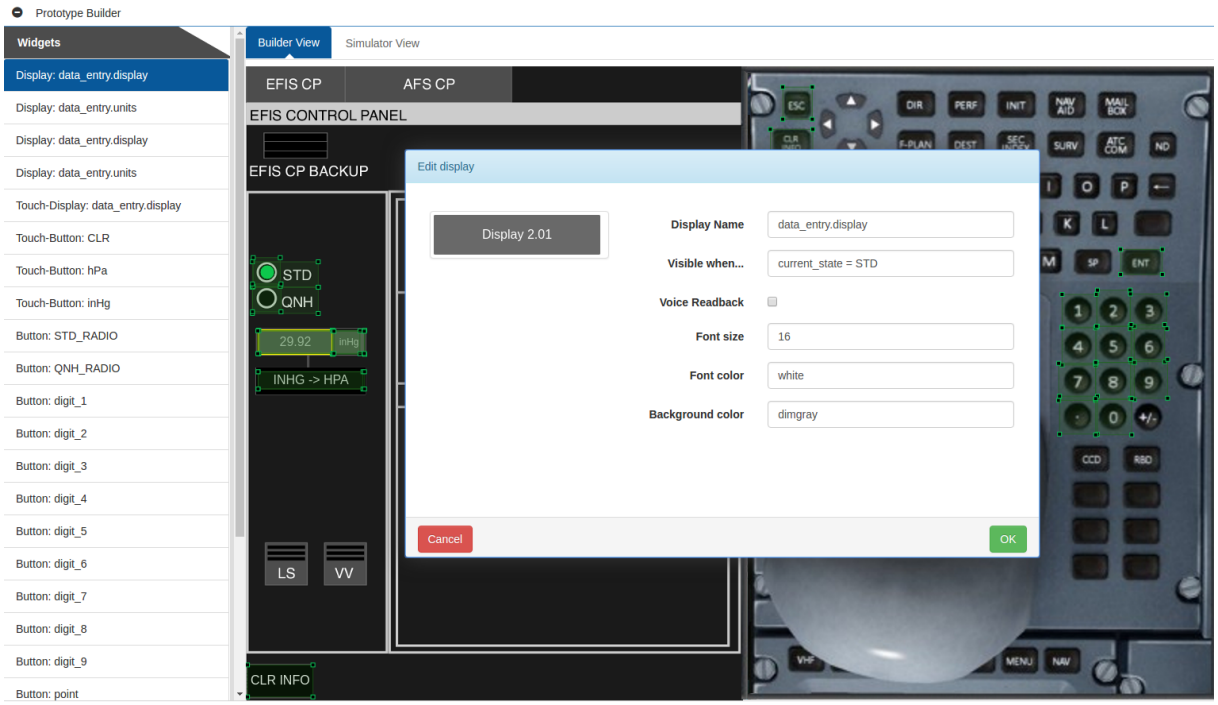


Figure 4: FCU Software prototype developed in the PVSio-web Prototype Builder. Shaded areas over the picture identify interactive system elements.

Step 1: Define the visual appearance of the prototype. The visual aspect of the prototype is based on a picture of the EFIS panel. The PVSio-web Prototype Builder is used to create the visual appearance of the prototype and is defined using the Prototype Builder. A picture of the EFIS panel and KCCU are loaded in the tool, and interactive areas are created over relevant buttons and display elements (see Figure 4). Fifteen input areas were created over the picture of the Keyboard and Cursor Control Unit, to capture user actions over the number pad keys, as well as over other data entry keys (ENT, CLR, ESC, and the units conversion button). Four display elements were created for rendering relevant status variables of the PVS model: a touchscreen display element handles user input on the EditBoxNumeric for entering the barometer pressure value; two display elements render the STD and QNH CheckButtons; an additional display element renders the pressure units.

Step 2: Define the behaviour of the prototype. The prototype is driven by a PVS model that includes an accurate description of the following features of the system: the modal behaviour of the data entry system; the numeric algorithm for units conversion; the logic for interactive data entry; and the data types used for computation (double, integer, Boolean). Modelling patterns were used to guide the development of the models (some of these patterns are described in [11]). The model was developed using, in combination, the PVSio-web Emucharts Editor and the Emacs-based model editor of PVS. The Emucharts editor allowed us to create a statechart-based diagram that can be automatically translated into PVS models. The Emacs-based model editor was used to build a library function linked to the Emucharts diagram to improve modelling efficiency. The developed Emucharts (shown in Figure 5) includes the following elements: 3 states (STD, QNH, and EDIT PRESSURE) representing three different modes of operation; 25 transitions, representing the effect of user actions on the Keyboard and Cursor Control Unit when adjusting the barometer settings, and internal timers handling timeouts due to inactivity during interactive

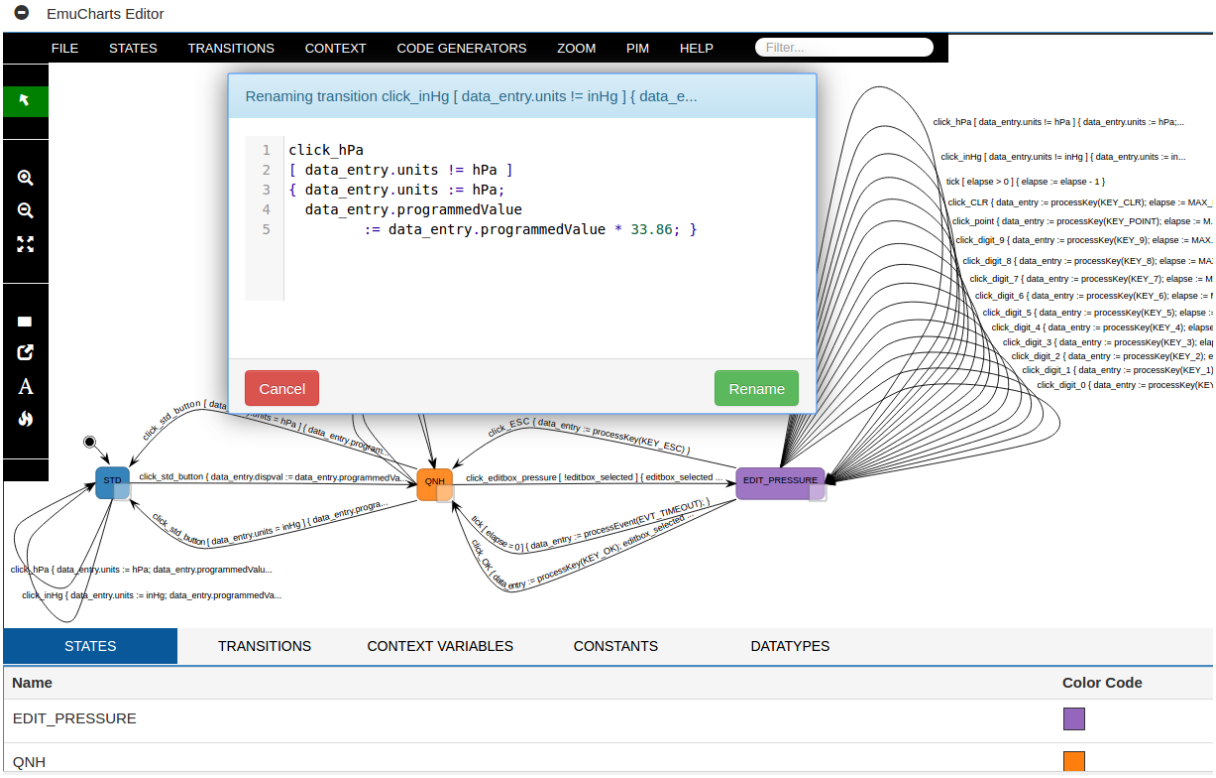


Figure 5: Emucharts model of the FCU Software created in the Emucharts Editor.

data entry; and 9 status variables, representing the state of the system (units, display value, programmed value, etc.). The library function, ProcessKey, is used within the Emucharts diagrams to define the effect on state variables of transitions associated with key presses.

Step 3: Model validation. This analysis ensures internal consistency of the model, as well as checking accuracy with respect to the real system. Internal consistency is assessed by discharging proof obligations (called type-check-conditions) automatically generated by the PVS theorem prover. These proof obligations check coverage of conditions, disjointness of conditions, and correct use of data types. For the developed Emucharts model, PVS generated 22 proof obligations, all of which were discharged automatically by the PVS theorem prover. Accuracy of the model is assessed by using the prototype to engage with Airbus cockpit experts. Experts can press buttons of the prototype, and watch the effect of interactions on the prototype displays. By this means it is possible to check that the prototype behaviour resembles that of the real system.

Step 4: Formal analysis. The prototype and the PVS theorem prover are used in combination to analyse the model. The prototype is used to perform a lightweight formal analysis suitable to establish a common understanding within a multidisciplinary team of the correct interpretation of safety requirements and usability properties. This analysis consists in the execution of sample input key sequences demonstrating scenarios where a given requirement is either satisfied or fails. This initial lightweight analysis based on test cases is extended to a full formal analysis using the PVS theorem prover, to check that requirements and properties of the model are satisfied for all input key sequences in all reachable model states. To perform this full analysis, PVS theorems need to be defined that capture the requirements and properties. They are expressed using structural induction and a set of templates described in [12]. An example

property that can be analysed is *consistency of device response to user actions*. The consistency property is motivated by the fact that users quickly develop a mental model that embodies their expectations of how to interact with a user interface. Because of this, the overall structure of a user interface should be consistent in its layout, screen structure, navigation, terminology, and control elements. Example consistency properties are: a designated set of function buttons always change the mode; a further set of keys, for example concerned with number entry, will always change the barometric variable relevant to the mode but do not change the mode; an *enter* key always changes the relevant parameter when in the relevant mode; an *escape* key ensures that the value set in the mode is discarded and the barometric value reverts to the value it had when it entered the mode.

In PVS, the consistency template is formulated as a property of a group of actions $A_c \subseteq \wp(S \to S)$, or it may be the same action under different modes, requiring that all actions in the group have similar effects on specific state variables selected using a filter. The relation $consistent : C \times C \to T$ connects a filtered state, before an action occurs (captured by $filter\_pre : S \times MS \to C$), with a filtered state after the action (captured by $filter\_post : S \times MS \to C$). The description of the filters and the *consistent* relation specify the consistency across states and across actions. Here *MS* is defined to be a set of modes. Two modes are relevant here. A set of modes not defined includes the mode that allows the entry of the barometer value. Within the barometer entry mode are two modes that relate to the different units that can be used to enter the barometric values, defined as: inHg and hPa. A general notion of consistency assumes that the property is restricted to a set of states focused by a guard: $guard : S \times MS \to T$. This guard may itself be limited by a mode. The general consistency template can therefore be expressed as:

Consistency

$$\forall a \in A_c \subseteq \wp(S \to S), s \in S, m \in MS :$$
$$guard(s,m) \wedge consistent(filter\_pre(s,m), filter\_post(a(s),m)$$

Two examples are now used to illustrate the use of the consistency template. The first is that a set of actions never change the barometric entry mode. The pre_filter and $post\_filter$ both extract the barometric entry mode, and are of the form `filter_baro(st: state): UnitsType = Units(st)`. This property relates directly to modes and therefore the mode parameter can be omitted in the filter definition. The set of actions determined by `state_transitions_actions` which encompasses the set of transitions as determined by the enabled actions in the barometric mode. In summary, the *consistent* relation in this case is equality and the theorem that instantiates the consistency template is:

```
modeinvariant: THEOREM FORALL (pre, post: state):
 state_transitions_actions(pre, post) => (filter_baro(pre) = filter_baro(post))
```

On the other hand the action $click_{hPa}$ always changes the entry mode. So here *consistent* is inequality.

```
alwayschgmode: THEOREM FORALL (pre, post: state):
 (post = click_hPa(pre) AND guard_baro(pre))
                                        => filter_baro(pre) /= filter_baro(post)
```

4 Tool comparison

In this section, we first present the criteria that were identified to compare the characteristics and functionalities of the two tools. These criteria form a basis for the comparison of these two tools.

4.1 Comparison criteria

We identified 22 criteria suitable to compare the characteristics and functionalities of the two tools. These criteria are general, and can be used as a reference to define a taxonomy suitable to classify and compare other similar formal IDEs for user interface modelling and analysis. These criteria are divided in four categories:

- **General aspects of the tools**
  1. Scope/purpose of the tool within the development process, e.g., requirements analysis, prototyping, verification.
  2. Tool features, e.g., modelling of user tasks and goals, analysis of usability properties, simulation of user tasks.
  3. Tool extensibility, e.g., to model systems from different application domains, or to perform a different type of analysis
  4. Prerequisites and background knowledge, e.g., distributed systems, object oriented languages, Petri Nets, task modelling, PVS.
  5. IDE instance and principle, e.g., Eclipse plugin, Web, Netbeans API
  6. IDE availability, e.g., snapshot, demo, downloadable, open source.
- **Modelling features**
  7. Notation names, e.g., ICO, HAMSTERS, Emucharts, PVS.
  8. Notation instance, e.g., Petri Net, state machines, higher-order logic.
  9. Notation paradigm, e.g., event-based, state-based, declarative.
  10. Structuring models, e.g., object-oriented, functional, component-based.
  11. Model editing features, e.g., textual, visual, autocompletion support.
  12. Suggestions for model improvements, e.g., strengthening of pre-conditions.
- **Prototyping features**
  13. Support for prototype building, e.g., visual editor, library of widgets.
  14. Execution environment of the prototype, e.g., Java virtual machine, Javascript execution environment.
  15. User interface testing, e.g., automatic generation of input test cases.
  16. Human-machine interaction techniques, e.g., Pre-WIMP (input dissociated from output), WIMP, post-WIMP, tangible, multimodal.
  17. Code generation, e.g., C, C++, Java.
- **Analysis of human-machine interaction**
  18. Verification type, e.g., functional verification, performance analysis, hierarchical task analysis;
  19. Verification technology, e.g., theorem proving, static analysis.
  20. Scalability of the analysis, e.g., illustrative examples, industrial size.
  21. Support for the analysis of the wider socio-technical system.
  22. Related development process, e.g., user centered design, waterfall development process, agile development.

4.2 CIRCUS and PVSio-web comparison

In this section, we discuss, following the four categories of criteria identified above, the comparison of CIRCUS and PVSio-web. A detailed assessment of all the criteria presented above is presented in tabular form in the Appendix.

General aspects of the tools. From a high-level perspective, the scope of CIRCUS and PVSio-web is the formal development of user interfaces. Both tools support modelling and analysis of the interaction logic of the user interface software. However, the two tools offer different modelling and analysis technologies that are tailored to support two different (and complementary) styles of assessment of user interfaces. CIRCUS supports explicit modelling of user tasks and goals, allowing developers to simulate user tasks and check their compatibility with the interactive behaviour of the system. PVSio-web, on the other hand, supports explicit modelling of general usability and safety properties, facilitating the assessment of compliance of a user interface design with design guidelines and best design practices (e.g., according to standards or regulatory frameworks). Whilst a certain level of background knowledge is needed to use the tools effectively, basic knowledge about Petri nets and task models (for CIRCUS) and state machines and state charts (for PVSio-web) is already sufficient to get started with illustrative examples. This is extremely useful to reduce the typical knowledge barriers faced by novice users. The two IDEs are developed using standard technologies supported by multiple platforms (Netbeans Visual API for CIRCUS, Web technologies for PVSio-web), and can be executed on any standard desktop/laptop computer.

Modelling features. Both tools provide powerful graphical IDEs designed to assist developers in the creation of formal models. CIRCUS uses specialised graphical notations and diagrams: the ICO notation is used for building system models; the HAMSTERS notation is used for describing user tasks. ICOs are based on object-oriented extensions to Petri nets, and support both event-based and state-based modelling. HAMSTERS is a procedural notation. The complexity of models is handled using information hiding (as in object-oriented programming languages), and component-based model structures. This facilitates the creation of complex models, as well as the implementation of editing features that are important for developers, such as auto-completion of models and support for parametric models. The use of specific notations, however, limits the ability of developers to import external models created with other tools, or export CIRCUS models to other tools. PVSio-web, on the other hand, uses modelling patterns to support the modelling process. Developers can use either a graphical notation (Emucharts diagrams, or a textual notation (PVS higher-order logic), or a combination of both, to specify the system model. This has many benefits: software developers that are familiar with Statecharts can build models using a language that is familiar for them, and gradually learn PVS modelling by examples, checking how the Emucharts model translates into PVS; Emucharts models can be translated into popular formal modelling languages different than PVS (e.g., VDM); expert PVS users can still develop entire models using PVS higher-order logic only, and software developers can import these PVS models as libraries, thus facilitating model re-use. The main drawback is that the current implementation of Emucharts lacks mechanisms for model optimisation (e.g., a battery of similar PVS functions is generated instead of a single function with a parameter), and technical skills are necessary to understand model improvements suggested by the tool (through the PVS type-checker).

Prototyping features. Both IDEs provide a visual editor for rapid generation of prototypes supporting a range of interaction styles, including: graphical user interfaces with windows, icons, menus, and pointer (WIMP); user interfaces with physical buttons (pre-WIMP); touchscreen-based user interfaces (post-WIMP); and multi-modal user interfaces (e.g., providing both visual and auditory feedback). Both tools promote the use of the Model-View-Controller (MVC [15]) paradigm, with a clear separation between

the visual appearance of the prototype and the logic behaviour. Whilst prototypes developed with the two IDEs share these similarities, prototype building and implementation is substantially different in the two IDEs. CIRCUS prototypes are developed in Java (for their visual appearance) and in ICO models (for their behaviour). Developers can define their own widgets library. For example, for the case study presented in Section 3, we created a library of widgets whose visual aspect and behaviour is compatible with that described in the ARINC 661 standard. PVSio-web prototypes are developed in JavaScript, and their behaviour is defined by a PVS executable model. Rapid prototyping is enabled by a lightweight building process where the visual aspect of the prototype is defined by a picture of the real device, virtually reducing to zero the time and effort necessary to define the visual appearance of the prototype. Initial support for code generation is also available for MISRA-C, for behavioural models developed using Emucharts [21]. A specialised tool (Prototype Builder) is provided with the IDE, to facilitate the identification of interactive areas over the picture, and to link these areas to the PVS model. The current implementation of the Prototype Builder supports only the definition of push buttons and digital display elements, and developers need to edit a JavaScript template manually to introduce more sophisticated widgets (e.g., knobs, graphical displays, etc.). Integration of these more sophisticated widgets in the Prototype Builder is currently under development.

Analysis of human-machine interaction. Multiple verification technologies are used in the two IDEs to enable the efficient analysis of human-machine interaction. Both tools build on established formal methods technologies, and enable lightweight formal analysis based on simulation and testing. CIRCUS implements static analysis techniques from Petri nets theory to perform automatic analysis of well-formedness properties of the model (absence of deadlocks, token conservation), and of basic aspects of the interactive system design (e.g., reinitiability of the user interface and availability of widgets). Simulation is used for functional analysis and quantitative assessment of the system. Either direct interaction with the prototype and automated execution of task models can be used during simulations. Properties verified by this means include: compliance with task models; statistics about the total number of user tasks, and estimation of the cognitive workload of the user based on the types of human-machine interactions necessary to operate the system. PVSio-web uses the standard PVS theorem proving system to analyse well-formedness properties of the model (coverage of conditions, disjointness of conditions, and correct use of data types). Usability and safety requirements can be verified using both lightweight formal verification and full formal verification. Lightweight verification is based on interactive simulations with the prototypes. User interactions can be recorded and used later as a basis for automated testing in a way similar to the way task models are used in CIRCUS. Full formal verification is carried out in the PVS theorem prover, and is partially supported by property templates capturing common usability and safety requirements described in the ANSI/AAMI/IEC HF75 usability standard. Although the full formal analysis is in general not fully automatic, the combined use of property templates and modelling patterns usually leads to proof attempts where minimal human intervention is necessary to guide the theorem prover (typically, for case-splitting and instantiation of symbolic identifiers). Proof tactics for full automatic verification of a standard battery of property templates are currently under development. Dedicated front-ends presenting verification results in a form accessible to human factors specialists are also being investigated.

5 Conclusion and perspectives

In this paper, we presented a first step towards providing guidance to developers to understand which formal tool can be used most effectively for which kind of analysis of interactive systems. This is

achieved through the identification of 22 criteria enabling the characterisation of IDEs for interactive systems formal prototyping and development. These criteria have been used to compare two state-of-the-art formal tools developed by two different research teams: CIRCUS, a toolkit for model-based development of interactive systems; and PVSio-web, a toolkit for model-based development of user interface software based on the PVS theorem proving system. In order to assess all the criteria, we modelled and analysed a case study from the avionics domain using these two tools. The result of this comparison led to the conclusion that the two studied tools are complementary rather than competitive tools. Whilst they have roughly the same scope (formal development of user interfaces), these two tools enable different kinds of modelling and analysis. For instance, CIRCUS supports explicit modelling of user tasks and goals, allowing developers to simulate user tasks and check their compatibility with the interactive behaviour of the system while PVSio-web supports explicit modelling of general usability and safety properties, facilitating the assessment of compliance of a user interface design with design guidelines and best design practices. These two analysis styles are complementary, and both provide important insights about how to develop high-confidence user interfaces. Based on this understanding, we are now developing means to integrate the two IDEs, to enable new powerful analysis features, such as automated scenario-based testing of user interfaces [5]. The envisioned integration is introduced at two levels: at the modelling level, developing PVSio-web extensions for importing/translating HAMSTERS task models into PVS models and properties; and at the simulation level, building CIRCUS extensions for co-execution of task models and PVSio-web prototypes. Additional extensions under development for the two toolkits include: modelling patterns for describing human-machine function allocation; proof tactics and complementary use of different verification technologies for improved automation of usability and safety properties; innovative front-ends for inspecting formal proofs supporting safety and usability claims of user interfaces; and widgets libraries for different application domains.

Acknowledgment. This work is partially supported by: Project NORTE-01-0145-FEDER-000016, financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF); Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) PhD scholarship.

References

[1] SAS Airbus (2016): *Airbus A380 Flight Crew Operating Manual*. `http://www.airbus.com/`.

[2] Airlines Electronic Engineering Committee (2002): *ARINC 661 specification: Cockpit Display System Interfaces To User Systems*. Aeronautical Radio Inc.

[3] Eric Barboni, Jean-François Ladry, David Navarre, Philippe Palanque & Marco Winckler (2010): *Beyond Modelling: An Integrated Environment Supporting Co-execution of Tasks and Systems Models*. In: *Proceedings of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '10, ACM, pp. 165–174, doi:10.1145/1822018.1822043.

[4] Judy Bowen & Steve Reeves (2013): *Modelling Safety Properties of Interactive Medical Systems*. In: *Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '13, ACM, pp. 91–100, doi:10.1145/2494603.2480314.

[5] José C. Campos, Camille Fayollas, Célia Martinie, David Navarre, Philippe Palanque & Miguel Pinto (2016): *Systematic Automation of Scenario-based Testing of User Interfaces*. In: *Proceedings of the 8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, EICS '16, ACM, New York, NY, USA, pp. 138–148, doi:10.1145/2933242.2948735.

[6] José C. Campos & Michael D. Harrison (2001): *Model Checking Interactor Specifications*. Automated Software Engineering. 8(3-4), pp. 275–310, doi:10.1023/A:1011265604021.

[7] José C. Campos & Michael D. Harrison (2009): *Interaction Engineering Using the IVY Tool*. In: Proceedings of the 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS '09, ACM, pp. 35–44, doi:10.1145/1570433.1570442.

[8] Camille Fayollas, Célia Martinie, Philippe Palanque, Eric Barboni, Racim Fahssi & Arnaud Hamon (In Press, 2016): *Exploiting Action Theory as a Framework for Analysis and Design of Formal Methods Approaches: Application to the CIRCUS Integrated Development Environment*. In: Formal Methods in Human Computer Interaction, Springer.

[9] Camille Fayollas, Célia Martinie, Philippe Palanque, Yannick Deleris, Jean-Charles Fabre & David Navarre (2014): *An Approach for Assessing the Impact of Dependability on Usability: Application to Interactive Cockpits*. In: Proceedings of the 2014 Tenth European Dependable Computing Conference, EDCC '14, IEEE Computer Society, pp. 198–209, doi:10.1109/EDCC.2014.17.

[10] Peter Forbrig, Célia Martinie, Philippe Palanque, Marco Winckler & Racim Fahssi (2014): *Rapid Task-Models Development Using Sub-models, Sub-routines and Generic Components*. In: Human-Centered Software Engineering: 5th IFIP WG 13.2 International Conference, HCSE 2014, Paderborn, Germany, September 16-18, 2014. Proceedings, Springer Berlin Heidelberg, pp. 144–163, doi:10.1007/978-3-662-44811-3_9.

[11] Michael D. Harrison, José C. Campos & Paolo Masci (2016): *Patterns and templates for automated verification of user interface software design in PVS*. Technical Report, Newcastle University. Available at `http://www.ncl.ac.uk/computing/research/publication/225438`.

[12] Michael D. Harrison, Paolo Masci, José C. Campos & Paul Curzon (In Press, 2016): *The specification and analysis of use properties of a nuclear control system*. In: Formal Methods in Human Computer Interaction, Springer.

[13] Constance Heitmeyer, James Kirby, Bruce Labaw & Ramesh Bharadwaj (1998): *SCR: A toolset for specifying and analyzing software requirements*. In Alan J. Hu & Moshe Y. Vardi, editors: Computer Aided Verification: 10th International Conference, CAV'98, 1427, Springer Berlin Heidelberg, pp. 526–531, doi:10.1007/BFb0028775.

[14] Ian Horrocks (1999): *Constructing the User Interface with Statecharts*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[15] Glenn E. Krasner & Stephen T. Pope (1988): *A Cookbook for Using the Model-view Controller User Interface Paradigm in Smalltalk-80*. J. Object Oriented Program. 1(3), pp. 26–49. Available at `http://dl.acm.org/citation.cfm?id=50757.50759`.

[16] Célia Martinie, Philippe Palanque, Eric Barboni & Martina Ragosta (2011): *Task-model based assessment of automation levels: application to space ground segments*. In: Systems, Man, and Cybernetics (SMC), 2011 IEEE International Conference on, IEEE, pp. 3267–3273, doi:10.1109/ICSMC.2011.6084173.

[17] Célia Martinie, Philippe Palanque & Marco Winckler (2011): *Structuring and Composition Mechanisms to Address Scalability Issues in Task Models*. In: Human-Computer Interaction – INTERACT 2011: 13th IFIP TC 13 International Conference, 2011, Proceedings, Part III, Springer Berlin Heidelberg, pp. 589–609, doi:10.1007/978-3-642-23765-2_40.

[18] Paolo Masci, Peter G. Larsen & Paul Curzon (2015): *Integrating the PVSio-web modelling and prototyping environment with Overture*. In: 13th Overture Workshop, satellite event of FM2015, Grace Technical Reports, Grace-TR 2015-06, pp. 33–47. Available at `http://grace-center.jp/wp-content/uploads/2012/05/13thOverture-Proceedings.pdf`.

[19] Paolo Masci, Patrick Oladimeji, Yi Zhang, Paul Jones, Paul Curzon & Harold Thimbleby (2015): *PVSio-web 2.0: Joining PVS to HCI*. In Daniel Kroening & S. Corina Păsăreanu, editors: Computer Aided Verification: 27th International Conference, CAV 2015, Proceedings, Part I, Springer International Publishing, pp. 470–478, doi:10.1007/978-3-319-21690-4_30. Tool available at `http://www.pvsioweb.org`.

[20] MathWorks: *Mathworks Simulink*. `http://www.mathworks.com/products/simulink`.

[21] Gioacchino Mauro, Harold Thimbleby, Andrea Domenici & Cinzia Bernardeschi (2016): *Extending a user interface prototyping tool with automatic MISRA C code generation*. In: *3rd Workshop on Formal Integrated Development Environment (F-IDE), satellite workshop of Formal Methods 2016*, Electronic Proceedings in Theoretical Computer Science (EPTCS).

[22] César A Muñoz & Ricky Butler (2003): *Rapid prototyping in PVS*. Available at `http://ntrs.nasa.gov/search.jsp?R=20040046914`. NASA/CR-2003-212418, NIA Report No.2003-03.

[23] David Navarre, Philippe Palanque, Jean-Francois Ladry & Eric Barboni (2009): *ICOs: A Model-based User Interface Description Technique Dedicated to Interactive Systems Addressing Usability, Reliability and Scalability*. *ACM Transactions on Computer-Human Interaction (TOCHI)* 16(4), pp. 18:1–18:56, doi:10.1145/1614390.1614393.

[24] Sam Owre, John M. Rushby & Natarajan Shankar (1992): *PVS: A Prototype Verification System*. In: *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*, CADE-11, Springer Berlin Heidelberg, pp. 748–752, doi:10.1007/3-540-55602-8_217.

[25] James Lyle Peterson (1981): *Petri Net Theory and the Modeling of Systems*. Prentice Hall.

[26] José-Luis Silva, Camille Fayollas, Arnaud Hamon, Célia Martinie, Eric Barboni et al. (2014): *Analysis of WIMP and Post WIMP Interactive Systems based on Formal Specification*. *Electronic Communications of the EASST* 69, doi:10.14279/tuj.eceasst.69.967.

Appendix

	Formal IDE	CIRCUS	PVSio-web
1.	Scope/purpose	Interactive system prototyping, development, and analysis.	User interface software prototyping and analysis.
2.	Tool features	User task and goals description, interaction logic (dialog) and interaction techniques modelling, interactive system prototyping, support for verification of properties, assessment of compatibility between user tasks and interactive system prototype.	Interaction logic modelling, rapid prototyping of user interface software, verification of safety requirements and usability properties, code generation and documentation.
3.	Tool extensibility	Each tool within CIRCUS offers an API supporting connection to other computing systems. For instance, connecting PetShop execution engine to cockpit software simulators or connecting Petri net analysis tools to PetShop analysis module.	PVSio-web has a plug-in based architecture that enables the rapid introduction of new modelling, prototyping, and analysis tools; support for new widgets types and widgets libraries can be introduced in Prototype Builder; Emucharts Editor can be extended with new model generators and code generators.
4.	Background knowledge	Object-Oriented Petri Nets (for Petshop), Java programming, distributed systems, hierarchical task modelling.	State machines, PVS higher order logic and PVS theorem proving (only required for full formal verification).
5.	IDE principles	Netbeans Visual API	Web
6.	IDE availability	Available upon request for collaborations only.	Open source, downloadable at `http://www.pvsioweb.org`
7.	Notation names	ICO, HAMSTERS.	Emucharts, PVS.
8.	Notation instance	Petri Net, task models.	Statecharts, higher-order logic.
9.	Notation paradigm	Event-based, state-based, procedural.	Event-based, state-based, functional.
10.	Structuring models	Object-oriented, component-based.	Module-based.
11.	Model editing features	Graphical editing of task models, ICO models and their correspondences, auto-completion features of models, visual representation of properties on models, simulation of models at editing time.	Graphical and textual editing of models, automatic generation of PVS models.
12.	Suggestions for model improvement	Suggestions for model correction by real time analysis of models and continuous visualization of analysis results.	Strengthening of pre- and post- conditions of transition functions (based on proof obligations generated by PVS).

continues on next page...

	Formal IDE	CIRCUS	PVSio-web
13.	Prototype building	Use of graphical user interface editor of NetBeans for standard interactions (e.g. WIMP), possible to create interactive components and assemble them for non standard intereactions (e.g. multitouch).	Visual editing, based on a picture of the real system.
14.	Prototype execution	Java Virtual Machine.	Javascript execution environment, Lisp.
15.	User interface testing	Automatic execution of test sequences based on a task model	Automated execution of input test sequences recorded during interactions with the prototype.
16.	Human-machine interaction techniques	Pre-WIMP, WIMP, post-WIMP, multimodal, multi-touch. Run-time re-configuration of interaction techniques.	Pre-WIMP, WIMP, post-WIMP, multimodal.
17.	Code generation	Run-time execution of ICO models (to support prototyping and co-execution of task and system models).	Run-time execution of PVS executable models through the PVS ground evaluator (to support rapid prototyping), and automatic generation of production code compliant to MISRA-C (only for formal models developed using Emucharts diagrams).
18.	Verification types	Well-formedness of the model: absence of deadlocks, token conservation. Functional analysis: reinitiability; availability of widgets; compliance with task models. Quantitative analysis: statistics about the total number of user tasks; estimation of the cognitive workload of the user based on the types of human-machine interactions necessary to operate the system. Simulation-based analysis through model animation.	Functional analysis, including: coverage of conditions, disjointness of conditions, correct use of data types, compliance with design requirements. Simulation-based analysis through model animation.
19.	Technology	Static analysis of Petri Nets; interactive simulation of task and system models. Proofs and properties verification left to the analyst.	Theorem proving; interactive simulations.
20.	Scalability	Applied to very large scale (industrial) applications (more than 200 models).	User interface prototype of stand-alone devices.
21.	Analysis of the wider socio-technical system	Modelling of integrated views of the three elements of socio-technical systems (organization, human and interactive systems); however, FRAM-based description of organization and variability of performance has only be addressed at model level and not a tool level.	Modelling patterns based on distributed cognition theory have been explored in PVS but are not currently integrated in the IDE.
22.	Related development process	User centered design (task-based design), iterative development, model-based engineering.	User centered design, agile development, model-based engineering.

The Tinker GUI for graphical proof strategies (tool demo)*

Gudmund Grov
Heriot-Watt University, Edinburgh, UK
G.Grov@hw.ac.uk

Yuhui Lin
Heriot-Watt University, Edinburgh, UK
Y.Lin@hw.ac.uk

Pierre Le Bras
Heriot-Watt University, Edinburgh, UK
PL196@hw.ac.uk

1 Background

Most interactive theorem provers provide users with a tactic language in which they can encode common proof strategies in order to reduce user interaction. To encode proof strategies, these languages typically provide: a set of functions, called *tactics*, which reduces sub-goals into smaller and simpler sub-goals; and a set of combinators, called *tacticals*, which combines tactics in different ways.

Composition in most tacticals either relies on the number and the order of sub-goals, or it will try all tactics on all sub-goals. The former is brittle as the number and the order could be changed if any of the sub-tactics changes; and the latter is hard to debug and maintain, as if a proof fails the actual position is hard to find. It is also difficult for others to see the intuition behind tactic design.

2 PSGraph

To overcome these issues we developed *PSGraph*, a graphical proof strategy language [2], where complex tactics are represented as directed hierarchical graphs. Here, the nodes contain tactics or nested graphs, and are composed by labelled wires. The labels are called *goal types*: predicates describing expected properties of sub-goals. Each sub-goal becomes a special *goal node* on the graph, which "lives" on a wire. Evaluation is handled by applying a tactic to a goal node that is on one of its input wires. The resulting sub-goals are sent to the out wires of the tactic node. To add a goal node to a wire, the goal type must be satisfied. This mechanism is used to control that goals are sent to the right place, independent of number and order of sub-goals. For more details see [2].

A PSGraph can have the following types of nodes:

Atomic tactics An atomic tactic wraps a tactic of the underlying theorem prover. The default behaviour is to threat the label of the node as a tactic name and apply it as a function (tactic). For example, if a node is labelled by $strip\_\wedge$ then this name will be parsed and executed as a tactic in the underlying theorem prover, and fail if no such tactic exists.

*This work has been supported by EPSRC grants EP/J001058 and EP/K503915. This extended abstract is an adaptation of the features presented in [7].

Hierarchical nodes Modularity is achieved by hierarchies. This can also help to reduce the complexity and size of a PSGraph by hiding parts of it. We will illustrate the new hierarchy features below.

Identity nodes Identity nodes are used to fanout and join wires. As the name suggests, they do not change the sub-goals.

Breakpoints A novel feature of Tinker is the introduction of breakpoint nodes, which can be added/removed from wires by a simple mouse click.

Goal nodes A goal node wraps a sub-goal of a proof, and this can not be modified by the user, i.e. these nodes can only be changed through tactic applications, and introduced by the CORE when a new proof is started.

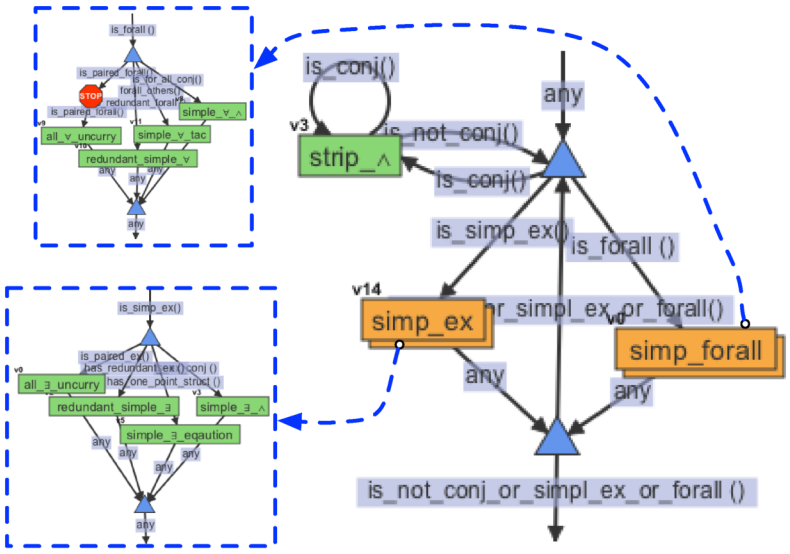


Figure 1: A PSGraph example [7].

To illustrate, consider the PSGraph in Figure 1. This example is taken from [7]. It is an encoding of a simple tactic from ProofPower to eliminate quantifiers. The overall strategy (i.e. the top-level graph on the right) contains an atomic tactic that repeatedly strip conjunctions (*strip_∧*), and two hierarchical nodes: one deals with existentially quantifier variables (*simp_ex*); the other deals with universal quantifiers (*simp_forall*). The goal types are used to send the goals to the correct node, and to decide when to terminate the loop.

3 The Tinker Tool

In [3], we introduced the Tinker tool, which implements PSGraph with support for *Isabelle/HOL* [2, 3], *ProofPower* [3, 7, 6]and *Rodin* [4]. Within the ProofPower tool, we are also investigating industrial usage of PSGraph and Tinker [6].

A PSGraph in Tinker can be applied as a normal tactic/method within the (Isabelle, ProofPower or Rodin) proof IDE. This is the normal execution. However, if it fails, it can instead be run in an ‘interactive

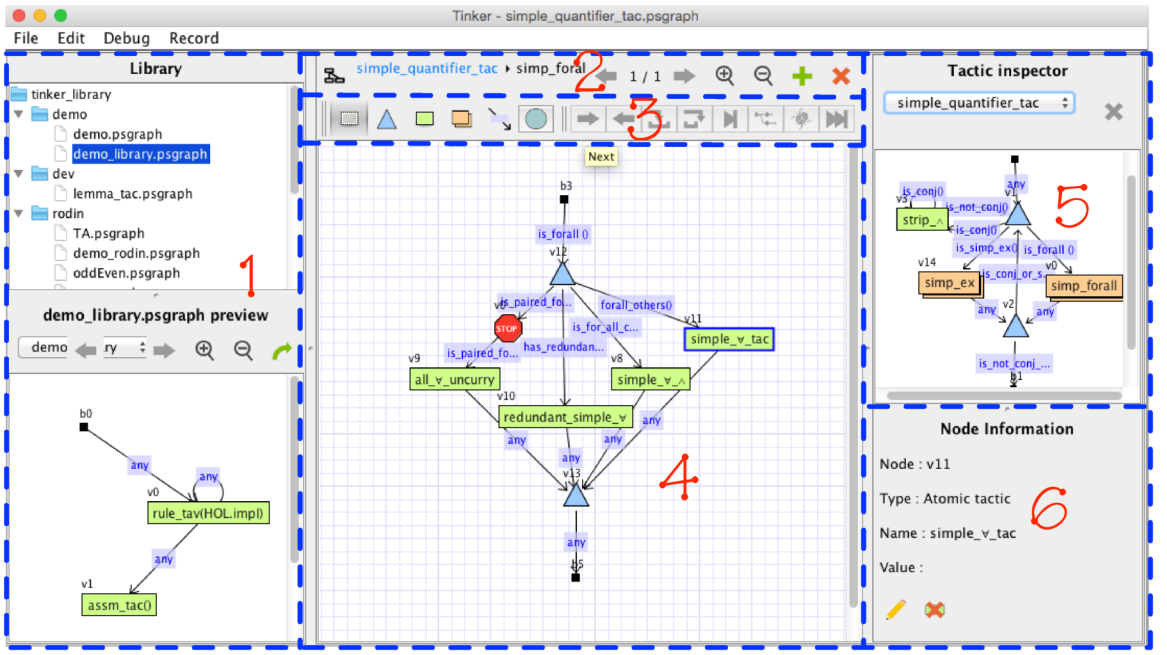


Figure 2: The Tinker GUI and its layout. [7]

mode' where the Tinker GUI is used to visualise and guide how the proof proceeds and identify where it failed. Figure 2 shows this GUI.

A user can draw a PSGraph from the *Graph panel* by selecting the type of node from the *Drawing and evaluation controls* panel. Nodes are connected by dragging a line between them. When selecting an entity, the details are displayed in the *Information panel*, and they can be edited by double clicking. The *Drawing and evaluations controls* panel gives users a lot of flexibility when developing and applying proof strategies, including:

1. select which goal to apply;
2. choose between stepping into and stepping over the evaluation of hierarchical nodes;
3. apply and complete the current hierarchical tactic;
4. apply and finish the whole proof strategy;
5. insert a breakpoint and evaluate a graph automatically until the break point is reached by a goal.

Tinker also provides features to export PSGraphs and record proofs – see [7] for further details.

4 Summary

With the exception of simple proof visualisation (e.g. [5]), we are not familiar with any other graphical proof tools to support theorem provers. While there are tactic languages that support robust tactics (e.g. Ltac [1] for Coq or Eisbach [8] for Isabelle), we believe that the development and debugging

features of Tinker are novel. Several features of the tool have been motivated by working with D-RisQ (`www.drisq.com`) in encoding their highly complex Supertac proof strategy in ProofPower [6].

In the future, we would like to improve static checking of PSGraph; for example that all atomic tactics used in the graph exist. We are also interested in investigating (semi-)automatic translations from traditional tactic language into PSGraph. This will also include more modern tactic languages, such as Ltac and Eisbach. We also plan to improve the layout algorithm, and develop and implement a better framework for combining evaluation and user edits of PSGraphs.

References

[1] D. Delahaye (2002): *A Proof Dedicated Meta-Language*. *Electronic Notes in Theoretical Computer Science* 70(2), pp. 96–109.

[2] G. Grov, A. Kissinger & Y. Lin (2013): *A Graphical Language for Proof Strategies*. In: *LPAR*, Springer, pp. 324–339.

[3] G. Grov, A. Kissinger & Y. Lin (2014): *Tinker, tailor, solver, proof*. In: *UITP 2014*, *ENTCS* 167, Open Publishing Association, pp. 23–34.

[4] Yibo Liang, Yuhui Lin & Gudmund Grov (2016): *'The Tinker' for Rodin*. In: *ABZ 2016*, Springer, pp. 262–268, doi:10.1007/978-3-319-33600-8_19.

[5] T Libal, M Riener & M Rukhaia (2014): *Advanced Proof Viewing in ProofTool*. In: *UITP 2014*, *EPTCS* 167, Open Publishing Association, pp. 35–47.

[6] Y. Lin, O'Halloran C. Grov, G. and & P. G (2016): *A Super Industrial Application of PSGraph*. In: *ABZ 2016*, Springer, pp. 319–325.

[7] Yuhui Lin, Pierre Le Bras & Gudmund Grov (2016): *Developing and Debugging Proof Strategies by Tinkering*. In Marsha Chechik & Jean-François Raskin, editors: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, Springer, Berlin, Heidelberg, pp. 573–579, doi:10.1007/978-3-662-49674-9_-37. Available at `http://dx.doi.org/10.1007/978-3-662-49674-9_37`.

[8] Daniel Matichuk, Makarius Wenzel & Toby Murray (2014): *An Isabelle Proof Method Language*, pp. 390–405. Springer International Publishing, Cham, doi:10.1007/978-3-319-08970-6_25. Available at `http://dx.doi.org/10.1007/978-3-319-08970-6_25`.

Extending the Dafny IDE with tactics and dead annotation analysis (tool demo)*

Gudmund Grov
Heriot-Watt University, Edinburgh, UK
G.Grov@hw.ac.uk

Yuhui Lin
Heriot-Watt University, Edinburgh, UK
Y.Lin@hw.ac.uk

Léon McGregor
Heriot-Watt University, Edinburgh, UK
lm356@hw.ac.uk

Vytautas Tumas
Heriot-Watt University, Edinburgh, UK
vt50@hw.ac.uk

Duncan Cameron
Heriot-Watt University, Edinburgh, UK
dac31@hw.ac.uk

1 Introduction

Dafny [4] is a verification-aware programming language where the specification of desired properties is intertwined with their implementation in the program text. It uses an automated theorem prover to *prove* that the specification is satisfied by the program. A specification serves two purposes: (1) it *specifies* the properties to be proven and acts as a *documentation* of the program, which is desirable to include in the program text; (2) it is used to *guide the prover* if a property cannot be verified without help. This is a necessary evil, which is not desirable and may obfuscate the readability of the program text. We will call these specfication element that are only there to guide the prover for *proofs*.

Dafny has a Visual Studio IDE plug-in [6], which seamlessly applies the verification in the background. In this demo we will show two extensions to this IDE:

- In [3] we developed an extension to Dafny that enables users to encode verification patterns of proofs as tactics, while [2] extends the language and illustrates several patterns as tactics. We will illustrate ongoing work on integrating tactics into the IDE – this is described further in §2.
- We are working on developing a feature to automatically remove unnecessary proof elements and integrating this into the IDE – this is described in §3.

2 Working with Dafny tactics in the IDE

Tacny is a conservative extension of Dafny with features to implement verification patterns as *tactics* [3, 2]. This tactic language is a *meta-language* for Dafny, where evaluation of a tactic works at the Dafny level: it takes a Dafny program with tactics and tactic applications, evaluates the applications and produces a new valid Dafny program, where tactic calls are replaced by Dafny constructs that tactics have generated.

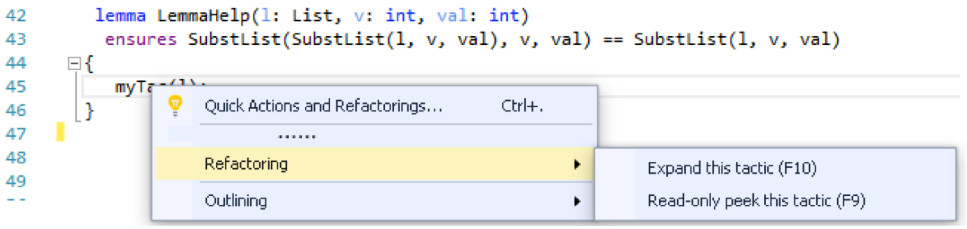
*This work has been supported by EPSRC grants EP/M018407/1 and EP/N014758/1. Special thanks to Rustan Leino and his colleagues at MSR. The work is an adaptation of features presented in [3, 2, 1] or submitted in parallel to a different conference.

A *tactic* is a special Dafny ghost method, recognisable by the **tactic** keyword. It contains many features to talk *about* a program, and features to *generate* proofs in terms of Dafny by *transforming* the program. A crucial property is that neither the program, nor the actual (non-proof) specification, can be changed – which we call *contract-preserving transformations* [3]. To illustrate, consider the following tactic:

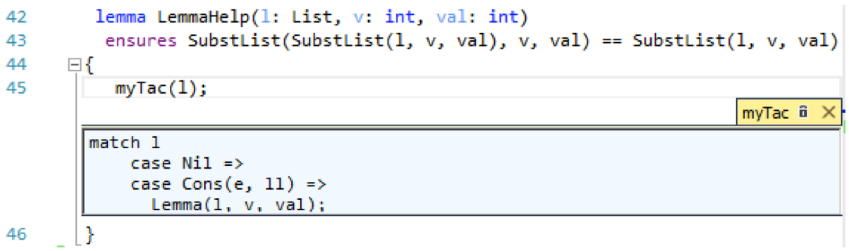
```
tactic myTac(v:Element, t:Tactic)
{
    tactic match v { t(); }
}
```

The myTac tactic generalises the notion of pattern matches (A Dafny **match** statement). As argument it takes a variable v which is assumed to be of an inductively defined datatype. The **tactic match** statement will then generate a **match** statement and a case for each constructor of the datatype, and apply tactic t() to each case.

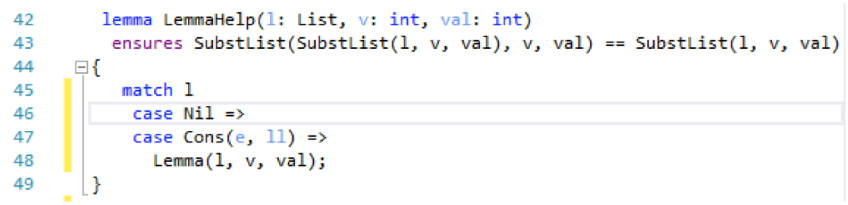
A tactic is evaluated by the Dafny verifier and the code generated is hidden from the user. Within a verified method that contains a tactic application, the code generated by the tactic can be made available in two different ways. This is illustrated by the 'refactoring' menu available by right-clicking the tactic application:



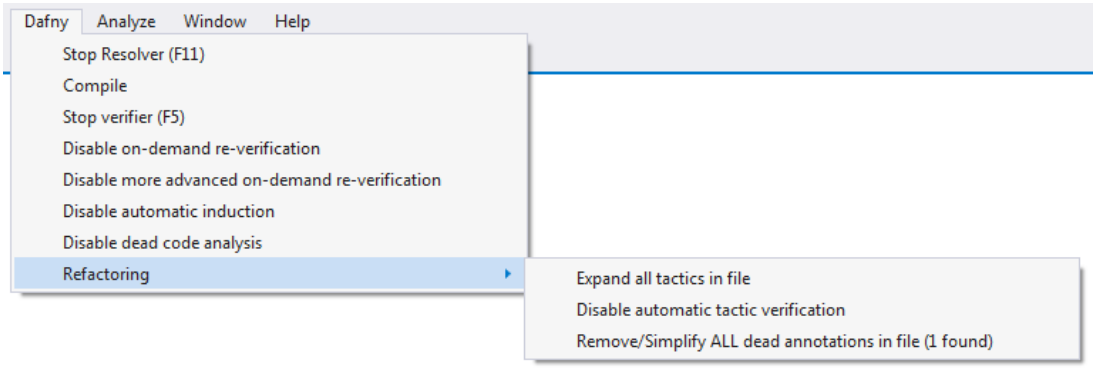
If the user selects 'Read-only peek this tactic (F9)' then a read-only window will appear in the editor, where the code is shown. This will not make any changes to source code:



A second alternative is to 'Expand this tactic (F10)'. Here, the code that the tactic application generated will replace the tactic call. This is illustrated by the following screenshot:



Finally, in the top-level menu, a user can either disable tactics, meaning all tactic applications will be ignored, or expanding all tactic applications in the program:



The last option is to remove all "dead annotaions", which we discuss next.

3 Dead annotation removal[1]

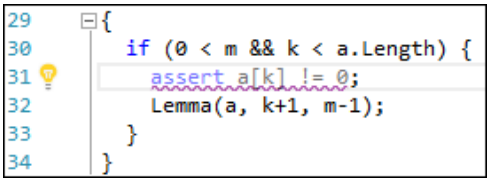
A fully verified Dafny program may contain unnecessary proof annotations, which may obfuscate the program text. There are at least two reasons for why such "dead annotations" may be present:

1. Proof constructs are normally added incrementally to the program text until a proof is found. Previous increments may not have helped to progress the proof, but are left by the user.
2. The underlying verifier is improved such that guidance that used to be required is no longer needed.

Inspired by the *dead code optimisation* found in most compilers, we have developed a tool called DARe (Dead Annotation Removal) that works by traversing the Dafny abstract syntax tree and remove as many annotations as possible. Each time an annotation is removed, Dafny is applied to check if the program still verifies, and the constructs will only be removed if Dafny does not complain. Here an annotation can be:

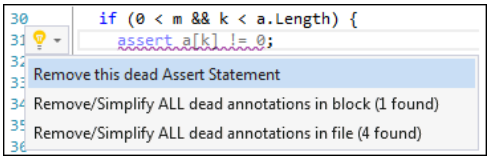
- An assertion.
- A call to lemma.
- A loop invariant.
- A loop variant (decreases clause).
- A step in a calculation[2].

We are integrating the DARe tool into the Dafny Visual Studio IDE. It will initially apply DARe to all methods (that does not have any verification errors) after the IDE has remained idle (i.e. no user-interaction) for at least 10 seconds. As DARe will be relatively slow to run, it will terminate (with failure) once a user starts interacting with the system. When it successfully terminates, all unnecessary (or dead) annotations will be highlighted (underlined and greyed out). This is illustrated on line 31 of the following screen-shot:



[1]This section is an adaptation [1].
[2]Dafny supports Dijkstra style calculational proofs [5].

The user can then press the light-bulb to get the options of removing the given dead annotation or all dead annotations of the method or file, as illustrated below:



The different options will behave as follows:

- The first option will only remove the selected dead annoation.
- The second option will remove all dead annotations in the current block (e.g. the method).
- The final option will remove all dead annotations in the entire file.

The tool will keep track of any methods that are changed since last time DARe was applied and will reapply DARe when a method have changed. Again, this will only happen if the method does not have any verification errors and the system remains idle for 10 seconds.

4 Summary

This tool demo shows two extensions to the Dafny Visual Studio IDE. Both of the extensions have the ability to improve the program text and to support users when developing Dafny programs: re-usable tactics can replace proofs by high-level proof patterns, while proof elements that are not required can be removed in a semi-automatic manner.

References

[1] Duncan Cameron, Gudmund Grov, and Léon McGregor. What is your actual annotation overhead? In informal proceedings of the 28th Nordic Workshop on Programming Theory (NWPT 2016). To appear.

[2] Gudmund Grov, Yuhui Lin, and Vytautas Tumas. Mechanised Verification Patterns for Dafny. In *21st International Conference on Formal Methods*. Springer, 2016. to appear.

[3] Gudmund Grov and Vytautas Tumas. Tactics for the Dafny Program Verifier. In Marsha Chechik and Jean-François Raskin, editors, *22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 36–53. Springer, 2016.

[4] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, volume 6355 of *LNCS*, pages 348–370. Springer-Verlag, 2010.

[5] K. R. M. Leino and N. Polikarpova. Verified Calculations. In *VSTTE*, 2013.

[6] K Rustan M Leino and Valentin Wüstholz. The Dafny integrated development environment. *arXiv preprint arXiv:1404.6602*, 2014.

Predicting SMT Solver Performance for Software Verification

Andrew Healy Rosemary Monahan James F. Power
Dept. of Computer Science, Maynooth University, Maynooth, Ireland
ahealy@cs.nuim.ie rosemary@cs.nuim.ie jpower@cs.nuim.ie

The Why3 IDE and verification system facilitates the use of a wide range of Satisfiability Modulo Theories (SMT) solvers through a driver-based architecture. We present Where4: a portfolio-based approach to discharge Why3 proof obligations. We use data analysis and machine learning techniques on static metrics derived from program source code. Our approach benefits software engineers by providing a single utility to delegate proof obligations to the solvers most likely to return a useful result. It does this in a time-efficient way using existing Why3 and solver installations — without requiring low-level knowledge about SMT solver operation from the user.

1 Introduction

The formal verification of software generally requires a software engineer to use a system of tightly integrated components. Such systems typically consist of an IDE that can accommodate both the implementation of a program and the specification of its formal properties. These two aspects of the program are then typically translated into the logical constructs of an intermediate language, forming a series of goals which must be proved in order for the program to be fully verified. These goals (or "proof obligations") must be formatted for the system's general-purpose back-end solver. Examples of systems which follow this model are Spec# [3] and Dafny [28] which use the Boogie [2] intermediate language and the Z3 [19] SMT solver.

Why3 [22] was developed as an attempt to make use of the wide spectrum of interactive and automated theorem proving tools and overcome the limitations of systems which rely on a single SMT solver. It provides a driver-based, extensible architecture to perform the necessary translations into the input formats of two dozen provers. With a wide choice of theorem-proving tools now available to the software engineer, the question of choosing the most appropriate tool for the task at hand becomes important. It is this question that Where4 answers.

As motivation for our approach, Table 1 presents the results from running the Why3 tool over the example programs included in the Why3 distribution (version 0.87.1), using eight SMT solvers at the back-end. Each Why3 file contains a number of theories requiring proof, and these in turn are broken down into a number of goals for the SMT solver; for the data in Table 1 we had 128 example programs, generating 289 theories, in turn generating 1048 goals. In Table 1 each row presents the data for a single SMT solver, and the three main data columns give data totalled on a per-file, per-theory and per-goal basis. Each of these three columns is further broken down to show the number of programs/theories/goals that were successfully solved, their percentage of the total, and the average time taken in seconds for each solver to return such a result. Program verification by modularisation construct is particularly relevant to the use of Why3 on the command line as opposed to through the IDE.

Table 1 also has a row for an imaginary "theoretical" solver, Choose Single, which corresponds to choosing the best (fastest) solver for each individual program, theory or goal. This solver performs significantly better than any individual solver, and gives an indication of the maximum improvement that could be achieved *if it was possible to predict in advance which solver was the best for a given program,*

EPTCS ??, 20??, pp. 1–18, doi:10.4204/EPTCS.??.??

Table 1: Results of running 8 solvers on the example Why3 programs with a timeout value of 10 seconds. In total our dataset contained 128 files, which generated 289 theories, which in turn generated 1048 goals. Also included is a theoretical solver Choose Single, which always returns the best answer in the fastest time.

	File			Theory			Goal		
	# proved	% proved	Avg time	# proved	% proved	Avg time	# proved	% proved	Avg time
Choose Single	**48**	**37.5%**	**1.90**	**190**	**63.8%**	**1.03**	**837**	**79.9%**	**0.42**
Alt-Ergo-0.95.2	25	19.5%	1.45	118	39.6%	0.77	568	54.2%	0.54
Alt-Ergo-1.01	34	26.6%	1.70	142	47.7%	0.79	632	60.3%	0.48
CVC3	19	14.8%	1.06	128	43.0%	0.65	597	57.0%	0.49
CVC4	19	14.8%	1.09	117	39.3%	0.51	612	58.4%	0.37
veriT	5	4.0%	0.12	79	26.5%	0.20	333	31.8%	0.26
Yices	14	10.9%	0.53	102	34.2%	0.22	368	35.1%	0.22
Z3-4.3.2	25	19.5%	0.56	128	43.0%	0.36	488	46.6%	0.38
Z3-4.4.1	26	20.3%	0.58	130	43.6%	0.40	581	55.4%	0.35

theory or goal. In general, the method of choosing from a range of solvers on an individual goal basis is called *portfolio-solving*. This technique has been successfully implemented in the SAT solver domain by SATzilla [38] and for model-checkers [20][36]. Why3 presents a unique opportunity to use a common input language to develop a portfolio SMT solver specifically designed for software verification.

The main contributions of this paper are:

1. The design and implementation of our portfolio solver, Where4, which uses supervised machine learning to predict the best solver to use based on metrics collected from goals.
2. The integration of Where4 into the user's existing Why3 work-flow by imitating the behaviour of an orthodox SMT solver.
3. A set of metrics to characterise Why3 goal formulae.
4. Statistics on the performance of eight SMT solvers using a dataset of 1048 Why3 goals.

Section 2 describes how the data was gathered and discusses issues around the accurate measurement of results and timings. A comparison of prediction models forms the basis of Section 3 where a number of evaluation metrics are introduced. The Where4 tool is compared to a range of SMT tools and strategies in Section 5. The remaining sections present a review of additional related work and a summary of our conclusions.

2 System Overview and Data Preparation

Due to the diverse range of input languages used by software verification systems, a standardised benchmark repository of verification programs does not yet exist [8]. For our study we chose the 128 example programs included in the Why3 distribution (version 0.87.1) as our corpus for training and testing purposes. The programs in this repository are written in WhyML, a dialect of ML with added specification syntax and verified libraries. Many of the programs are solutions to problems posed at software verification competitions such as VerifyThis [12], VSTTE [26] and COST [14]. Other programs are implementations of benchmarks proposed by the VACID-0 [29] initiative. It is our assumption that these programs

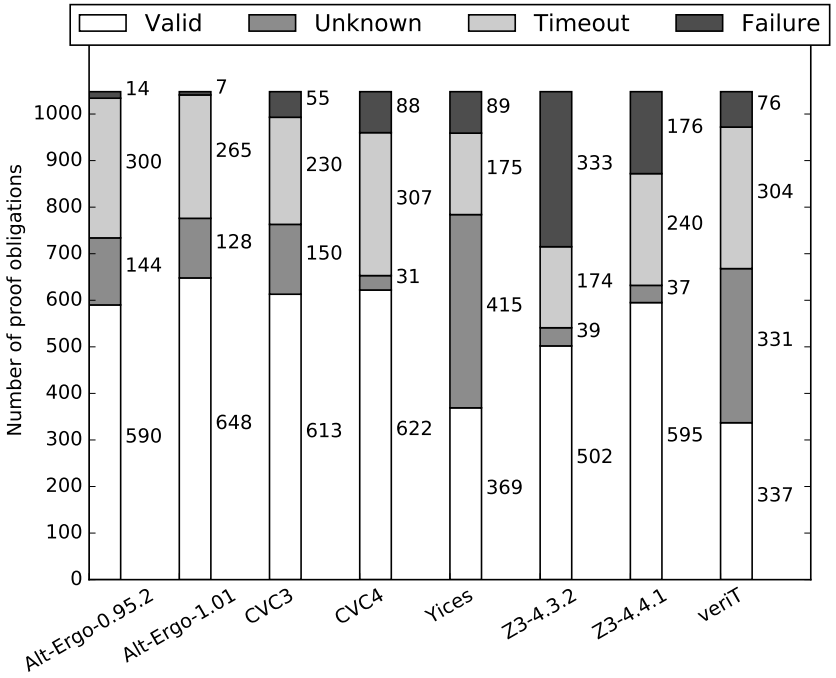


Figure 1: The relative amount of *Valid/Unknown/Timeout/Failure* answers from the eight SMT solvers (with a timeout of 60 seconds). Note that no tool returned an answer of *Invalid* for any of the 1048 proof obligations.

are a representative software verification workload. Alternatives to this dataset are discussed in Section 6.

We used six current, general-purpose SMT solvers supported by Why3: Alt-Ergo [18] versions 0.95.2 and 1.01, CVC3 [6] ver. 2.4.1, CVC4 [4] ver. 1.4, veriT [15], ver. 201506[1], Yices [21] ver. 1.0.38[2], and Z3 [19] ver. 4.3.2 and 4.4.1. We expanded the range of solvers to eight by recording the results for two of the most recent major versions of two popular solvers - Alt-Ergo and Z3.

When a solver is sent a goal by Why3 it returns one of the five possible answers *Valid*, *Invalid*, *Unknown*, *Timeout* or *Failure*. As can be seen from Table 1 and Fig. 1, not all goals can be proved Valid or Invalid. Such goals usually require the use of an interactive theorem prover to discharge goals that require reasoning by induction. Sometimes a splitting transformation needs to be applied to simplify the goals before they are sent to the solver. Our tool does not perform any transformations to goals other than those defined by the solver's Why3 driver file. In other cases, more time or memory resources need to be allocated in order to return a conclusive result. We address the issue of resource allocation in Section 2.1.1.

2.1 Problem Quantification: predictor and response variables

Two sets of data need to be gathered in supervised machine learning [31]: the independent/predictor variables which are used as input for both training and testing phases, and the dependent/response variables

[1]The most recent version - 201506 - is not officially supported by Why3 but is the only version available

[2]We did not use Yices2 as its lack of support for quantifiers makes it unsuitable for software verification

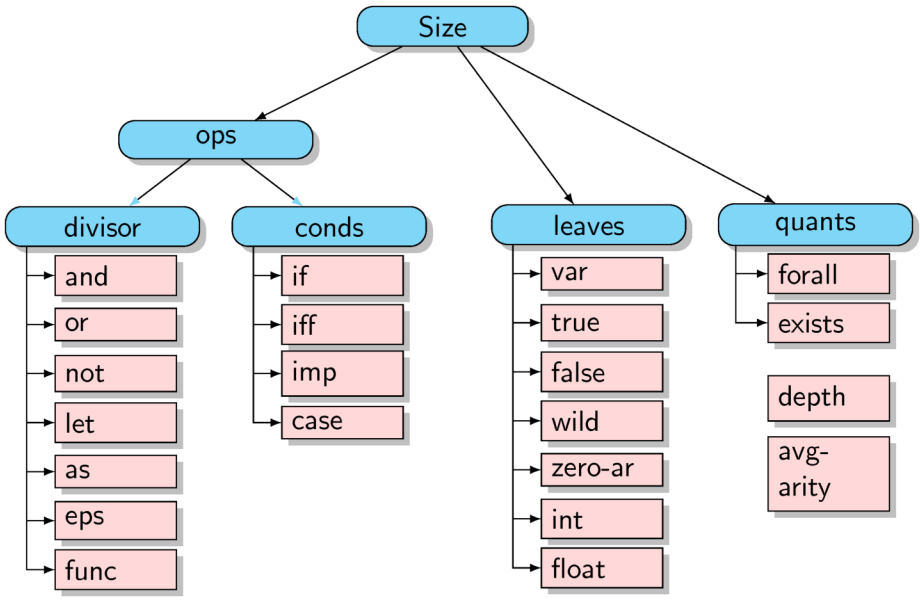


Figure 2: Tree illustrating the Why syntactic features counted individually (*pink nodes*) while traversing the AST. The rectangles represent individual measures, and the rounded blue nodes represent metrics that are the sum of their children in the tree.

which correspond to ground truths during training. Of the 128 programs in our dataset, 25% were held back for system evaluation (Section 5). The remaining 75% (corresponding to 96 WhyML programs, 768 goals) were used for training and 4-Fold cross-validation.

2.1.1 Independent/Predictor Variables

Fig. 2 lists the predictor variables that were used in our study. All of these are (integer-valued) metrics that can be calculated by analysing a Why3 proof obligation, and are similar to the *Syntax* metadata category for proof obligations written in the TPTP format [34]. To construct a feature vector from each task sent to the solvers, we traverse the abstract syntax tree (AST) for each goal and lemma, counting the number of each syntactic feature we find on the way. We focus on goals and lemmas as they produce proof obligations, with axioms and predicates providing a logical context.

Our feature extraction algorithm has similarities in this respect to the method used by Why3 for computing goal "shapes" [11]. These shape strings are used internally by Why3 as an identifying fingerprint. Across proof sessions, their use can limit the amount of goals in a file which need to be re-proved.

2.1.2 Dependent/Response Variables

Our evaluation of the performance of a solver depends on two factors: the time taken to calculate that result, and whether or not the solver had actually proven the goal.

In order to accurately measure the time each solver takes to return an answer, we used a measurement framework specifically designed for use in competitive environments. The BenchExec [9] framework was developed by the organisers of the SVCOMP [7] software verification competition to reliably measure CPU time, wall-clock time and memory usage of software verification tools. We recorded the time spent on CPU by each SMT solver for each proof obligation. To account for random errors in

measurement introduced at each execution, we used the methodology described by Lilja [30] to obtain an approximation of the true mean time. A 90% confidence interval was used with an allowed error of $\pm 3.5\%$.

By inspecting our data, we saw that most *Valid* and *Invalid* answers returned very quickly, with *Unknown* answers taking slightly longer, and *Failure/Timeout* responses taking longest. We took the relative utility of responses to be $\{Valid, Invalid\} > Unknown > \{Timeout, Failure\}$ which can be read as "it is better for a solver to return a *Valid* response than *Timeout*", etc. A simple function allocates a cost to each solver S's response to each goal G:

$$cost(S,G) = \begin{cases} time_{S,G}, \text{ if answer}_{S,G} \in \{Valid, Invalid\} \\ time_{S,G} + \text{timeout}, \text{ if answer}_{S,G} = Unknown \\ time_{S,G} + (\text{timeout} \times 2), \text{ if answer}_{S,G} \in \{Timeout, Failure\} \end{cases}$$

Thus, to penalise the solvers that return an *Unknown* result, the timeout limit is added to the time taken, while solvers returning *Timeout* or *Failure* are further penalised by adding double the timeout limit to the time taken. A response of *Failure* refers to an error with the backend solver and usually means a required logical theory is not supported. This function ensures the best-performing solvers always have the lowest costs. A ranking of solvers for each goal in order of decreasing relevance is obtained by sorting the solvers by ascending cost.

Since our cost model depends on the time limit value chosen, we need to choose a value that does not favour any one solver. To establish a realistic time limit value, we find each solver's "Peter Principle Point" [35]. In resource allocation for theorem proving terms, this point can be defined as the time limit at which more resources will not lead to a significant increase in the number of goals the solver can prove.

Fig. 3 shows the number of *Valid/Invalid/Unknown* results for each prover when given a time limit of 60 seconds. This value was chosen as an upper limit, since a time limit value of 60 seconds is not realistic for most software verification scenarios. Why3, for example, has a default time limit value of 5 seconds. From Fig. 3 we can see that the vast majority of useful responses are returned very quickly.

By satisfying ourselves with being able to record 99% of the useful responses which would be returned after 60 seconds, a more reasonable threshold is obtained for each solver. This threshold ranges from 7.35 secs (veriT) to 9.69 secs (Z3-4.3.2). Thus we chose a value of 10 seconds as a representative, realistic time limit that gives each solver a fair opportunity to return decent results.

3 Choosing a prediction model

Given a Why3 goal, a ranking of solvers can be obtained by sorting the cost for each solver. For unseen instances, two approaches to prediction can be used: (1) classification — predicting the final ranking directly — and (2) regression — predicting each solver's score individually and deriving a ranking from these predictions. With eight solvers, there are 8! possible rankings. Many of these rankings were observed very rarely or did not appear at all in the training data. Such an unbalanced dataset is not appropriate for accurate classification, leading us to pursue the regression approach.

Seven regression models were evaluated[3]: Linear Regression, Ridge Regression, K-Nearest Neighbours, Decision Trees, Random Forests (with and without discretisation) and the regression variant of Support Vector Machines. Table 2 shows the results for some of the best-performing models. Most

[3]We used the Python Sci-kit Learn [33] implementations of these models

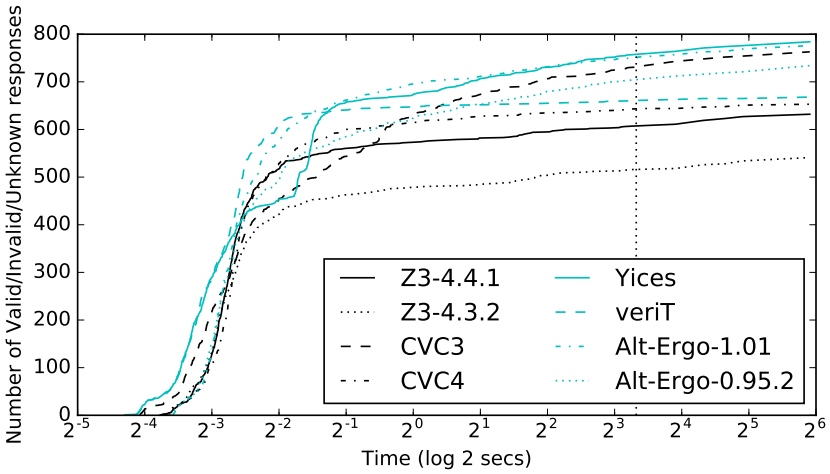


Figure 3: The cumulative number of *Valid/Invalid/Unknown* responses for each solver. The plot uses a logarithmic scale on the time axis for increased clarity at the low end of the scale. The chosen timeout limit of 10 secs (*dotted vertical line*) includes 99% of each solver's useful responses

models were evaluated with and without a weighting function applied to the training samples. Weighting is standard practice in supervised machine learning: each sample's weight was defined as the standard deviation of solver costs. This function was designed to give more importance to instances where there was a large difference in performance among the solvers.

Table 2 also shows three theoretical strategies in order to provide bounds for the prediction models. *Best* always chooses the best ranking of solvers and *Worst* always chooses the worst ranking (which is the reverse ordering to *Best*). *Random* is the average result of choosing every permutation of the eight solvers for each instance in the training set. We use this strategy to represent the user selecting SMT solvers at random without any consideration for goal characterisation or solver capabilities. A comparison to a *fixed* ordering of solvers for each goal is not made as any such ordering would be arbitrarily determined.

We note that the *Best* theoretical strategy of Table 2 is not directly comparable with the theoretical solver Choose Single from Table 1. The two tables' average time columns are measuring different results: in contrast to Choose Single, *Best* will call each solver in turn, as will all the other models in Table 2, until a *Valid/Invalid* result is recorded (which it may never be). Thus Table 2's *Time* column shows the average *cumulative* time of each such sequence of calls, rather than the average time taken by the single *best* solver called by Choose Single.

3.1 Evaluating the prediction models

Table 2's *Time* column provides an overall estimate of the effectiveness of each prediction model. We can see that the discretised Random Forest method provides the best overall results for the solvers, yielding an average time of 14.92 seconds.

The second numeric column of Table 2 shows the Normalised Discounted Cumulative Gain (***nDCG***), which is commonly used to evaluate the accuracy of rankings in the search engine and e-commerce recommender system domains [24]. Here, emphasis is placed on correctly predicting items higher in the

Table 2: Comparing the seven prediction models and three theoretical strategies

	Time (secs)	nDCG	R^2	MAE	Reg. error
Best	12.63	1.00	-	0.00	0.00
Random	19.06	0.36	-	2.63	50.77
Worst	30.26	0.00	-	4.00	94.65
Random Forest	15.02	0.48	**0.28**	2.08	**38.91**
Random Forest (discretised)	**14.92**	0.48	-0.18	2.13	39.19
Decision Tree	15.80	0.50	0.11	2.06	43.12
K-Nearest Neighbours	15.93	**0.53**	0.16	**2.00**	43.41
Support Vector Regressor	15.57	0.47	0.14	2.26	47.45
Linear Regression	15.17	0.42	-0.16	2.45	49.25
Ridge	15.11	0.42	-0.15	2.45	49.09

ranking. For a general ranking of length p, it is formulated as:

$$nDCG_p = \frac{DCG_p}{IDCG_p} \quad \text{where} \quad DCG_p = \sum_{i=1}^{p} \frac{2^{rel_i} - 1}{log_2(i+1)}$$

Here rel_i refers to the relevance of element i with regard to a ground truth ranking, and we take each solver's relevance to be inversely proportional to its rank index. In our case, $p = 8$ (the number of SMT solvers). The DCG_p is normalised by dividing it by the maximum (or *idealised*) value for ranks of length p, denoted $IDCG_p$. As our solver rankings are permutations of the ground truth (making *nDCG* values of 0 impossible), the values in Table 2 are further normalised to the range [0..1] using the lower *nDCG* bound for ranks of length 8 — found empirically to be 0.4394.

The third numeric column of Table 2 shows the R^2 score (or coefficient of determination), which is an established metric for evaluating how well regression models can predict the variance of dependent/response variables. The maximum R^2 score is 1 but the minimum can be negative. Note that the theoretical strategies return rankings rather than individual solver costs. For this reason, R^2 scores are not applicable. Table 2's fourth numeric column shows the *MAE* (Mean Average Error) — a ranking metric which can also be used to measure string similarity. It measures the average distance from each predicted rank position to the solver's index in the ground truth. Finally, the fifth numeric column of Table 2 shows the mean regression error (*Reg. error*) which measures the mean absolute difference in predicted solver costs to actual values.

3.2 Discussion: choosing a prediction model

An interesting feature of all the best-performing models in Table 2 is their ability to predict *multi-output* variables [13]. In contrast to the Support Vector model, for example, which must predict the cost for each solver individually, a multi-output model predicts each solver's cost simultaneously. Not only is this method more efficient (by reducing the number of estimators required), but it has the ability to account for the correlation of the response variables. This is a useful property in the software verification domain where certain goals are not provable and others are trivial for SMT solvers. Multiple versions of the same solver can also be expected to have highly correlated costs.

After inspecting the results for all learning algorithms (summarised in Table 2), we can see that random forests [16] perform well, relative to other methods. They score highest for three of the five

metrics (shown in bold) and have generally good scores in the others. Random forests are an ensemble extension of decision trees: random subsets of the training data are used to train each tree. For regression tasks, the set of predictions for each tree is averaged to obtain the forest's prediction. This method is designed to prevent over-fitting.

Based on the data in Table 2 we selected random forests as the choice of predictor to use in Where4.

4 Implementing Where4 in OCaml

Where4's interaction with Why3 is inspired by the use of machine learning in the Sledgehammer tool [10] which allows the use of SMT solvers in the interactive theorem prover Isabelle/HOL. We aspired to Sledgehammer's 'zero click, zero maintenance, zero overhead' philosophy in this regard: it should not interfere with a Why3 user's normal work-flow nor should it penalise those who do not use it.

We implement a "pre-solving" heuristic commonly used by portfolio solvers [1][38]: a single solver is called with a short time limit before feature extraction and solver rank prediction takes place. By using a good "pre-solver" at this initial stage, easily-proved instances are filtered with a minimum time overhead. We used a ranking of solvers based on the number of goals each could prove, using the data from Table 1. The highest-ranking solver installed locally is chosen as a pre-solver. For the purposes of this paper which assumes all 8 solvers are installed, the pre-solver corresponds to Alt-Ergo version 1.01. The effect pre-solving has on the method Where4 uses to return responses is illustrated in Alg. 1.

The random forest is fitted on the entire training set and encoded as a JSON file for legibility and modularity. This approach allows new trees and forests devised by the user (possibly using new SMT solvers or data) to replace our model. When the user installs Where4 locally, this JSON file is read and printed as an OCaml array. For efficiency, other important configuration information is compiled into OCaml data structures at this stage: e.g. the user's `why3.conf` file is read to determine the supported SMT solvers. All files are compiled and a native binary is produced. This only needs to be done once (unless the locally installed provers have changed).

The Where4 command-line tool has the following functionality:

1. Read in the WhyML/Why file and extract feature vectors from its goals.
2. Find the predicted costs for each of the 8 provers by traversing the random forest, using each goal's feature vector.
3. Sort the costs to produce a ranking of the SMT solvers.
4. Return a predicted ranking for each goal in the file, without calling any solver .
5. Alternatively, use the Why3 API to call each solver (if it is installed) in rank order until a *Valid/Invalid* answer is returned (using Alg. 1).

If the user has selected that Where4 be available for use through Why3, the file which lets Why3 know about supported provers installed locally is modified to contain a new entry for the Where4 binary. A simple driver file (which just tells Why3 to use the Why logical language for encoding) is added to the drivers' directory. At this point, Where4 can be detected by Why3, and then used at the command line, through the IDE or by the OCaml API just like any other supported solver.

5 Evaluating Where4's performance on test data

The evaluation of Where4 was carried out on a test set of 32 WhyML files, 77 theories, 263 goals (representing 25% of the entire dataset). This section is guided by the following three Evaluation Criteria:

Input: P, a Why3 program;
R, a static ranking of solvers for pre-proving;
ϕ, a timeout value
Output: $\langle A, T \rangle$ where
A = the best answer from the solvers;
T = the cumulative time taken to return A
begin
  /* Highest ranking solver installed locally */
  $S \leftarrow BestInstalled(R)$
  /* Call solver S on Why3 program P with a timeout of 1 second */
  $\langle A, T \rangle \leftarrow Call(P, S, 1)$
  if $A \in \{Valid, Invalid\}$ **then**
    return $\langle A, T \rangle$
  end
  /* extract feature vector F from program P */
  $F \leftarrow ExtractFeatures(P)$
  /* R is now based on program features */
  $R \leftarrow PredictRanking(F)$
  while $A \notin \{Valid, Invalid\} \wedge R \neq \emptyset$ **do**
    $S \leftarrow BestInstalled(R)$
    /* Call solver S on Why3 program P with a timeout of ϕ seconds */
    $\langle A_S, T_S \rangle \leftarrow Call(P, S, \phi)$
    /* add time T_S to the cumulative runtime */
    $T \leftarrow T + T_S$
    if $A_S > A$ **then**
      /* answer A_S is better than the current best answer */
      $A \leftarrow A_S$
    end
    /* remove S from the set of solvers R */
    $R \leftarrow R \setminus \{S\}$
  end
  return $\langle A, T \rangle$
end

Algorithm 1: Returning an answer and runtime from a Why3 input program

Table 3: Number of files, theories and goals proved by each strategy and individual solver. The percentage this represents of the total 32 files, 77 theories and 263 goals and the average time (in seconds) are also shown.

	File			Theory			Goal		
	# proved	% proved	Avg time	# proved	% proved	Avg time	# proved	% proved	Avg time
Where4	11	34.4%	1.39	44	57.1%	0.99	203	77.2%	1.98
Best			0.25			0.28			0.37
Random			4.19			4.02			5.70
Worst			14.71			13.58			18.35
Alt-Ergo-0.95.2	8	25.0%	0.78	37	48.1%	0.26	164	62.4%	0.34
Alt-Ergo-1.01	10	31.3%	1.07	39	50.6%	0.26	177	67.3%	0.33
CVC3	5	15.6%	0.39	36	46.8%	0.21	167	63.5%	0.38
CVC4	4	12.5%	0.56	32	41.6%	0.21	147	55.9%	0.35
veriT	2	6.3%	0.12	24	31.2%	0.12	100	38.0%	0.27
Yices	4	12.5%	0.32	32	41.6%	0.15	113	43.0%	0.18
Z3-4.3.2	6	18.8%	0.46	31	40.3%	0.20	145	55.1%	0.37
Z3-4.4.1	6	18.8%	0.56	31	40.3%	0.23	145	55.1%	0.38

5.1 EC1: How does Where4 perform in comparison to the 8 SMT solvers under consideration?

When each solver in Where4's ranking sequence is run on each goal, the maximum amount of files, theories and goals are provable. As Table 3 shows, the difference between Where4 and our set of reference theoretical strategies (*Best, Random*, and *Worst*) is the amount of time taken to return the *Valid/Invalid* result. Compared to the 8 SMT provers, the biggest increase is on individual goals: Where4 can prove 203 goals, which is 26 (9.9%) more goals than the next best single SMT solver, Alt-Ergo-1.01.

Unfortunately, the average time taken to solve each of these goals is high when compared to the 8 SMT provers. This tells us that Where4 can perform badly with goals which are not provable by many SMT solvers: expensive *Timeout* results are chosen before the *Valid* result is eventually returned. In the worst case, Where4 may try and time-out for all 8 solvers in sequence, whereas each individual solver does this just once. Thus, while having access to more solvers allows more goals to be proved, there is also a time penalty to portfolio-based solvers in these circumstances.

At the other extreme, we could limit the portfolio solver to just using the best predicted individual solver (after "pre-solving"), eliminating the multiple time-out overhead. Fig. 4 shows that the effect of this is to reduce the number of goals provable by Where4, though this is still more than the best-performing individual SMT solver, Alt-Ergo-1.01.

To calibrate this cost of Where4 against the individual SMT solvers, we introduce the notion of a *cost threshold*: using this strategy, after pre-solving, solvers with a predicted cost above this threshold are not called. If no solver's cost is predicted below the threshold, the pre-solver's result is returned.

Fig. 5 shows the effect of varying this threshold, expressed in terms of the average execution time (top graph) and the number of goals solved (bottom graph). As we can see from both graphs in Fig. 5, for the goals in the test set a threshold of 7 for the cost function allows Where4 to prove more goals than any single solver, in a time approximately equal to the four slower solvers (CVC4, veriT and both

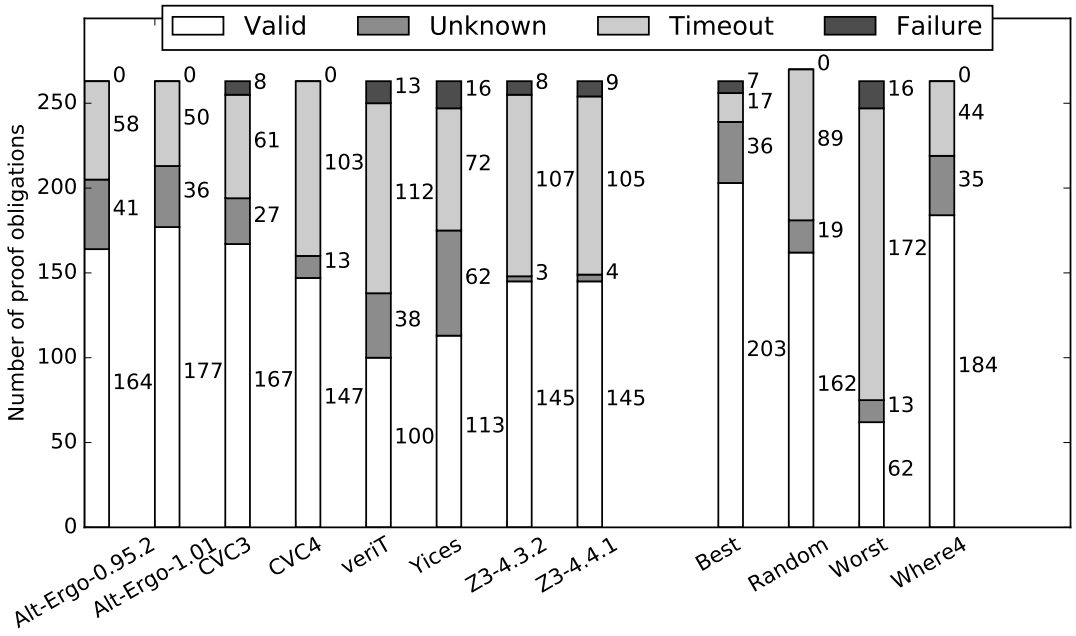


Figure 4: The relative amount of Valid/Unknown/Timeout/Failure answers from the eight SMT solvers. Shown on the right are results obtainable by using the top solver (only) with the 3 ranking strategies and the Where4 predicted ranking (with an Alt-Ergo-1.01 pre-solver).

versions of Z3).

5.2 EC2: How does Where4 perform in comparison to the 3 theoretical ranking strategies?

Fig. 6 compares the cumulative time taken for Where4 and the 3 ranking strategies to return the 203 valid answers in the test set. Although both Where4 and *Random* finish at approximately the same time, Where4 is significantly faster for returning *Valid/Invalid* answers. Where4's solid line is more closely correlated to *Best*'s rate of success than the erratic rate of the *Random* strategy. *Best*'s time result shows the capability of a perfect-scoring learning strategy. It is motivation to further improve Where4 in the future.

5.3 EC3: What is the time overhead of using Where4 to prove Why3 goals?

The timings for Where4 in all plots and tables are based solely on the performance of the constituent solvers (the measurement of which is discussed in Sec. 2.1.2). They do not measure the time it takes for the OCaml binary to extract the static metrics, traverse the decision trees and predict the ranking. We have found that this adds (on average) 0.46 seconds to the time Where4 takes to return a result for each file. On a per goal basis, this is equivalent to an increase in 0.056 seconds.

The imitation of an orthodox solver to interact with Why3 is more costly: this is due to Why3 printing each goal as a temporary file to be read in by the solver individually. Future work will look at bypassing this step for WhyML files while still allowing files to be proved on an individual theory and goal basis.

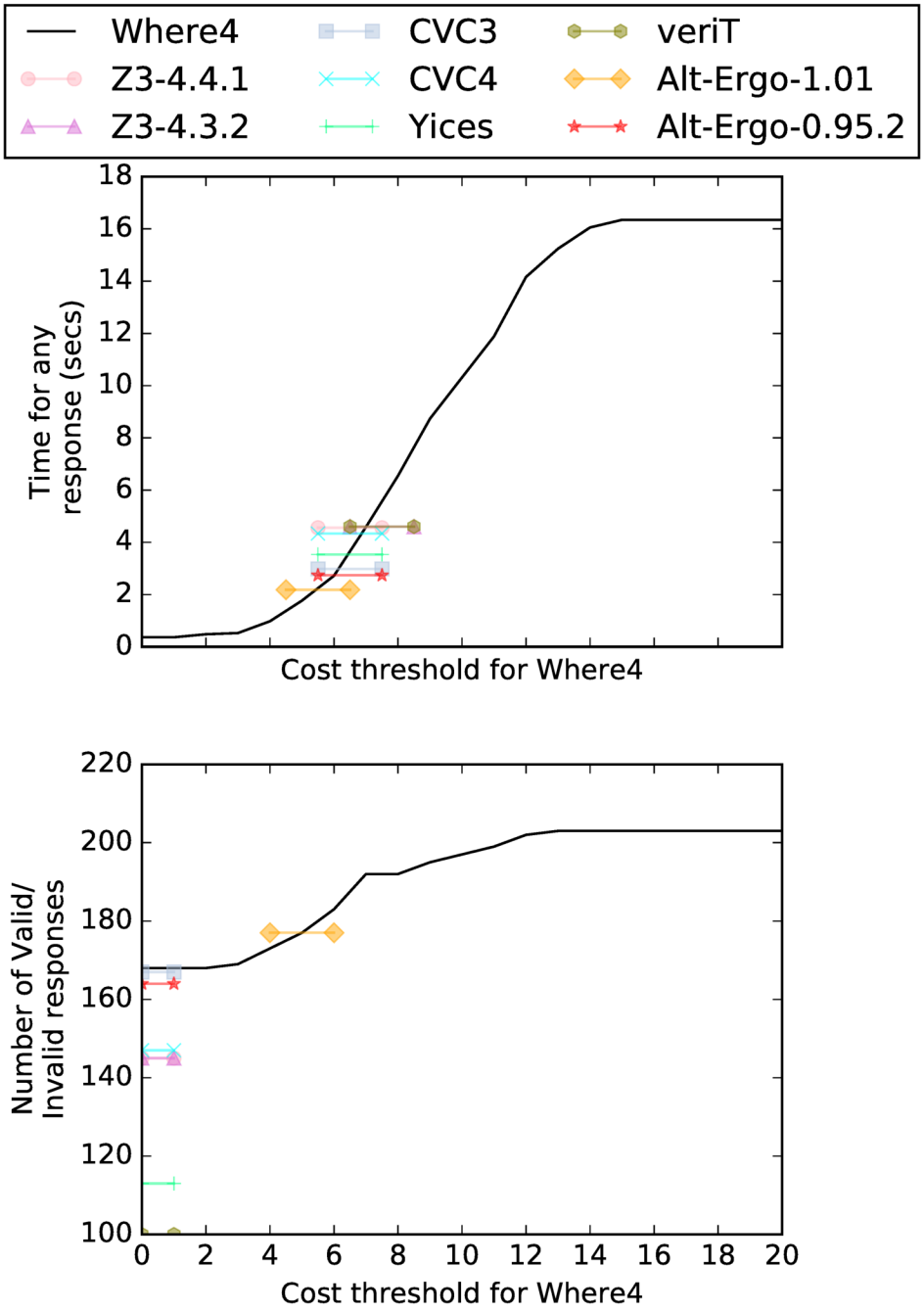


Figure 5: The effect of using a cost threshold. (*top*) The average time taken for Where4 to return an answer compared to 8 SMT solvers. (*bottom*) The number of Valid/Invalid answers returned by Where4 compared to 8 SMT solvers. For the 7 solvers other than Alt-Ergo-1.01, the number of provable goals is indicated by a mark on the y-axis rather than an intersection with Where4's results.

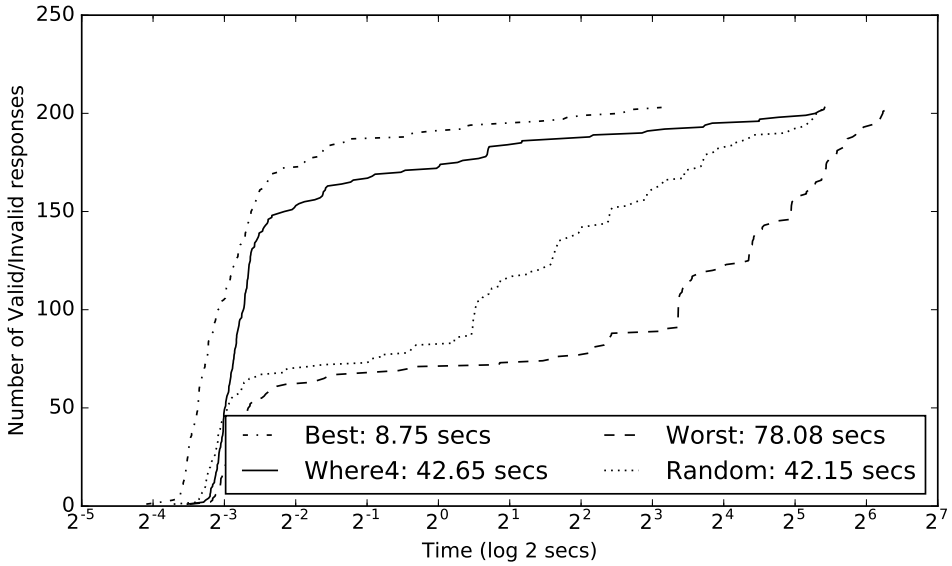


Figure 6: The cumulative time each theoretical strategy, and Where4 to return all *Valid/Invalid* answers in the test dataset of 263 goals

5.4 Threats to Validity

We categorise threats as either *internal* or *external*. Internal threats refer to influences that can affect the response variable without the researcher's knowledge and threaten the conclusions reached about the *cause* of the experimental results [37]. Threats to external validity are conditions that limit the generalisability and reproducibility of an experiment.

5.4.1 Internal

The main threat to our work's internal validity is selection bias. All of our training and test samples are taken from the same source. We took care to split the data for training and testing purposes on a *per file* basis. This ensured that Where4 was not trained on a goal belonging to the same theory or file as any goal used for testing. The results of running the solvers on our dataset are imbalanced. There were far more *Valid* responses than any other response. No goal in our dataset returned an answer of *Invalid* on any of the 8 solvers. This is a serious problem as Where4 would not be able to recognize such a goal in real-world use. In future work we hope to use the TPTP benchmark library to remedy these issues. The benchmarks in this library come from a diverse range of contributors working in numerous problem domains [35] and are not as specific to software verification as the Why3 suite of examples.

Use of an independent dataset is likely to influence the performance of the solvers. Alt-Ergo was designed for use with the Why3 platform — its input language is a previous version of the Why logic language. It is natural that the developers of the Why3 examples would write programs which Alt-Ergo in particular would be able to prove. Due to the syntactic similarities in input format and logical similarities such as support for type polymorphism, it is likely that Alt-Ergo would perform well with any Why3 dataset. We would hope, however, that the gulf between it and other solvers would narrow.

There may be confounding effects in a solver's results that are not related to the independent variables we used (Sec. 2.1.1). We were limited in the tools available to extract features from the domain-specific

Why logic language (in contrast to related work on model checkers which use the general-purpose C language [20][36]). We made the decision to keep the choice of independent variables simple in order to increase generalisability to other formalisms such as Microsoft's Boogie [2] intermediate language.

5.4.2 External

The generalisability of our results is limited by the fact that all dependent variables were measured on a single machine.[4] We believe that the number of each response for each solver would not vary dramatically on a different machine of similar specifications. By inspecting the results when each solver was given a timeout of 60 seconds (Fig. 3), the rate of increase for *Valid/Invalid* results was much lower than that of *Unknown/Failure* results. The former set of results are more important when computing the cost value for each solver-goal pair.

Timings of individual goals are likely to vary widely (even across independent executions on the same machine). It is our assumption that although the actual timed values would be quite different on any other machine, the *ranking* of their timings would stay relatively stable.

A "typical" software development scenario might involve a user verifying a single file with a small number of resultant goals: certainly much smaller than the size of our test set (263 goals). In such a setting, the productivity gains associated with using Where4 would be minor. Where4 is more suited therefore to large-scale software verification.

5.5 Discussion

By considering the answers to our three Evaluation Criteria, we can make assertions about the success of Where4. The answer to EC1, Where4's performance in comparison to individual SMT solvers, is positive. A small improvement in *Valid/Invalid* responses results from using only the top-ranked solver, while a much bigger increase can be seen by making the full ranking of solvers available for use. The time penalty associated with calling a number of solvers on an un-provable proof obligation is mitigated by the use of a *cost threshold*. Judicious use of this threshold value can balance the time-taken-versus-goals-proved trade-off: in our test set of 263 POs, using a threshold value of 7 results in 192 *Valid* responses – an increase of 15 over the single best solver – in a reasonable average time per PO (both *Valid* and otherwise) of 4.59 seconds.

There is also cause for optimism in Where4's performance as compared to the three theoretical ranking strategies — the subject of Evaluation Criterion 2. All but the most stubborn of *Valid* answers are returned in a time far better than *Random* theoretical strategy. We take this random strategy as representing the behaviour of the non-expert Why3 user who does not have a preference amongst the variety of supported SMT solvers. For this user, Where4 could be a valuable tool in the efficient initial verification of proof obligations through the Why3 system.

In terms of time overhead — the concern of EC3 — our results are less favourable, particularly when Where4 is used as an integrated part of the Why3 toolchain. The costly printing and parsing of goals slows Where4 beyond the time overhead associated with feature extraction and prediction. At present, due to the diversity of languages and input formats used by software verification tools, this is an unavoidable pre-processing step enforced by Why3 (and is indeed one of the Why3 system's major advantages).

[4] All data collection was conducted on a 64-bit machine running Ubuntu 14.04 with a dual-core Intel i5-4250U CPU and 16GB of RAM.

Overall, we believe that the results for two out of three Evaluation Criteria are encouraging and suggest a number of directions for future work to improve Where4.

6 Comparison with Related Work

Comparing verification systems: The need for a standard set of benchmarks for the diverse range of software systems is a recurring issue in the literature [8]. The benefits of such a benchmark suite are identified by the SMTLIB [5] project. The performance of SMT solvers has significantly improved in recent years due in part to the standardisation of an input language and the use of standard benchmark programs in competitions [17][7]. The TPTP (Thousands of Problems for Theorem Provers) project [34] has similar aims but a wider scope: targeting theorem provers which specialise in numerical problems as well as general-purpose SAT and SMT solvers. The TPTP library is specifically designed for the rigorous experimental comparison of solvers [35].

Portfolio solvers: Portfolio-solving approaches have been implemented successfully in the SAT domain by SATzilla [38] and the constraint satisfaction / optimisation community by tools such as CPHydra [32] and sunny-cp [1]. Numerous studies have used the SVCOMP [7] benchmark suite of C programs for model checkers to train portfolio solvers [36][20]. These particular studies have been predicated on the use of Support Vector Machines (SVM) with only a cursory use of linear regression [36]. In this respect, our project represents a more wide-ranging treatment of the various prediction models available for portfolio solving. The need for a strategy to delegate Why3 goals to appropriate SMT solvers is stated in recent work looking at verification systems on cloud infrastructures [23].

Machine Learning in Formal Methods: The FlySpec [25] corpus of proofs has been the basis for a growing number of tools integrating interactive theorem provers with machine-learning based fact-selection. The MaSh engine in Sledgehammer [10] is a related example. It uses a Naive Bayes algorithm and clustering to select facts based on syntactic similarity. Unlike Where4, MaSh uses a number of metrics to measure the *shape* of goal formulæas features. The weighting of features uses an inverse document frequency (IDF) algorithm. ML4PG (Machine Learning for Proof General) [27] also uses clustering techniques to guide the user for interactive theorem proving.

Our work adds to the literature by applying a portfolio-solving approach to SMT solvers. We conduct a wider comparison of learning algorithms than other studies which mostly use either SVMs or clustering. Unlike the interactive theorem proving tools mentioned above, Where4 is specifically suited to software verification through its integration with the Why3 system.

7 Conclusion and Future Work

We have presented a strategy to choose appropriate SMT solvers based on Why3 syntactic features. Users without any knowledge of SMT solvers can prove a greater number of goals in a shorter amount of time by delegating to Where4 than by choosing solvers at random. Although some of Where4's results are disappointing, we believe that the Why3 platform has great potential for machine-learning based portfolio-solving. We are encouraged by the performance of a theoretical *Best* strategy and the convenience that such a tool would give Why3 users.

The number of potential directions for this work is large: parallel solving, minimal datasets for practical local training, larger and more generic datasets for increased generalisability, etc. The TPTP repository represents a large source of proof obligations which can be translated into the Why logic language. The number of goals provable by Where4 could be increased by identifying which goals need

to be simplified in order to be tractable for an SMT solver. Splitting transforms would also increase the number of goals for training data: from 1048 to 7489 through the use of the `split_goal_wp` transform, for example. An interesting direction for this work could be the identification of the appropriate transformations. Also, we will continue to improve the efficiency of Where4 when used as a Why3 solver and investigate the use of a minimal benchmark suite which can be used to train the model using new SMT solvers and theorem provers installed locally.

Data related to this paper is hosted at `github.com/ahealy19/F-IDE-2016`. Where4 is hosted at `github.com/ahealy19/where4`.

Acknowledgments.

This project is being carried out with funding provided by Science Foundation Ireland under grant number 11/RFP.1/CMS/3068

References

[1] Roberto Amadini, Maurizio Gabbrielli & Jacopo Mauro (2015): *SUNNY-CP: A Sequential CP Portfolio Solver*. In: *ACM Symposium on Applied Computing*, Salamanca, Spain, pp. 1861–1867, doi:10.1145/2695664.2695741.

[2] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs & K. Rustan M. Leino (2005): *Boogie: A Modular Reusable Verifier for Object-Oriented Programs*. In: *Formal Methods for Components and Objects: 4th International Symposium*, Amsterdam, The Netherlands, pp. 364–387, doi:10.1007/11804192_17.

[3] Mike Barnett, K. Rustan M. Leino & Wolfram Schulte (2004): *The Spec# Programming System: An Overview*. In: *Construction and Analysis of Safe, Secure and Interoperable Smart devices*, Marseille, France, pp. 49–69, doi:10.1007/978-3-540-30569-9_3.

[4] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds & Cesare Tinelli (2011): CVC4. In: *Computer Aided Verification*, Snowbird, UT, USA, pp. 171–177, doi:10.1007/978-3-642-22110-1_14.

[5] Clark Barrett, Aaron Stump & Cesare Tinelli (2010): *The Satisfiability Modulo Theories Library (SMT-LIB)*. Available at `http://www.smt-lib.org`.

[6] Clark Barrett & Cesare Tinelli (2007): *CVC3*. In: *Computer Aided Verification*, Berlin, Germany, pp. 298–302, doi:10.1007/978-3-540-73368-3_34.

[7] Dirk Beyer (2014): *Status Report on Software Verification*. In: *Tools and Algorithms for the Construction and Analysis of Systems*, Grenoble, France, pp. 373–388, doi:10.1007/978-3-642-54862-8_25.

[8] Dirk Beyer, Marieke Huisman, Vladimir Klebanov & Rosemary Monahan (2014): *Evaluating Software Verification Systems: Benchmarks and Competitions (Dagstuhl Reports 14171)*. *Dagstuhl Reports* 4(4), doi:10.4230/DagRep.4.4.1.

[9] Dirk Beyer, Stefan Löwe & Philipp Wendler (2015): *Benchmarking and Resource Measurement*. In: *Model Checking Software - 22nd International Symposium, SPIN 2015*, Stellenbosch, South Africa, pp. 160–178, doi:10.1007/978-3-319-23404-5_12.

[10] Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein & Josef Urban (2016): *A Learning-Based Fact Selector for Isabelle/HOL*. *Journal of Automated Reasoning*, pp. 1–26, doi:10.1007/s10817-016-9362-8.

[11] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond & Andrei Paskevich (2013): *Preserving User Proofs across Specification Changes*. In: *Verified Software: Theories, Tools, Experiments: 5th International Conference*, Menlo Park, CA, USA, pp. 191–201, doi:10.1007/978-3-642-54108-7_10.

[12] François Bobot, Jean-Christophe Filliâtre, Claude Marché & Andrei Paskevich (2015): *Let's verify this with Why3*. International Journal on Software Tools for Technology Transfer 17(6), pp. 709–727, doi:10.1007/s10009-014-0314-5.

[13] H. Borchani, G. Varando, C. Bielza & P. Larranaga (2015): *A survey on multi-output regression*. Data Mining And Knowledge Discovery 5(5), pp. 216–233, doi:10.1002/widm.1157.

[14] Thorsten Bormer, Marc Brockschmidt, Dino Distefano, Gidon Ernst, Jean-Christophe Filliâtre, Radu Grigore, Marieke Huisman, Vladimir Klebanov, Claude Marché, Rosemary Monahan, Wojciech Mostowski, Nadia Polikarpova, Christoph Scheben, Gerhard Schellhorn, Bogdan Tofan, Julian Tschannen & Mattias Ulbrich (2011): *The COST IC0701 Verification Competition 2011*. In: Formal Verification of Object-Oriented Software, Torino, Italy, pp. 3–21, doi:10.1007/978-3-642-31762-0_2.

[15] Thomas Bouton, Diego Caminha B. de Oliveira, David Déharbe & Pascal Fontaine (2009): *veriT: An Open, Trustable and Efficient SMT-Solver*. In: 22nd International Conference on Automated Deduction, Montreal, Canada, pp. 151–156, doi:10.1007/978-3-642-02959-2_12.

[16] Leo Breiman (2001): *Random Forests*. Machine Learning 45(1), pp. 5–32, doi:10.1023/A:1010933404324.

[17] David R. Cok, Aaron Stump & Tjark Weber (2015): *The 2013 Evaluation of SMT-COMP and SMT-LIB*. Journal of Automated Reasoning 55(1), pp. 61–90, doi:10.1007/s10817-015-9328-2.

[18] Sylvain Conchon & Évan Contejean (2008): *The Alt-Ergo automatic theorem prover*. Available at `http://alt-ergo.lri.fr/`.

[19] Leonardo De Moura & Nikolaj Bjørner (2008): *Z3: An Efficient SMT Solver*. In: Tools and Algorithms for the Construction and Analysis of Systems, Budapest, Hungary, pp. 337–340, doi:10.1007/978-3-540-78800-3_24.

[20] Yulia Demyanova, Thomas Pani, Helmut Veith & Florian Zuleger (2015): *Empirical Software Metrics for Benchmarking of Verification Tools*. In: Computer Aided Verification, San Francisco, CA, USA, pp. 561–579, doi:10.1007/978-3-319-21690-4_39.

[21] Bruno Dutertre & Leonardo de Moura (2006): *The Yices SMT Solver*. Available at `http://yices.csl.sri.com/papers/tool-paper.pdf`.

[22] Jean-Christophe Filliâtre & Andrei Paskevich (2013): *Why3 - Where Programs Meet Provers*. In: Programming Languages and Systems - 22nd European Symposium on Programming,, Rome, Italy, pp. 125–128, doi:10.1007/978-3-642-37036-6_8.

[23] Alexei Iliasov, Paulius Stankaitis, David Adjepon-Yamoah & Alexander Romanovsky (2016): *Rodin Platform Why3 Plug-In*. In: ABZ 2016: Abstract State Machines, Alloy, B, TLA, VDM, and Z: 5th International Conference, Linz, Austria, pp. 275–281, doi:10.1007/978-3-319-33600-8_21.

[24] Kalervo Järvelin (2012): *IR Research: Systems, Interaction, Evaluation and Theories*. SIGIR Forum 45(2), pp. 17–31, doi:10.1145/2093346.2093348.

[25] Cezary Kaliszyk & Josef Urban (2014): *Learning-Assisted Automated Reasoning with Flyspeck*. Journal of Automated Reasoning 53(2), pp. 173–213, doi:10.1007/s10817-014-9303-3.

[26] Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich & Benjamin Weiß (2011): *The 1st Verified Software Competition: Experience Report*. In: FM 2011: 17th International Symposium on Formal Methods, Limerick, Ireland, pp. 154–168, doi:10.1007/978-3-642-21437-0_14.

[27] Ekaterina Komendantskaya, Jónathan Heras & Gudmund Grov (2012): *Machine Learning in Proof General: Interfacing Interfaces*. In: 10th International Workshop On User Interfaces for Theorem Provers, Bremen, Germany, pp. 15–41, doi:10.4204/EPTCS.118.2.

[28] K. Rustan M. Leino (2010): *Dafny: An Automatic Program Verifier for Functional Correctness*. In: Logic for Programming, Artificial Intelligence, and Reasoning: 16th International Conference, Dakar, Senegal, pp. 348–370, doi:10.1007/978-3-642-17511-4_20.

[29] K. Rustan M. Leino & Michał Moskal (2010): *VACID-0: Verification of Ample Correctness of Invariants of Data-structures, Edition 0*. In: *Tools and Experiments Workshop at VSTTE*. Available at `https://www.microsoft.com/en-us/research/wp-content/uploads/2008/12/krml209.pdf`.

[30] David J Lilja (2000): *Measuring computer performance: a practitioner's guide*. Cambridge Univ. Press, Cambridge, UK, doi:10.1017/CBO9780511612398.

[31] Tom M. Mitchell (1997): *Machine Learning*. McGraw-Hill, New York, USA.

[32] Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent & Barry O'Sullivan (2008): *Using case-based reasoning in an algorithm portfolio for constraint solving*. In: *Irish Conference on Artificial Intelligence and Cognitive Science*, Cork, Ireland, pp. 210–216. Available at `http://homepages.laas.fr/ehebrard/papers/aics2008.pdf`.

[33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot & E. Duchesnay (2011): *Scikit-learn: Machine Learning in Python*. *Journal of Machine Learning Research* 12, pp. 2825–2830. Available at `http://dl.acm.org/citation.cfm?id=1953048.2078195`.

[34] Geoff Sutcliffe & Christian Suttner (1998): *The TPTP Problem Library*. *Journal Automated Reasoning* 21(2), pp. 177–203, doi:10.1023/A:1005806324129.

[35] Geoff Sutcliffe & Christian Suttner (2001): *Evaluating general purpose automated theorem proving systems*. *Artificial Intelligence* 131(1-2), pp. 39–54, doi:10.1016/S0004-3702(01)00113-8.

[36] Varun Tulsian, Aditya Kanade, Rahul Kumar, Akash Lal & Aditya V. Nori (2014): *MUX: algorithm selection for software model checkers*. In: *11th Working Conference on Mining Software Repositories*, Hydrabad, India, pp. 132–141, doi:10.1145/2597073.2597080.

[37] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, Björn Regnell & Anders Wesslén (2012): *Experimentation in Software Engineering*. Springer, New York, USA, doi:10.1007/978-3-642-29044-2.

[38] Lin Xu, Frank Hutter, Holger H. Hoos & Kevin Leyton-Brown (2008): *SATzilla: Portfolio-based Algorithm Selection for SAT*. *Journal of Artificial Intelligence Research* 32(1), pp. 565–606. Available at `http://dl.acm.org/citation.cfm?id=1622673.1622687`.

User Assistance Characteristics of the USE Model Checking Tool

Frank Hilken Martin Gogolla
University of Bremen, Computer Science Department
28359 Bremen, Germany
{fhilken,gogolla}@informatik.uni-bremen.de

The Unified Modeling Language (UML) is a widely used general purpose modeling language. Together with the Object Constraint Language (OCL), formal models can be described by defining the structure and behavior with UML and additional OCL constraints. In the development process for formal models, it is important to make sure that these models are (a) correct, i.e. consistent and complete, and (b) testable in the sense that the developer is able to interactively check model properties. The USE tool (UML-based Specification Environment) allows both characteristics to be studied. We demonstrate how the tool supports modelers to analyze, validate and verify UML and OCL models via the use of several graphical means that assist the modeler in interpreting and visualizing formal model descriptions. In particular, we discuss how the so-called USE model validator plugin is integrated into the USE environment in order to allow non domain experts to use it and construct object models that help to verify properties like model consistency.

1 Introduction

Model-Driven Engineering (MDE) is an approach to software development concentrating on models in contrast to traditional code-centric development approaches. Within MDE, models are frequently formulated in the Unified Modeling Language (UML) with accompanying formal restrictions expressed in the Object Constraint Language (OCL). UML models are visually specified with several diagram kinds emphasizing structural and behavioral system aspects. Visual model descriptions offer a great potential for a user-friendly development process. Naturally, tools must take up the challenge and provide interfaces that support the developer in an easy-going way.

The present paper studies formal system descriptions employing UML class diagrams that are restricted by OCL invariants. The feature set of UML class diagrams that is handled here and the employed OCL elements pose a formal semantics. We regard this combination of UML and OCL as a formal method. The aim of this contribution is to demonstrate how the development of formal UML and OCL models can be supported by a user-friendly interface. Employing this interface it is possible to verify properties like model consistency.

2 Preliminaries

2.1 Running Example in UML and OCL

In UML, class diagrams describe the structure of models with classes and class attributes, which are templates for 'things' and their properties, e.g. persons and their personal information.

EPTCS ??, 20??, pp. 1–7, doi:10.4204/EPTCS.??.??

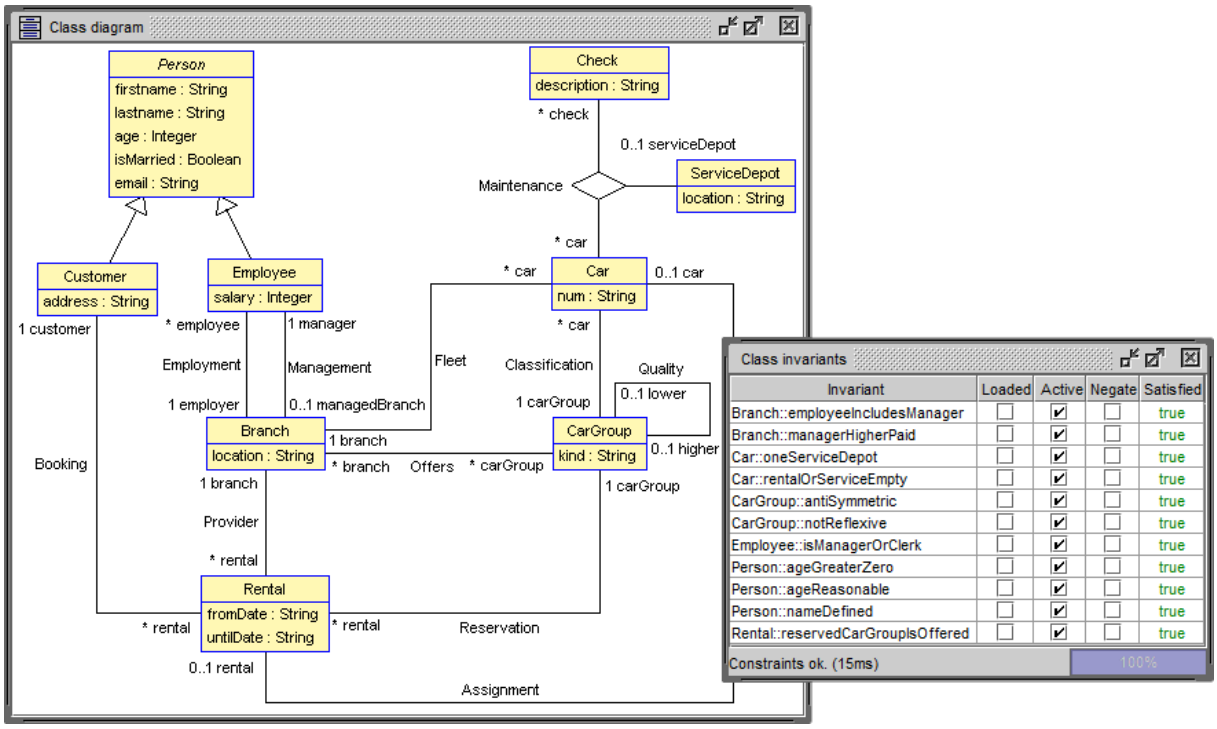


Figure 1: Car rental running example class diagram (left) and invariants (right).

Additionally, associations put the classes in relation with each other. Figure 1 (left) shows the running example model description. It shows a *Car Rental* model with cars that are assigned to branches and their maintenance history. Additionally, there is a categorization for the cars into car groups. Finally, customers can rent cars from the branches that are run by their employees. The model uses a wide variety of UML features.

The class diagram is instantiated to create actual scenarios to describe car rentals. These system states are represented by UML object diagrams. They are restricted by the semantics of the class diagram and must satisfy all model inherent constraints given by, e.g. generalizations, multiplicities or compositions (not present in this model).

In addition to the UML descriptions, OCL invariants are used to employ further restrictions on the model that are not expressible by UML alone. These constraints are pictured in Fig. 1 on the right. They handle further relations between classes and ensure that the model does not allow system states that are not intended. For example, in the car rental model, the categorization of the car groups shall be cycle free and all employees must be connected to a branch, which cannot be handled by multiplicities, because there are multiple ways to represent this relation (`Employment` and `Management`).

The goal is to create a model description that can represent all intended situations but not more. Using model checking tools, the models can then be checked for certain properties. Usually, these properties regard safety, but other concerns can be checked as well. A valid system state must satisfy all model inherent constraints as well as all concrete constraints given by the OCL invariants.

2.2 Model Verification with the USE Tool

In this paper, the *UML-based Specification Environment* tool (USE, [2]) is used together with the so-called *model validator* plugin that allows to generate system states for UML/OCL models based on a relational logic encoding [6]. There is earlier work showing features of the USE tool that help to analyze and debug OCL expressions, namely the evaluation browser [1], but we concentrate on the model validator and its model checking aspects in this work.

In order for the model validator to search for a valid system state, it needs several inputs:

- A description of the *model* in the form of a class diagram optionally enhanced with invariants given as OCL expressions, see Fig. 1.
- A *configuration*, which defines the search space by providing the domain of basic data types (Integer, String and Real) and lower and upper bounds for classes and associations, i.e. specifies how many objects of each class – or links of each association respectively – are required and how many are allowed at most. Furthermore, the configuration contains rules for the assignments of class attributes, e.g. restricting domain values for certain attributes specifically. Finally, it allows to disable or negate invariants.
- Optionally a *partial system state* can be instantiated before the validation process that is used as a base for the task. The model validator adds elements to the system state until it: (a) conforms the bounds given in the configuration; and (b) is a valid system state as defined by the model. This can be seen as a lower bound on the model level.
- Smaller, model independent parameters include the choice of the *SAT solver* and *bitwidth* for the encoding of the model.

The second bullet point, the configuration, previously required the modeler to edit a text file containing key-value pairs to setup values for certain keys. The keys are determined by the model. For example for each class and association a lower and an upper bound is expected. The required values are mostly numbers, but more complex constructs were required to specify, e.g. preexisting links. This process requires a deep understanding of the existing syntax to enter the values. Moreover, special values exist for some keys with different meanings, e.g. unlimited upper bounds. Even experienced users regularly required the manual of the configuration.

In the following sections, we explain how a new graphical user interface helps to simplify the configuration process and reduce the necessity for a separate manual to the tool. Furthermore, additional analysis features of the tool are presented to help identify potential problems with the verification process caused by the inputs.

3 Iterative Instantiation of the Running Example

We now study for our running example four use cases corresponding to four model validator configurations that result in UML object diagrams. The basic structure of the GUI contains three tabs for (1) the datatypes, (2) the classes and associations, and (3) the invariants. We iteratively build up the configuration to generate multiple object diagrams.

Datatypes First, only basic OCL types (e.g. `Integer` and `String`) are configured by giving bounds for their domain. For example, it is determined that integers may range from -10 to 10 and we want to use at most 10 string values, which are automatically generated.

Parts of the Model In the next step, the bounds for some of the classes of the model are configured and their relevant invariants are enabled. Figure 2 shows the configuration tabs and the resulting object diagram for this step. Here, the bounds for the classes `Customer`, `Employee` and `Branch` are set to `1..1` and all others are set to `0`. The class `Person` is abstract and, therefore, cannot be setup. All invariants based on these classes are enabled as well and none are negated. These settings are useful to setup certain model verification tasks, e.g. invariant independence [3]. In addition, the associations `Management` and `Employment` are enabled with a bound configuration of `1..1`. The object diagram shows the default string values for attributes, since these have not been specified further.

Full Model Once parts of the model have successfully been instantiated, a configuration enabling all elements is built and run through the model validator. With these smaller steps per configuration, there is less margin for errors and if there is one, it is easier detectable.

Application Specific Values Finally, application specific datatype values are employed for class attributes and basic types. They lead in the constructed object diagram to a state that seems more realistic, more domain specific than the previous object diagram. This also allows to specify that , e.g. address strings are not used for names.

4 Configuration GUI

First off, with the release of the configuration GUI the interface of configuration files was extended to allow storing multiple configurations in one file. These configurations can be named individually and the configuration GUI offers operations to manage them, e.g. cloning configurations, renaming or deleting them etc. This allows for an easy iterative construction of the configurations as shown in Sect. 3. The GUI also offers common file system operations to deal with the generated configuration files. For convenience, a configuration file with the same base name as the loaded model is automatically opened if one exists.

The GUI is split into three tabs that each cover a part of the configuration.[1] The first tab is the basic types tab in which the domains for the basic types of OCL are defined. The domains can be defined as ranges or specific values that will be used by the model validator.

The second tab defines the model dependent bounds and domains. These include bounds for classes, attributes and associations as seen in the top of Fig. 2. The GUI only requires the values for the specific settings which eliminates the need to know the syntax for each model element, which makes the creation of configurations simpler and faster. Further, abstract classes cannot be setup, which is represented with non-editable fields in the GUI. If a value cannot be parsed, it is highlighted red and the modeler immedietly sees problems in the configuration. Features that require expert knowledge are hidden behind a checkbox.

The final, third tab, configures the invariants. Invariants can be individually deactivated and negated, which is required for certain verification tasks [3].

5 Analysis of Potential Modeling Problems

So far, we have shown how the user is assisted by the graphical user interface to setup the configuration of a verification task. However, besides a bad configuration, other problems can interfere with the checking process, in particular problems that the user is not aware of.

[1] A detailed explanation of all three tabs in detail including screenshots can be found in [4].

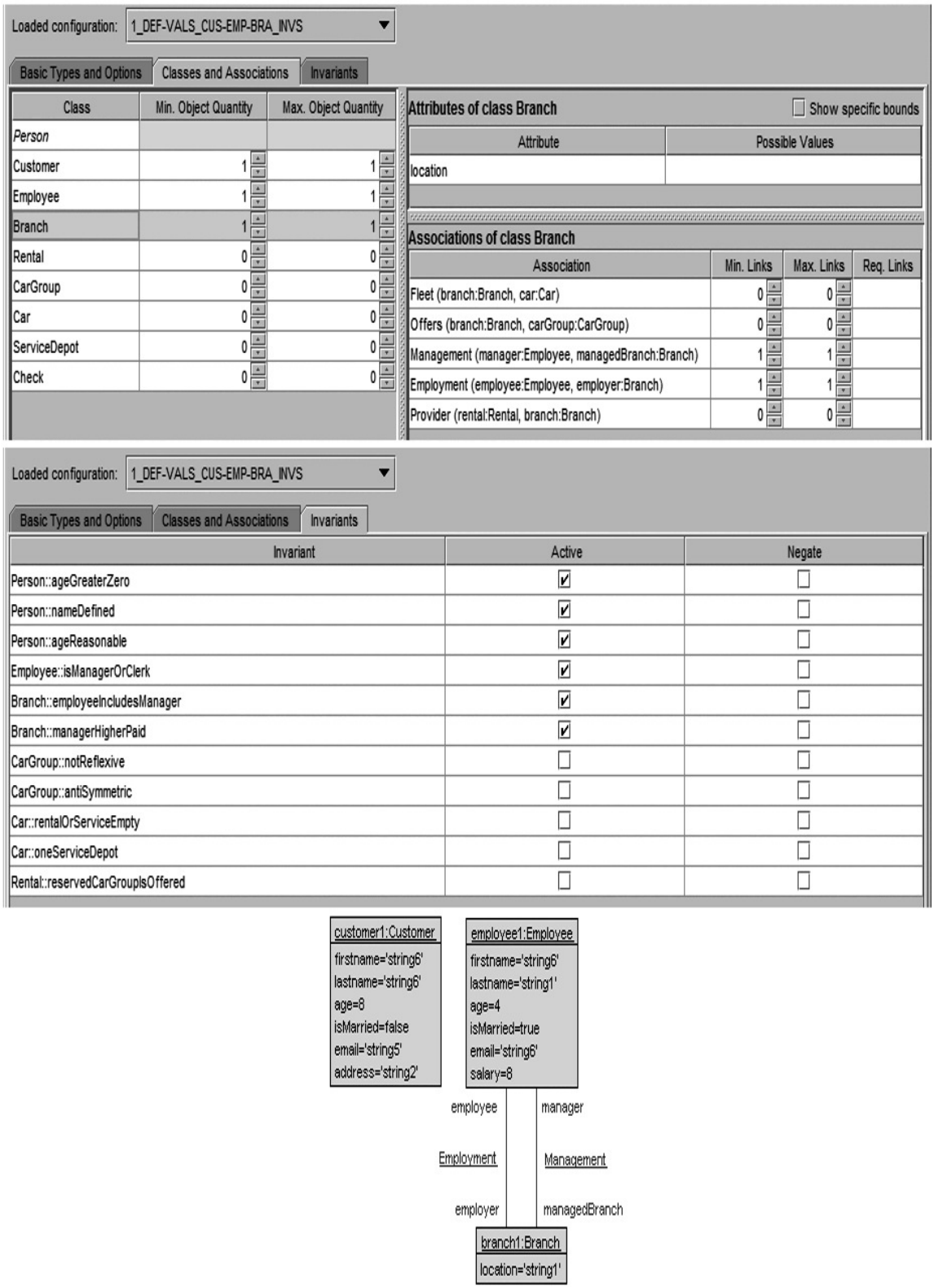


Figure 2: Configuration of parts of the model considering only a few classes and associations.

UML and OCL are rich languages filled with features for all kinds of purposes. Trying to support all of them is not only a lot of work, but also reduces the efficiency of the tools, the more features they support [5]. Therefore, it is common practice to restrict UML and OCL verification engines to a subset of the languages. This results in some features being completely unsupported and others only having limited support. Both categories pose problems to the users of the tools. If there are no means in place to detect the limitations, the outcome might differ from the user's expectations and it is not feasible to keep track of all limitations from long tool manuals.

The verification engine of the USE tool, the model validator, has good support for UML and OCL, but also has limits. The underlying solving engine of the model validator is based on relational logic, which has great support for set operations and, thus, the integration of the OCL `Set` collection type is extensive. Adding support for the other collection types `Bag`, `Sequence` and `OrderedSet` would pose a significant overhead though, i.e. will be less efficient. This restriction to the `Set` collection type is particularly problematic for OCL navigation expressions. This expression allows to navigate the classes of the model using associations, more precisely the roles visible in Fig. 1. Usually a 1-n navigation results in a set of elements, because at most one link is allowed between two objects, but under certain circumstances – namely starting with a set of objects rather than a single one – the navigation results in the duplicate preserving `Bag` type, because the result might contain the same value multiple times after the navigation. Due to the implicit nature of this effect and the strict interpretation of bags as sets in the model validator, simple expressions might already suffer unintended side effects. To help identify those potential problems, the occurrences are made visible to the modeler via a warning.

```
WARNING: Collect operation '[...].employee.age' results in unsupported type 'Bag'. It
    will be interpreted as 'Set'.
```

The implicit type change from `Set` to `Bag` in OCL, which is consequently interpreted as `Set` by the model validator, brings more potential problems with it. Most OCL collection operations are defined on all collection types, but the results are different. Consider the operation `sum()`, which sums all integer elements of a collection. Here, the implicit conversion from `Set` to `Bag` is usually helpful when collecting, for example, the ages of persons to calculate an average. However, the interpretation as a `Set` does not work in this situation. To assist the user, the model validator checks for these situations and warns the modeler about this potential problem, which only the modeler can decide whether it needs to be addressed or not.

```
WARNING: The evaluation of sum expression '[...].employee.age->sum()' might be wrong
    if source contains duplicates (Collection is interpreted as Set).
```

Other problems might arise from contradictions in the model itself. Navigation expressions through the model can become quite long and obscure the resulting type. In these situations, typecasts like `oclAsSet()` need to be used to be able to compare differing types, but the textual representation of OCL alone is often insufficient to recognize such disparities. USE is able to structurally analyze the expressions in the model for type contradictions and gives hints about (sub)expressions that were determined to be contradicting, resulting in constant values.

```
WARNING: Expression 'Set{ 1 } = Bag{ 1 }' can never evaluate to true because
    'Set(Integer)' and 'Bag(Integer)' are unrelated.
```

Finally, the underlying solving engine is bound to a bitwidth that has to be specified with the verification task. Setting the bitwidth as small as necessary increases the efficiency of the

tool, thus we leave the task to the user to choose an appropriate bitwidth. But if the bitwidth is chosen too small, undefined behavior occurs when dealing with arithmetic operations exceeding the bitwidth. Finally, if the user is not aware of – or forgets – that the bitwidth is specified in two's-complement, off-by-one errors can occur. In order to alleviate the problem, the model validator analyzes the configuration and model description for integer literals and checks them against the given bitwidth. If it is determined that the chosen bitwidth is too small, a warning is displayed including the appropriate bitwidth for the current model and configuration.

```
WARNING: The configured bitwidth is too small for the property Integer max value
    (237). Required bitwidth: 9 or greater.
```

6 Conclusion and Future Work

We have presented the USE tool together with its model validator plugin and have shown the steps necessary to apply model checking to given UML/OCL models. Furthermore, the simplifications of the process by integrating the graphical user interface have been discussed and the possibilities of the configuration GUI and the coverage mode are demonstrated. Finally, the possibilities of the model validator plugin to detect potential problems have been demonstrated to guide users in finding incompatibilities in their models.

Besides the configuration of the model domains and bounds, we have presented more aspects that have to be setup before a system state can be generated including the verification task itself, e.g. by manipulating the invariants. Future work should concentrate on the simplification of all steps of the setup and provide easy interfaces for each of them. Additionally, the evaluation of the configuration GUI and other presented interfaces is an ongoing process and new assistance features are constantly added and improved.

Acknowledgement. We thank Subi Aili for his contributions to the configuration GUI – ideas and implementation – in his diploma thesis.

References

[1] Jens Brüning, Martin Gogolla, Lars Hamann & Mirco Kuhlmann (2012): *Evaluating and Debugging OCL Expressions in UML Models*. In Achim D. Brucker & Jacques Julliand, editors: *Proc. 6th Int. Conf. Tests and Proofs (TAP 2012)*, Springer, Berlin, LNCS 7305, pp. 156–162.

[2] Martin Gogolla, Fabian Büttner & Mark Richters (2007): *USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming* 69, pp. 27–34.

[3] Martin Gogolla, Mirco Kuhlmann & Lars Hamann (2009): *Consistency, Independence and Consequences in UML and OCL Models*. In Catherine Dubois, editor: *Tests and Proofs, TAP*, *Lecture Notes in Computer Science* 5668, Springer, pp. 90–104.

[4] Frank Hilken & Martin Gogolla (2016): *User Assistance Characteristics of the USE Model Checking Tool*. Technical Report, University of Bremen. Available at `http://www.db.informatik.uni-bremen.de/publications/intern/HG2016.pdf`.

[5] Frank Hilken, Philipp Niemann, Martin Gogolla & Robert Wille (2014): *Filmstripping and Unrolling: A Comparison of Verification Approaches for UML and OCL Behavioral Models*. In Martina Seidl & Nikolai Tillmann, editors: *Tests and Proofs, TAP*, *LNCS* 8570, Springer, pp. 99–116.

[6] Mirco Kuhlmann & Martin Gogolla (2012): *From UML and OCL to Relational Logic and Back*. In Robert France, Juergen Kazmeier, Ruth Breu & Colin Atkinson, editors: *Proc. 15th Int. Conf. Model Driven Engineering Languages and Systems (MoDELS'2012)*, Springer, Berlin, LNCS 7590, pp. 415–431.

Industrial Experience Report on the Formal Specification of a Packet Filtering Language Using the K Framework

Gurvan LE GUERNIC
DGA Maîtrise de l'Information
35998 Rennes Cedex 9, France

Benoit COMBEMALE
José A. GALINDO
INRIA RENNES – BRETAGNE ATLANTIQUE
Campus universitaire de Beaulieu
35042 Rennes Cedex, France

Many project-specific languages, including in particular filtering languages, are defined using non-formal specifications written in natural languages. This leads to ambiguities and errors in the specification of those languages. This paper reports on an industrial experiment on using a tool-supported language specification framework ($\mathbb{K}$) for the formal specification of the syntax and semantics of a filtering language having a complexity similar to those of real-life projects. This experimentation aims at estimating, in a specific industrial setting, the difficulty and benefits of formally specifying a packet filtering language using a tool-supported formal approach.

1 Introduction

Packet filtering (accepting, rejecting, modifying or generating packets, i.e. strings of bits, belonging to a sequence) is a recurring problematic in the domain of information systems security. Such filters can serve, among other uses, to reduce the attack surface by limiting the capacities of a communication link to the legitimate needs of the system it belongs to. This type of filtering can be applied to network links (which is the most common use), product interfaces, or even on the communication buses of a product. If the filtering policy needs to be adapted during the deployment or operational phases of the system or product, it is often required to design a specific language $\mathscr{L}$ (syntax and semantics) to express new filtering policies during the lifetime of the system or product. This language is the basis of the filters that are applied to the system or product. Hence, it plays an important role in the security of this system or product. It is therefore important to have strong guarantees regarding the expressivity, precision, and correctness of the language $\mathscr{L}$ (meaning that everything that need to be expressed can, and that everything that can be expressed has the most obvious semantics). Those guarantees can be partly provided by a formal design (and development) process.

Among diverse duties, the DGA (Direction Générale de l'Armement, a french procurement agency) is involved in the supervision of the design and development of filtering components or products. Those filters come in varying shapes and roles. Some of them are network apparatuses filtering standard Internet protocol packets (such as firewalls); while others are small parts of integrated circuits filtering specific proprietary packets transiting on computer buses. Their common definition is: "a tool sitting on a communication channel, analyzing the sequence of packets (strings of bits with a beginning and an end) transiting on that channel, and potentially dropping, modifying or adding packets in that sequence". Whenever the filtering algorithm applied is fixed for the lifetime of the component or product, this algorithm is often "hard coded" into the component or product with the potential addition of a configuration file allowing to slightly alter the behavior of the filter. However, sometimes the filtering algorithm to apply may depend on the deployment context, and may have to evolve during the lifetime of the component or product to adapt to new uses or attackers. In this case, it is often necessary to be able to easily

EPTCS ??, 20??, pp. 1–15, doi:10.4204/EPTCS.??.??

write new filtering algorithms for the specific product and context. Those algorithms are then often described using a Domain Specific Language (DSL) that is designed for the expression of a specific type of filters for a specific product. The definition of the syntax and semantics of this DSL is an important task. This DSL is the link between the filtering objectives and the process that is really applied on the packet sequences. Often, language specifications (when there is one) are provided using natural language. In the majority of cases, this leads to ambiguities or errors in the specification which propagate to implementations and final user code. This is for example the case for common languages such as C/C++ or Java™ [12].

> "Unfortunately, the current specification has been found to be hard to understand and has subtle, often unintended, implications. Certain synchronization idioms sometimes recommended in books and articles are invalid according to the existing specification. Subtle, unintended implications of the existing specification prohibit common compiler optimizations done by many existing Java virtual machine implementations. [...] Several important issues, [...] simply aren't discussed in the existing specification."
>
> JSR-133 expert group [12]

Some of those ambiguities, as the memory model of multi-threaded Java™ programs [12], required a formal specification in order to be solved.

This paper is an industrial experience report on the use of a tool-supported language specification framework (the $\mathbb{K}$ framework) for the formal specification of the syntax and semantics of a filtering language having a complexity similar to those of real-life projects. The tool used to formally specify the DSL is introduced in Sect. 2. For confidentiality reasons, in order to be allowed by the DGA to communicate on this experimentation, the language specified for this experiment is not linked to any particular product or component. It is a generic packet filtering language that tries to cover the majority of features required by packet filtering languages. This language is introduced in Sect. 3 while its formal specification is described in Sect. 4. This language is tested in Sect. 5 by implementing and simulating a filtering policy enforcing a sequential interaction for a made-up protocol similar to DHCP. Before concluding in Sect. 7, this paper discusses the results of the experimentation in Sect. 6.

2 Introduction to the $\mathbb{K}$ Framework

Surprisingly, even if it is a niche for tools, there exists quite a number of tools specifically dedicated to the formal *specification* of languages (our focus in this work is on specifying rather than implementing DSLs). Those tools include among others: PLT Redex [6, 13], Ott [23], Lem [19], Maude MSOS Tool [3], and the $\mathbb{K}$ framework [20, 26]. All those tools focus on the (clear formal) specification of languages rather than their (efficient) implementation, which is more the focus of tools and languages such as Rascal [16, 2, 15] or its ancestor The Meta-Environment [14, 25], Kermeta [9, 10], and others. PLT Redex is based on reduction relations. PLT Redex is an extension (internal DSL) of the Racket programming language [7]. Ott and Lem are more oriented towards theorem provers. Ott and Lem allow to generate formal definitions of the language specified for Coq, HOL, and Isabelle. In addition, Lem can generate executable OCaml code. Ott is more programming language syntax oriented, while Lem is a more general purpose semantics specification tool. Ott and Lem can be used together in some contexts. The Maude MSOS Tool, whose development has stopped in 2011, is based on an encoding of modular structural operational semantics (MSOS) rules into Maude. Similarly to the Maude MSOS Tool, the $\mathbb{K}$ framework is based on rewriting and was also originally implemented on top of Maude.

The goal set for the experiment reported in this paper is to estimate the difficulty and benefits for an average engineer (i.e. an engineer with education and experience in computer science but no specific knowledge in formal language semantics) to use an "appropriate" tool for the formal specification of a packet filtering language. The "appropriate" tool needs to: be easy to use; be able to produce (or take as input) "human readable" language specifications; provide some level of correctness guarantees for the language specified; and be executable (simulatable) in order to test (evaluate) the language specified. The $\mathbb{K}$ framework seems to meet those requirements and has been chosen to be the "appropriate" tool after a short review of available tools. As there has been no in depth comparison of the different tools available, there is no claim in this paper that the $\mathbb{K}$ framework is better than the other tools, even in our specific setting.

This section introduces the $\mathbb{K}$ framework [21] by relying on the example of a language allowing to compute additions over numbers using Peano's encoding [8]. The $\mathbb{K}$ source code of this language specification is provided below.

```
module PEANO-SYNTAX
  syntax Nb ::= "Zero" | "Succ" Nb
  syntax Exp ::= Nb | Id | Exp "+" Exp      [strict,left]
  syntax Stmt ::= Id ":=" Exp ";"           [strict(2)]
  syntax Prg ::= Stmt | Stmt Prg
endmodule

module PEANO imports PEANO-SYNTAX
  syntax KResult ::= Nb

  configuration
    <env color="green"> .Map </env>
    <k color="cyan"> $PGM:K </k>

  rule N:Nb + Zero => N
  rule N1:Nb + Succ N2:Nb => ( Succ N1 ) + N2

  rule
    <env> ... Var:Id |-> Val:Nb ... </env>
    <k> ( Var:Id => Val:Nb ) ... </k>

  rule
    <env> Rho:Map (.Map => Var |-> Val ) </env>
    <k> Var:Id := Val:Nb ; => . ... </k>
    when notBool (Var in keys(Rho))

  rule
    <env> ... Var |-> ( _ => Val ) ... </env>
    <k> Var:Id := Val:Nb ; => . ... </k>

  rule S:Stmt P:Prg => S ~> P [structural]
endmodule
```

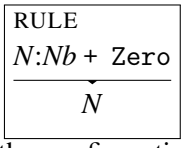
A $\mathbb{K}$ definition is divided into three parts: the *syntax* definition, the *configuration* definition, and the *semantics* (rewriting rules) definition. The definition of the language *syntax* is given in a module whose name is suffixed with "-SYNTAX". It uses a BNF-like notation [1, 17]. Every non-terminal is introduced by a `syntax` rule. For example, the definition of the notation for numbers (`Nb`) in this language, provided on line 2, is equivalent to the definition given by the regular expression "`(Succ)`* `Zero`".

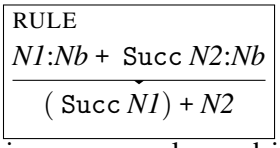
The *configuration* definition part is introduced by the keyword `configuration` and defines a set of (potentially nested) cells described in an XML-like syntax. This configuration describes the "abstract machine" used for defining the semantics of the language. The initial state (or configuration) of the abstract machine is the one described in this configuration part. The parsed program (using the syntax definition of the previous part) is put in the cell containing the `$PGM` variable (of type `K`). For the `Peano` language, the `env` cell is used to store variable values in a map initially empty (`.Map` is the empty map). From this definition, the $\mathbb{K}$ framework can produce a graphical representation of the configuration, provided in Fig. 1

Figure 1: Peano's $\mathbb{K}$ configuration

The *semantics* definition part is composed of a set of rewriting rules, each one of them introduced by the keyword `rule`. In the $\mathbb{K}$ source file, rules are roughly denoted as "*CCF* => *NCF*" where *CCF* and *NCF* are configuration fragments. The meaning of "*CCF* => *NCF*" can be summarized as: if *CCF* is a fragment of the current abstract machine state (or configuration) then the rule may apply and the fragment matching *CCF* in the current configuration would then be replaced by the new configuration fragment *NCF*. In order to increase the expressivity of rules, *CCF* may contain free variables that are reused in expressions in *NCF*. If a specific valuation of the free variables *V* in *CCF* allows a fragment of the current configuration to match *CCF*, then this fragment may be replaced by *NCF* where the variables *V* are replaced by their matching valuation.

The rules for addition over numbers (`Nb` and not `Exp`), on lines 15 and 16, follows closely this representation. For those rules, *CCF* is a program fragment that can be matched in any cell of the configuration. For those two rules, the $\mathbb{K}$ framework can then produce the following graphical representations:





For other rules, the configuration fragment matching is more complex and involves precise configuration cells that are explicitly identified. In order to compress the representation, *CCF* and *NCF* are not stated separately anymore. The common parts are stated only once, and the parts differing are again denoted "CCF_i => NCF_i", where CCF_i is a sub-fragment in *CCF* and NCF_i is the corresponding sub-fragment in *NCF*. Cells that have no impact on a rule *R* and are not impacted by *R* do not appear explicitly in the rule. Cells heads and tails (potentially empty) that are not modified by a rule can be denoted "...", instead of using a free variable that would not be reused.

For example, the rule which starts on line 18is the rule used to evaluate variables. The current configuration needs to contain a mapping from a variable `Var` to a value `Val` ("`X |-> V`" denotes a mapping from `X` to `V`) somewhere in the map contained in the `env` cell. It also needs to contain the variable `Var` at the beginning of cell `k`. This rule has the effect of replacing the instance of `Var` at the beginning of cell `k` by the value `Val`. For this rule, the $\mathbb{K}$ framework generates the graphical representation given in Fig. 2.

The last rule on line 31 involves other internal aspects of the $\mathbb{K}$ framework. It roughly states that, in order to evaluate a statement `S` followed by the rest `P` of the program, `S` must first be evaluated to a

`KResult` (defined on line 9) and then `P` is evaluated.

3 GPFL Context

The language specified in the experiment reported in this paper, named GPFL, is a generic packet filtering language. For confidentiality reasons, GPFL is not a language actually used in any specific real product. GPFL has been made-up in order to be able to communicate on the experimentation on tool supported formal specification of filtering languages reported in this paper. However, GPFL covers the majority of features needed in packet filtering languages dealt with by the DGA. GPFL can be seen as the "mother" of the majority of packet filtering languages.

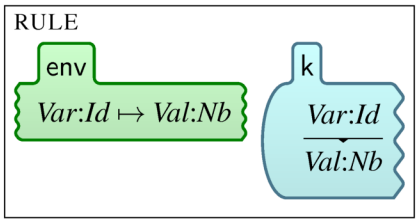
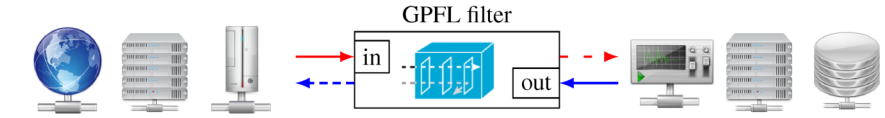


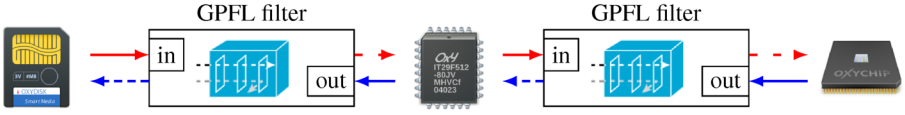
Figure 2: Peano's $\mathbb{K}$ rule for variables

GPFL aims at expressing a wide variety of filters. Those filters can be placed at the level of network, interfaces, or even communication buses between electronic components. They can be applied on standard protocols such as IP, TCP, UDP, ... or on proprietary protocols, which are more common for component communication protocols. However, all those filters are assumed to be placed on a communication link. Messages (packets) that get through the filter can only get through in two ways, either "going in" or "going out"; there is no switching taking place in GPFL filters. Those different use cases are illustrated in Fig. 3.



(a) Network filtering

(b) Interface filtering



(c) Bus filtering

Figure 3: Use cases for GPFL-based filters

GPFL focuses on the internal logic of the filter. Decoding and encoding of packets is assumed to be handled outside of GPFL programs (filters), potentially using technologies such as ASN.1 [11, 5]. For GPFL programs, a packet is a record (a set of valued fields). A GPFL program (dynamically) inputs a sequence of records and outputs a sequence of records. Figure 4 describes the architecture of GPFL-based filters. An incoming packet (on either side) is first parsed (decoded) before being handed over to

the GPFL program. If the packet can not be parsed, depending on the type of filter (white list or black list), the packet is either dropped or passed to the other side without going through the GPFL program. Any packet (record) output by the GPFL program (on either side) is encoded before being sent out. In addition, the GPFL program can generate alarms due to packets not complying with the encoded filtering policy.

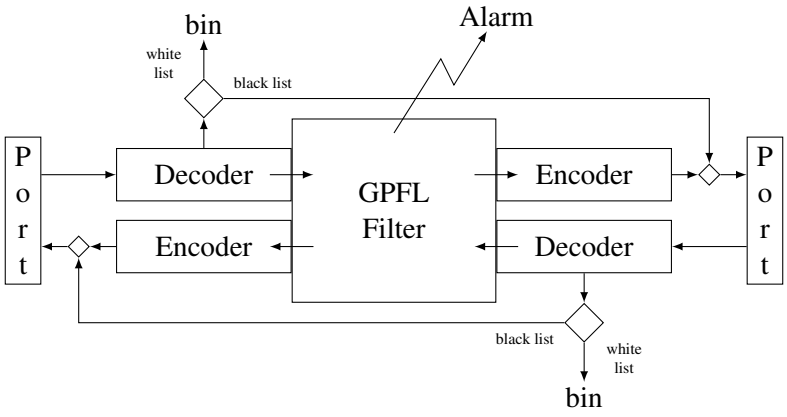


Figure 4: Architecture of GPFL-based filters

The GPFL language must allow to: drop, modify or accept the current packet being filtered; generate new packets; and generate alarms. GPFL must allow to base the decision to take any of those actions on information pieces concerning the current packet being filtered and previously filtered packets. Those information pieces must include: some timing information, current or previous packets directions through the filter ("in" or "out"), and characteristics of current or previous packets including field values and computed properties such as, for example, a packet "type" or total length. The computation of those properties and decoding of packet fields is outside of the scope of GPFL; it is left to the decoders.

In order to gradually build a decision, GPFL must allow to interact with variables (reading, writing, and computing expressions) and automata (triggering a transition in an automaton and querying its current state). The intent for automata is to be used to track the current step of sessions of complex protocols. GPFL must allow to combine filtering statements using: sequential control statements (executing two statements in sequence); conditional control statements (executing a statement only if a condition is true); iterating control statements (repeatedly executing a statement for a fixed number of repetitions). There is no requirement for a loop (or while) statement whose exit condition is controlled by an expression recomputed after every iteration. For the experiment reported in this paper (on formal specification of a filtering language), the iterating statement is considered sufficient for the intended use of GPFL and close enough to a loop statement from a semantics point of view, while exhibiting interesting properties for future analyses (for example, any GPFL program terminates).

4 GPFL's Specification

Due to lack of space, GPFL's specification and testing is only summarized in this paper. However, a full specification of GPFL and a testing section can be found in the companion technical report [18].

Syntax. To the exception of expressions and expression fragments, GPFL's syntax is formally defined by the $\mathbb{K}$ source fragment provided below.

```
18  syntax Cmd ::= "nop" | "accept" | "drop" | "send(" Port "," Fields ")"
               |  "alarm(" Exp ")"                                    [strict(1)]
20             |  "set(" Id "," Exp ")"                               [strict(2)]
               |  "newAutomaton(" String "," AutomatonId ")"
22             |  "step(" AutomatonId "," Exp "," Stmt ")"           [strict(2)]
    syntax Stmt ::= Cmd
24              |  "cond(" Exp "," Stmt ")"                          [strict(1)]
                |  "iter(" Exp "," Stmt ")"                          [strict(1)]
26              |  "newInterrupt(" Int "," Bool "," Stmt ")"
                |  Stmt Stmt                                         [right]
28              |  "{" Stmt "}"                                      [bracket]

30  syntax AutomataDef ::= "AUTOMATA" String AutomataDefTail
    syntax AutomataDefTail ::= "init" "=" AStateId ATransitions | ATransitions
32  syntax ATransitions ::= List{ATransition,""}
    syntax ATransition ::= AStateId "-" AEvtId "->" AStateId
34  syntax AStateId ::= String
    syntax AEvtId ::= String
36  syntax InitSeq ::= "INIT" Stmt
    syntax PrologElt ::= AutomataDef | InitSeq
38  syntax Prologues ::= PrologElt | PrologElt Prologues

40  syntax Program ::= "PROLOGUE" Prologues "FILTER" Stmt
```

A GPFL program is composed of a prologue, executed only once in order to initialize the execution environment, and a filter statement, executed once for every incoming packet. A prologue is composed of automaton kind definitions and initialization sequences. An automaton kind definition specifies an identifier K, an initial state for automata of kind K and a set of transitions for automata of kind K. A transition definition is composed of: two automaton states F and T, and an automaton event that triggers the transition from F to T.

A GPFL statement is composed of GPFL commands or statements combined sequentially. Some statements can be guarded by an expression and executed only if that expression evaluates to true (`cond`). Some statements (`iter`), associated with an expression e, are exectued v times, where v is the value of e before the first iteration. Finally, `newInterrupt` statements register a statement to be executed in the future, potentially periodically.

GPFL commands are the basic units having an effect on the execution environment. The `nop` command has no effect and serves mainly as a place holder. The `accept`, resp. `drop`, command states to accept, resp. drop, the current packet and stop the filtering process for this packet. The `send` command sends a packet on one of the ports. The `alarm` command generates a message on the alarm channel. The `set` command sets the value of a variable. The `newAutomaton` command initializes an automaton of the provided kind, and assigns this newly created automaton to the provided identifier. The `step` command tries to trigger an automaton transition by sending an event e to an automaton a. If there is no transition from the current state of a triggered by the event e, then the associated statement is executed.

Semantics The full formal specification of GPFL's semantics can be found in the companion technical report [18]. GPFL's semantics rules are defined on the configuration presented graphically in Fig. 5. The `prg` cell contains the GPFL program. After initialization of the program, automaton kind definitions are stored in the `automatonKindDefs` cell and the `filter` cell contains the filter (GPFL statement)

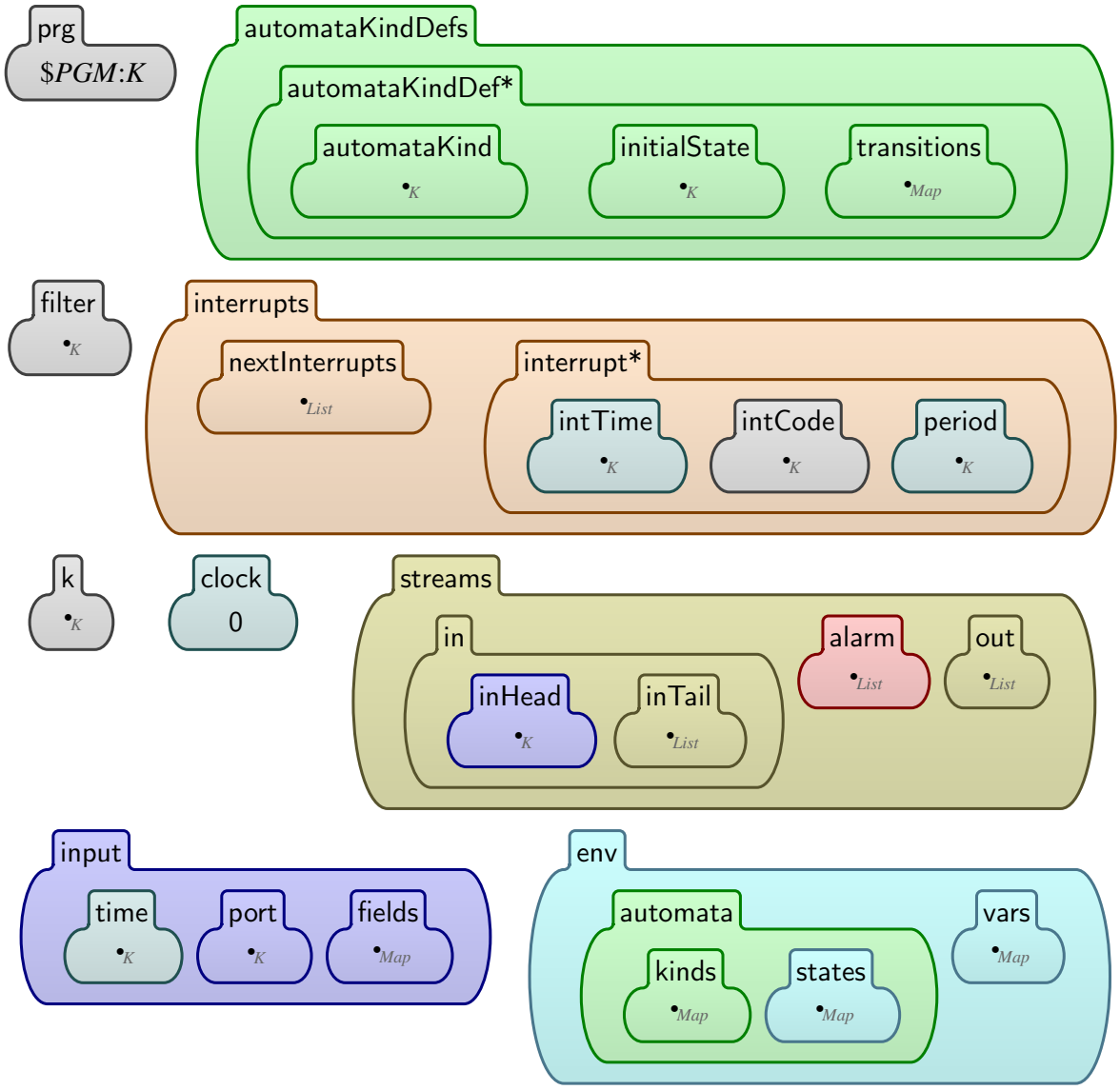


Figure 5: $\mathbb{K}$ configuration of GPFL

that is to be executed for every packet. The `interrupts` cell contains a set of interrupt definitions (`interrupt*`). An interrupt is a triplet composed of: the time when the interrupt is to be triggered, the code (statement) to be executed, and a "Time" value equal to the interruption period for a periodic interruption (or nothing for a non-periodic interruption). In addition, the `interrupts` cell contains an ordered list of the next "times" when an interrupt is to be executed. The `clock` cell registers the current "time". The configuration also contains a `k` cell that holds the GPFL statement under execution. Each time a new packet is input, the content of the `k` cell is replaced by the content of the `filter` cell, and the newly arrived packet is stored in the `input` cell with its arrival time and port.

Packets are input from the `streams` cell which contains: the packet input stream divided into the next packet to arrive (`inHead`) and the rest of the stream (`inTail`); the packet output stream; and the alarm output stream. In the input stream, resp. output stream, packets arriving, resp. leaving, on both

ports are mixed together, but contains information on the port of entry, resp. exit. Some choices made to represent those streams are not an intrinsic part of GPFL's formal specification. The division of the input stream into a head and a tail is such a choice. Those choices are made in order to be able to execute the specification. It is then required to implement, in the $\mathbb{K}$ framework, a mechanism to retrieve and parse strings describing packet sequences sent to the filter. In order to help distinguish between the formal specification of GPFL and the mechanisms put in place to execute it, whenever possible, implementation choices, such as the format of strings describing packets, are defined in another file which is loaded in the main specification file with the `require` instruction.

Finally, the `env` cell is the main dynamic part of the execution environment. It corresponds to a "record" of maps that associate: automaton kind and current state to automaton identifiers (`automata` cell); and values to variables.

5 Testing GPFL's Specification

GPFL's specification, introduced above and contained in the companion technical report [18], is not necessarily perfect. By a matter of fact, imperfections of GPFL's specification are of interest to the experimentation reported in this paper. Indeed, the goal of the experimentation is to see how a tool such as the $\mathbb{K}$ framework can help to spot and correct imperfections in filtering language specifications. One way to do so is by "testing" the new language specified, which is possible if the framework used to specify the language supports the execution or simulation of language specifications, which is the case for the $\mathbb{K}$ framework.

The test scenario used assumes a network of clients and servers. The clients request resources to servers using a made-up protocol, called "DHCP cherry", summarized in Fig. 6. The test scenario as-

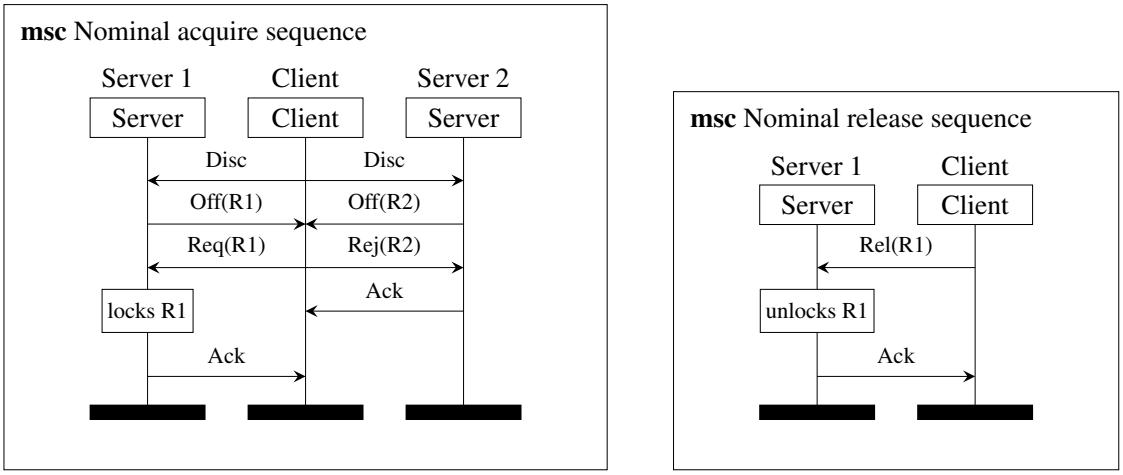


Figure 6: Nominal packet sequences of DHCP cherry protocol

sumes that servers behave poorly when interacting concurrently with different clients. The objective of the test scenario is then to filter communications in front of servers in order to prevent any concurrent client-server interactions with any given server. This test scenario is obviously made-up for this experimentation, which is a requirement due to confidentiality issues. However, it is still covering the most frequently used features of filtering languages similar to GPFL, while remaining simple enough for a first experimentation.

From the point of view of servers, non-concurrent interactions are sequential instances of only three generic atomic packet sequences. Those atomic packet sequences are the ones accepted by the automaton in Fig. 7. In this automaton, "in:*MP*", resp. "out:*MP*", is a transition trigger matching any incoming

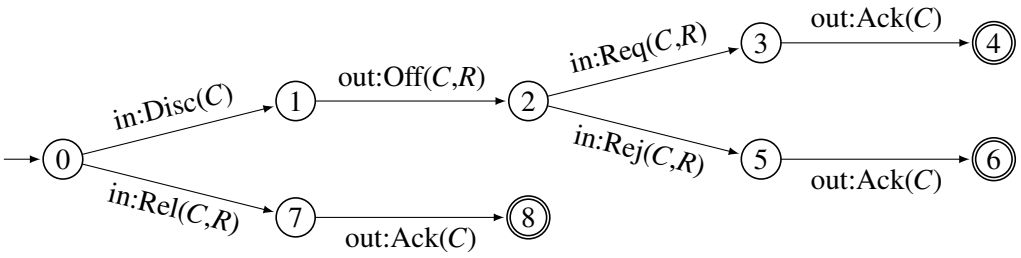


Figure 7: Automaton of server-side atomic packet sequences

packet (from the rest of the network to the server), resp. outgoing packet, matching packet pattern *MP*. *C*, resp. *R*, is a client, resp. ressource, identifier variable. *C*, resp. *R*, has to be instantiated in the same way (have the same value) for any packet of the same atomic packet sequence accepted by the automaton. The automaton of Fig. 7 is refined into a filtering policy automaton described in Fig. 8. Variables *C* and *R* have the same constraints as for the automaton of Fig. 7. The variable "$*$" matches any value, packet pattern "out:$*$" matches any outgoing packet, and packet pattern "out:$*$ - Ack(*C*)" matches any outgoing packet except Ack(*C*). This filtering policy accepts every outgoing packet; thus having no

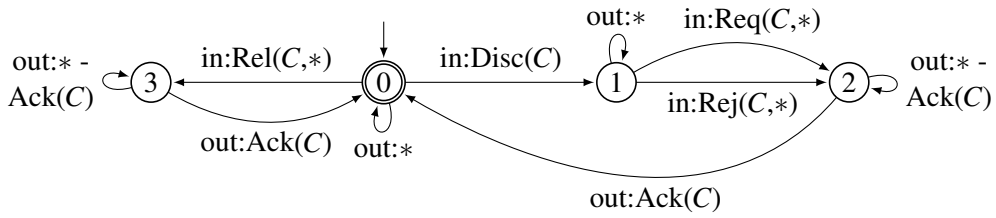


Figure 8: Filtering policy automaton

effect on the packets generated by the server. For incoming packets, if the current state of the automaton has no transition whose trigger matches the packet then the packet is discarded; otherwise, the packet is accepted and the associated transition is triggered. This filtering policy assumes that clients comply with the DHCP cherry protocol and ensures only that the filtered server only interacts sequentially with clients. If there is no idle server ready to receive a packet from a client, this client gets no answer and is expected to retry later.

This policy has been encoded in GPFL and executed using the following command (in Linux Bash): "`krun dhcp.gpfpl < dhcp_input-dataset.txt > dhcp_output.txt`" where the file `dhcp_input-dataset.txt` contains a sequence of packets already "parsed" (decoded packets, Fig. 4) input to the filter. The output of the simulation of the code (`dhcp.gpfpl`) written in the specified language (GPFL) is written in `dhcp_output.txt`.

6 Discussion on the Experimentation

The primary goal of this paper is not to set out the filtering policy described in Sect. 5 or, even, GPFL's specification described in Sect. 4. This paper is an industrial experience report on a primary evaluation of the cost and benefits of using formal specification tools in general, and the $\mathbb{K}$ framework in particular, to formally specify the syntax and semantics of filtering languages. Overall, it seems to the authors that using the $\mathbb{K}$ framework helped greatly to improve GPFL's specification quality. It forced the specification authors to be precise, and helped spot various errors and missing specification fragments.

With regard to the "cost", this experimentation argues in favor of tool supported formal specifications for high quality specifications of filtering languages. Of course, using natural language, it is possible to produce a cheaper, but ambiguous and approximate, specification. However, from the authors' natural language based experiences with packet filtering language specifications, using natural language to produce a specification with a similar level of precision and correctness would be more costly for engineers with operational semantics knowledge. With a decent knowledge of operational semantics concepts, the cost for newcomers to the $\mathbb{K}$ framework is relatively low, thanks to the numerous tutorials (in text and video), manuals and examples. In fact, having been exposed to operational semantics concepts (apart from general computer science concepts) seems to be the only prerequisite to efficiently using the $\mathbb{K}$ framework.

From the authors' previous experiences at formal specification of packet filtering language specifications without tool support, the cost of the constraints imposed by the $\mathbb{K}$ framework seems to the authors to be lower than the benefits provided by the tool support. Typically, the ability to simulate[1] the formal specification of the filtering language requires a particular handling of input/output related rules. However, this same ability to simulate the formal specification of the filtering language is highly beneficial when validating the correctness of the specification and expressivity of the language by "executing" test and documentation programs.

Other benefits of tool supported formal specifications of languages are numerous. In natural language documents specifying new languages, it is too common for program examples to be inconsistent with the language grammar. It is easily explained by the modifications brought to the language grammar during the specification document development. Examples directly related to the modified statements are usually modified accordingly. However, examples related to other aspects of the language are often forgotten. Using a tool supported formal specification, it is easy to adopt a "continuous/frequent integration" approach where examples are: written in separate files, regularly parsed to verify that they comply with the current grammar, and automatically imported in the specification document (the creation of this paper used this approach).

Additionally, use of a tool-supported formal specification approach modifies the workflow often applied when using natural language specification documents. With natural language specifications, the specification document writing process usually starts early after a short engineering phase (it may not be true for a language *development* process, however it is often the case in pure language *specification* processes), and the main part of the language specification is done during the specification document writing process. With a tool-supported formal specification approach, the specification of the language tend to be first developed inside the tool, and then the language specification is clarified during the specification document writing process. With a tool-supported formal specification approach, the language specification becomes a two phases process with two different views on the language specification. The

[1] The authors prefer to talk of "simulation" rather than "execution", as the loading time of the execution environment and limited ability to interact with other components would most likely prevent to use such an execution in a real world setting.

"two different views" aspect is particularly true with the $\mathbb{K}$ framework were semantics rules are entered textually in the source file and can be rendered graphically for the specification document. This two phases workflow (development then clarification and documentation) helps spot: differences of treatments (in particular for configuration cells), generalization and reuse opportunities (for example, in this experimentation, the use of only two internal commands, `iSend` and `iHalt`, to encode the three packet commands `accept`, `drop` and `send`), different concepts that are candidates to modularization (for example, in this experimentation, the externalization of packet data type definitions and string conversions), errors that manifest themselves in rare occasions (for example, in an earlier version of GPFL, automaton states and variable values where stored in the same map, which could trigger a key clash caused by variable and automaton identifiers having the same "name" part), or general simplifications (for example, during this report writing process, GPFL's configuration has been heavily reformatted to simplify the language specification and be closer to the concepts manipulated). From the authors experience, in general and compared to a natural language approach, a tool-supported formal specification process helps simplify and clarify a language specification.

Moreover, the ability to execute the formal specification allows to adopt an incremental approach for the specification of the different statements semantics. In such an approach, the syntax of the language is first specified. Then a program example making use of all the statements of the language in as much context as reasonable is written. The semantics of the statements is then defined statements by statements. The program is executed using $\mathbb{K}$'s run time; and the execution stops when reaching a statement whose semantics is not defined yet. All the semantics rules associated to this statement are then defined. When stopping an execution, $\mathbb{K}$'s run time displays the current state of the configuration which can help specify the missing semantics rules. As the test program execution goes further and further during the language semantics specification process, this incremental approach is more rewarding for people in charge of the specification. The impact of using this incremental approach (which is not required by the $\mathbb{K}$ framework) on the quality of the specifications produced remains to be investigated.

Finally, the ability to execute the formal specification allows to test and validate the language specification. Two important points to validate are: the expressivity of the language and its expected semantics. GPFL's test code (Sect. 5) provided in the companion technical report [18] emphasizes the limitations of the simple automata that can be defined using GPFL. It could be useful to have automaton state variables, and triggering conditions that test and check automaton state variable values. However, adding automaton state variables would complexify automata definitions. Similarly, GPFL's test code contains a recurring code sequence to handle alarms which is triggered only when a threshold of a specific event occurrences is reached. It could be useful to add a specific command to GPFL which would have the same semantics as this recurring sequence. The ability to test programs does not solve expressivity questions (which have to be answered on a per language basis), however it helps explicit those questions. With regard to expected semantics, writing test programs helps validate that programs have the semantics that users would expect. The initial version of GPFL's test code did not behave as expected. It ended up being a misplaced statement in the filter code, but could also have been a problem with the semantics specification. Discovering the cause of a misbehavior of a test program (error in the semantics or the program) could be greatly simplified by $\mathbb{K}$'s debugger which can "execute" formal specifications step by step; especially as Domain Specific Languages (specifications and implementations) usually have limited debugging facilities (which is in accordance with their philosophy of limited expressivity for the sake of simplification). However, sadly, $\mathbb{K}$'s debugger crashed on our program with the version of the $\mathbb{K}$ framework used for this experimentation (version 3.6). This can be explained by the fact that $\mathbb{K}$ development effort was focused on the next version to come (version 4.0 which exited the beta stage at the end of July 2016). Finally, the ability to execute the formal specification helps to validate a set of test programs that

can be used as smoke test for language implementations.

7 Conclusion

This paper reports on an industrial experiment to formally specify the syntax and semantics of a filtering language (GPFL) using the tool-supported framework $\mathbb{K}$. For confidentiality reasons, the filtering language specified in this report has been made up for this experimentation; however, it covers the majority of concepts usually encountered in filtering languages. No comparison between different tools is made in this experiment. The goal of the experiment is to study the feasibility of using a tool-supported formal approach for the specification of domain-specific filtering languages having a complexity similar to filtering languages encountered in real-life projects.

The $\mathbb{K}$ framework proved to be sufficiently expressive to naturally express the syntax and semantics of GPFL in a formal way. The effort required by this formal specification is judged reasonable by the authors, and within reach of average engineers which have been exposed previously to operational semantics theories. Newcomers life is made easier by the numerous manuals, examples and tutorials available for the $\mathbb{K}$ framework. The tool support is a welcome help during the specification process. In particular, the ability to execute (or simulate) $\mathbb{K}$ formal specifications helps greatly when developing and fine tuning the language specification, and when producing smoke tests for the implementation.

Following such a specification process may seem to be in complete contradiction to any agile development principles [4]. However, using a tool-supported *executable* specification methodology allows to comply with one of the pillars of agile development: *early feedback*. As the language specification is executable, it is possible to ask final users (if some are available) to test the language and provide feedbacks on different aspects of the language, including its expressivity. In fact, IBM's Continuous Engineering development methodology [24] advocates for the use of executable models at every steps of the development.

To summarize, with regard to the benefits of putting the effort to produce a *formal* specification, the authors opinion, on improved quality and usefulness of formal specifications compared to non formal specifications written in natural language, is relatively well summarized in the following statement by David Schmidt [22], which is supported by the numerous ambiguities (and their consequences) in natural language specifications of common programming languages like C/C++ or Java [12].

> "Since data structures like symbol tables and storage vectors are explicit, a language's subtleties are stated clearly and its flaws are exposed as awkward codings in the semantics. This helps a designer tune the language's definition and write a better language manual. With a semantics definition in hand, a compiler writer can produce a correct implementation of the language; similarly, a user can study the semantics definition instead of writing random test programs."
>
> David Schmidt in ACM Computing Surveys [22]

In the experimentation reported in this paper, no formal analysis of the formal specification produced has been attempted. In future work, the authors plan to try some of the experimental tools available with the $\mathbb{K}$ framework on GPFL's specification. If time allows, a similar experimentation could be repeated with other tools oriented toward the formal specification of languages.

References

[1] John W. Backus (1959): *The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference*. In: *Proc. Int. Conf. Information Processing*, UNESCO, pp. 125–132.

[2] H. J. S. Basten, J. van den Bos, M. A. Hills, P. Klint, A. W. Lankamp, B. Lisser, A. J. van der Ploeg, T. van der Storm & J. J. Vinju (2015): *Modular Language Implementation in Rascal – Experience Report*. *Science of Computer Programming* 114, pp. 7–19, doi:10.1016/j.scico.2015.11.003.

[3] Fabricio Chalub & Christiano Braga (2007): *Maude MSOS Tool*. In: *Proc. Int. Work. Rewriting Logic and its Applications*, *Electronic Notes in Theoretical Computer Science* 176, Elsevier Science Publishers B. V., pp. 133–146, doi:10.1016/j.entcs.2007.06.012.

[4] Alistair Cockburn (2007): *Agile Software Development: The Cooperative Game*, 2nd edition. Pearson Education.

[5] Olivier Dubuisson (2000): *ASN.1 – Communication between Heterogeneous Systems*. OSS Nokalva. Available at `http://www.oss.com/asn1/dubuisson.html`. Translated from French by Philippe Fouquart.

[6] Matthias Felleisen, Robert Bruce Findler & Matthew Flatt (2009): *Semantics Engineering with PLT Redex*. The MIT Press.

[7] Matthew Flatt & PLT (2010): *Reference: Racket*. PLT-TR 2010-1, PLT Design Inc. `https://racket-lang.org/tr1/`.

[8] Jean van Heijenoort (2002): *From Frege to Gödel: A Source Book in Mathematical Logic, 1879-1931*, chapter Peano (1889). The principles of arithmetic, presented by a new method. Source Books in the History of the Sciences, Harvard University Press. A translation and excerpt of Peano's 1889 paper "Arithmetices principia, nova methodo exposita".

[9] Jean-Marc Jézéquel, Olivier Barais & Franck Fleurey (2011): *Summer School on Generative and Transformational Techniques in Software Engineering*, chapter Model Driven Language Engineering with Kermeta, pp. 201–221. *Lecture Notes in Computer Science* 6491, Springer Berlin Heidelberg, doi:10.1007/978-3-642-18023-1_5.

[10] Jean-Marc Jézéquel, Benoit Combemale, Olivier Barais, Martin Monperrus & François Fouquet (2013): *Mashup of metalanguages and its implementation in the Kermeta language workbench*. *Software & Systems Modeling* 14(2), pp. 905–920, doi:10.1007/s10270-013-0354-4.

[11] Joint Technical Committee ISO/IEC JTC 1, Information technology, Subcommittee SC 6, Telecommunications and information exchange between systems (2015): *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*. International Standard 8824-1, ISO/IEC. ISO/IEC version of ITU-T X.680 (08/2015).

[12] JSR-133 expert group (2004): *JSR-133 Java™ Memory Model and Thread Specification Revision*. Java Specification Request (JSR) 133, Sun Microsystems, Inc. `https://jcp.org/en/jsr/detail?id=133`.

[13] Casey Klein, John Clements, Christos Dimoulas, Carl Eastlund, Matthias Felleisen, Matthew Flatt, Jay A. McCarthy, Jon Rafkind, Sam Tobin-Hochstadt & Robert Bruce Findler (2012): *Run Your Research: On the Effectiveness of Lightweight Mechanization*. In: *Proc. Symp. Principles of Programming Languages*, *SIGPLAN Not.* 47, ACM, New York, NY, USA, pp. 285–296, doi:10.1145/2103656.2103691.

[14] P. Klint (2009): *Tribute to a great Meta-Technologist: from Centaur to The Meta-Environment*. In Y. Bertot, G. Huet, J.-J. Levy & G. Plotkin, editors: *From Semantics to Computer Science, Essays in Honour of Gilles Kahn*, Cambridge University Press, pp. 235–264, doi:10.1017/CBO9780511770524.012.

[15] P. Klint, J. J. Vinju & M. A. Hills (2011): *RLSRunner: Linking Rascal with K for Program Analysis*. In: *Proc. Int. Conf. Software Language Engineering*, Springer, doi:10.1007/978-3-642-28830-2_19.

[16] Paul Klint, Tijs van der Storm & Jurgen Vinju (2009): *RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation*. In: *Proc. Int. Working Conf. Source Code Analysis and Manipulation*, IEEE Computer Society, pp. 168–177, doi:10.1109/SCAM.2009.28.

[17] Donald E. Knuth (1964): *Backus Normal Form vs. Backus Naur Form*. Commun. ACM 7(12), pp. 735–736, doi:10.1145/355588.365140.

[18] Gurvan Le Guernic & José A. Galindo (2016): *Experience Report on the Formal Specification of a Packet Filtering Language Using the K Framework*. Research report 8967, Inria. `https://hal.inria.fr/hal-01385541v1`.

[19] Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge & Peter Sewell (2014): *Lem: Reusable Engineering of Real-world Semantics*. In: Proc. Int. Conf. Functional Programming, SIGPLAN Not. 49, ACM, pp. 175–188, doi:10.1145/2692915.2628143.

[20] Grigore Roşu & Traian Florin Şerbănuţă (2010): *An Overview of the $\mathbb{K}$ Semantic Framework*. The Journal of Logic and Algebraic Programming 79(6), pp. 397–434, doi:10.1016/j.jlap.2010.03.012.

[21] Grigore Roşu & Traian Florin Şerbănuţă (2014): *$\mathbb{K}$ Overview and SIMPLE Case Study*. In: Proc. Int. Work. K Framework and its Applications (K 2011), Electronic Notes in Theoretical Computer Science 304, pp. 3–56, doi:10.1016/j.entcs.2014.05.002.

[22] David A. Schmidt (1996): *Programming Language Semantics*. ACM Computing Surveys 28(1), doi:10.1145/234313.234419.

[23] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar & Rok Strniša (2010): *Ott: Effective Tool Support for the Working Semanticist*. J. Functional Programming 20(1), pp. 71–122, doi:10.1017/S0956796809990293.

[24] Cathleen Shamieh (2014): *Continuous Engineering For Dummies®*. IBM Limited Edition, John Wiley & Sons, Inc.

[25] T. van der Storm & J. J. Vinju (2008): *Using the Meta-Environment for Domain Specific Language Engineering*. Technical Report SEN-R0805, CWI Software Engineering.

[26] Traian Florin Şerbănuţă, Andrei Arusoaie, David Lazar, Chucky Ellison, Dorel Lucanu & Grigore Roşu (2014): *The $\mathbb{K}$ Primer (version 3.3)*. Electronic Notes in Theoretical Computer Science 304, pp. 57–80, doi:10.1016/j.entcs.2014.05.003. Proc. Int. Work. K Framework and its Applications (K 2011).

Extending a user interface prototyping tool with automatic MISRA C code generation

Gioacchino Mauro[1], Harold Thimbleby[2], Andrea Domenici[1], and Cinzia Bernardeschi[1]

[1] Department of Information Engineering, University of Pisa, Pisa, Italy
{g.mauro,c.bernardeschi,a.domenici}@unipi.it
[2] Swansea University — Prifysgol Abertawe, Swansea/Abertawe, UK
harold@thimbleby.net

Abstract. We are concerned with systems, particularly safety-critical systems, that involve interaction between users and devices, such as the user interface of medical devices. We therefore developed a MISRA C code generator for formal models expressed in the PVSio-web prototyping toolkit. PVSio-web allows developers to rapidly generate realistic interactive prototypes for verifying usability and safety requirements in human-machine interfaces. The visual appearance of the prototypes is based on a picture of a physical device, and the behaviour of the prototype is defined by an executable formal model. Our approach transforms the PVSio-web prototyping tool into a model-based engineering toolkit that, starting from a formally verified user interface design model, will produce MISRA C code that can be compiled and linked into a final product. An initial validation of our tool is presented for the data entry system of an actual medical device.

Keywords: User interface prototype, formal methods, code generation, MISRA C

1 Introduction

Formal methods are important for developing and understanding safe and secure systems. The PVSio-web framework [20,21,22,26] allows developers to use formal methods in a friendly and appealing way as it provides realistic animations and is integrated with a graphical editor for the Emucharts language [23]. (Emucharts is a state machine formalism with guards and actions associated with transitions; it is explained further in Sect. 3.4 below.)

PVSio-web uses the formal modelling language of the Prototype Verification System (PVS) [27], including the PVSio extension [24]. PVS is an industrial-strength theorem proving system that allows formal verification of safety and reliability properties of hardware and software systems [5,31]. Although PVS itself is very effective, it is not widely used for model-based development and analysis of user interfaces, as the tool has a steep learning curve. PVSio-web softens this learning curve, making the tool more user-friendly and accessible, providing developers with a graphical modelling environment, and a toolbox for developing realistic visual prototypes of user interfaces.

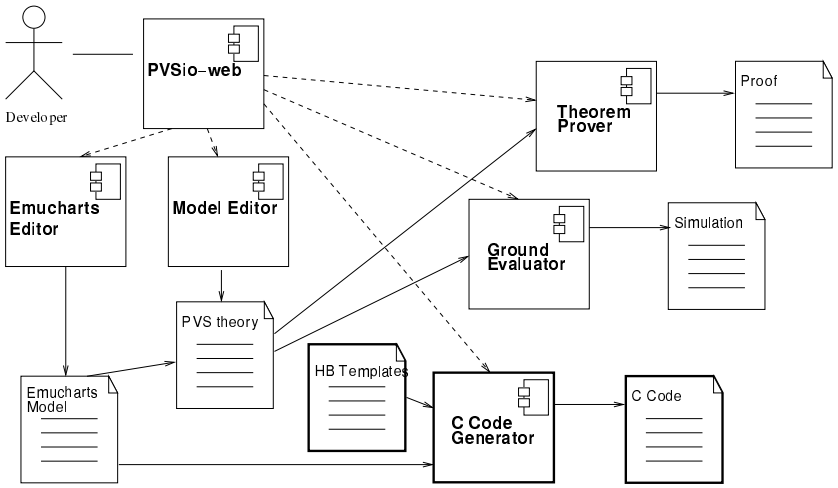


Fig. 1. C code generation in the PVSio-web development process.

The applications of embedded software in safety-critical applications increase continuously. Taking this into account, together with the requirements to reduce time and overall production costs, automatic code generation plays an essential role. Automatic code generation guarantees a smooth conversion from model to code and reduces the debugging and testing required for source code, provided that the correctness of code generation and of the high-level model have been verified.

We therefore present an extension to PVSio-web that generates C code. Specifically, our extension generates MISRA C [1], a safety-oriented subset of C developed by the Motor Industry Software Reliability Association (MISRA). MISRA C is commonly used in safety-critical subsystems, such as car braking in automotive systems.

With this new extension, formal PVS specifications generated from Emucharts diagrams are automatically converted to C, significantly shortening project development time. Because of the approach, the semantics of generated C code is equivalent to the formal models, and therefore the code retains the reliability and safety properties formally verified for the PVS model.

In summary, our main contribution is an approach to software development that integrates logic- and state machine-based formal modelling, validation by simulation, and automatic implementation by generating production code to be run on the actual system hardware, all based on an industrial-strength formal methods toolkit.

2 Related work

Model-based approaches are commonly used in the field of human computer interaction, for example [13]. Most approaches are focused on describing user interfaces and their implementations at various levels of abstraction. Developers of user interfaces for interactive systems also have to address heterogeneity and adaptation to the context of use. For example, in [29], a model-based declarative language for the design of interactive applications based on Web services in ubiquitous environments was presented. In

contrast to these familiar approaches, the present work proposes a framework enabling a formal verification of user interaction. The framework is meant for safety analysis of safety-critical devices and not with user interface design issues as discussed in [29],

Similarly to our approach, formal models were used in [6] to describe functionality and component interactions, where they were combined with user interface models in order to get the entire model of the system. Moreover, an Android emulator application was generated, using Java and XML technologies. Presentation models and presentation interaction models were used in [9] to model interactive software systems; these models were shown to be usable with a formal specification of the system functionality. In [7] the same formalisms were used to model user manuals of modal medical devices, proving that the user manual may be not always consistent with actual device behaviour.

In [17], model checking was used to model and prove properties of specifications of interactive systems so that possibly unexpected consequences of interface mode changes can be checked early in the design process. In [19], the complementary role of model checking and theorem proving in the analysis of interactive devices was considered. Recent work [16] explored the paths that a user will take in interacting with medical devices for the analysis of properties of the behaviour of safety-critical devices. A model-checking approach has also been used to analyse hardware behaviour [4].

A discussion of production code generation in model-based development can be found in [11]. Many papers deal with specific code generators, for example TargetLink [3]. Code generators specifically designed for medical systems are described in [2] and [28].

3 PVS, model-driven development and Emucharts

This section provides background information on the PVSio-web framework and its relationship to model-driven development.

3.1 PVS, the Prototype Verification System

The PVS is an interactive theorem prover for a typed higher-order logic language, providing an extensive set of inference rules based on the sequent calculus [30]. Its PVSio extension is a ground evaluator that can compute the results of ground function applications, that is PVS expressions consisting of a function name applied to variable-free arguments. PVS functions are purely declarative definitions of mathematical mappings, without any procedural information on how to compute them, but the PVSio package can derive and execute an algorithm to evaluate a ground function application, turning it into a procedure call. The PVSio package also provides functions with side effects, such as input and output, which do not interfere with the semantics of a theory.

A system is modelled in PVS as a *theory*, a collection of logical statements and definitions about the structural and behavioural aspects of the system. The system's required properties are expressed as theorems to be verified with the PVS theorem prover. If the behavioural aspects are expressed as functions, the system can also be simulated with the PVSio extension. The same logical model can then be used both for verification and simulation.

3.2 Model-driven development

Model-driven development (MDD) is based on creating an executable system model by assembling functional blocks. An executable model makes it possible both to simulate the system and to generate production software to control it. Together with the naturalness of the graphic language of functional blocks, these features make MDD very attractive to developers. However, this approach has two limits: first, functional blocks lend themselves to building design models, but not specification ones; and secondly, formal verification of a block-based model is tedious, and in fact it is uncommon in industrial practice.

A formal approach can be used to create both specification and design models and intrinsically lends itself to rigorous verification of system properties. In particular, logic specification languages, such as PVS, are supported by automatic or interactive theorem provers used by developers to check if system requirements, expressed as logical formulas, are implied by a system's description expressed in a logic theory. However, formal methods require expertise in languages and methods that are not widely known in the wider developer community. Further, most formal languages abstract from the familiar procedure-oriented computation model of popular programming languages, making it harder to generate executable software.

It is then desirable to have tools and methods providing developers with the features of both approaches. The present work is part of a research effort aimed at this goal. With the PVSio-web framework, a developer can build a model in a graphical state-machine language or a logic language, or both (Sect. 3). The graphical model is translated into the logic language automatically, and the resulting translation is both verifiable and executable using the PVSio ground evaluator, which acts as an interpreter for the PVS language. The PVSio-web framework thus provides features of the formal approach: A formal specification language and a verification tool, and features of MDD, thus providing a full graphical modelling language and a simulation engine. A translator from Emucharts to C makes it possible to generate code from a state machine-based model that can be validated by simulation and verified by theorem proving. The other important feature of MDD — generation of production code capable to be run on the actual system hardware — is a key contribution of this paper.

3.3 PVSio-web

The PVSio-web framework is a set of tools, co-ordinated by a web-based interface, for prototyping and simulation of interactive devices. Its main components are, besides PVS with its PVSio extension: (*i*) the **Prototype Builder**, a graphical tool used to choose a picture of an existing or anticipated device's front panel and to associate PVS functions with active areas of the picture representing device inputs (e.g., buttons or keys) and outputs (e.g., alphanumeric displays or lights); (*ii*) the **Model Editor**, a textual interface to write PVS code; (*iii*) the **Emucharts Editor**, a graphical tool to draw Emucharts state machine diagrams; (*iv*) a **Simulation Environment**; and (*v*) **Code Generators** for PVS and other formal languages (currently Presentation Interaction Models [8], Modal Action Logic [15], and Vienna Development Method [12]) — and for MISRA C, as presented in this paper.

PVSio-web can be used to prototype a new device interface, or to create a reverse-engineered model of an existing one. In either case, a developer creates formal descriptions of the device's responses to user actions, using the model and Emucharts editors, and associates these descriptions with the active areas of the simulated interface, using the prototype builder. In the simulation environment, the developer, or a domain expert or a potential user, interacts with the prototype clicking on the input widgets. These actions are translated to PVS function calls executed by the PVSio interpreter.

3.4 Emucharts

An Emucharts diagram is the representation of an extended state machine in the form of a directed graph composed of labelled *nodes* and *transitions*. Transitions are labelled with triples of the form $trigger[guard]\{action\}$, where *trigger* is the name of an event, *guard* is an enabling Boolean expression, and *action* is a set of assignments to typed variables declared in the state machine's *context*. The default guard is the *true* value and the default action is a no-operation. The *state* of the machine is defined by the current node and the current values of the context variables.

The code generator for PVS produces a theory containing functions that define the state machine behaviour on the occurrence of trigger events. Since an Emucharts diagram usually represents a device response to user actions, such events represent user actions, such as pressing a button on a control panel. During simulation on a PC, a user click on an active area of the device picture causes the simulator to generate a function application expression that is passed to the PVSio ground evaluator.

4 From Emucharts to safe C

The aim of programming code generation in the PVSio-web framework is producing a module that implements the user interface of a device which can be compiled and linked into the device software without any particular assumptions on its architecture. In this way, the user interface module can be used without forcing design choices on the rest of the software. In our approach, the generated module contains a set of C functions. The main ones are, for each Emucharts trigger: (*i*) a *permission* function, to check if the trigger event is *permitted*, i.e., whether it is associated with any transition from the current state, and (*ii*) a *transition* function that, according to the current state, updates it, provided that the guard condition of an outgoing transition holds. The code includes logically redundant tests (*assert* macros) to improve robustness.

To generate production-quality code fit for safety-critical applications we adopt MISRA guidelines. The MISRA guidelines for the C language, originally conceived for the automotive industry, enforce programming practices to improve maintainability and portability and, above all, to reduce the risk of malfunction due to implementation- or platform-dependent aspects of the C language. For instance, there are rules that bar the use of constructs such as `goto`, and rules requiring that numeric literals be suffixed to indicate their type explicitly. The generated code currently complies with the first version of the 1998 MISRA C guidelines.

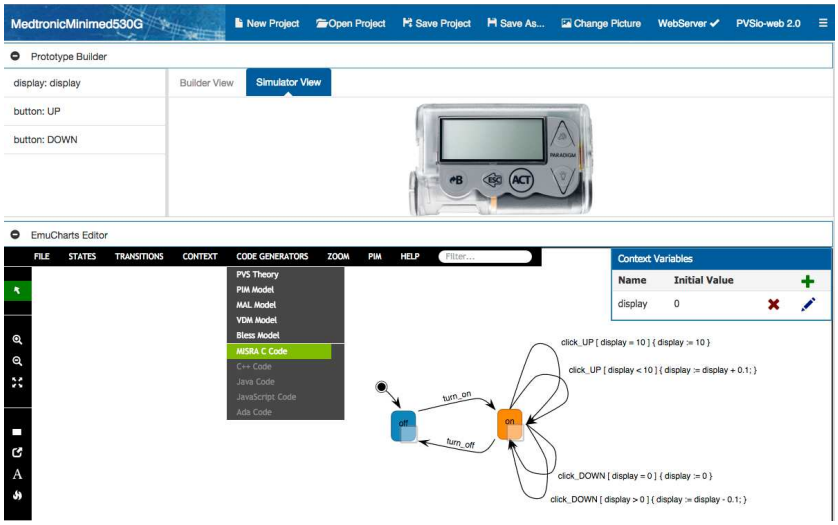


Fig. 2. The PVSio-web user interface with the Prototype Builder and Emucharts Editor frames.

4.1 Code generation

Our MISRA C code generator was implemented in JavaScript using Handlebars [14], a macro-expansion tool for web applications. A Handlebars template is a piece of text containing "Handlebar expressions," which refer to elements of the surrounding context, typically an HTML document. A Handlebars expression specifies a character string as a function of context elements, which is compiled into a JavaScript function that returns the template text with the substitutions computed by the Handlebars expressions. For example, a template fragment for a C preprocessor `#include` directive is `#include "{{filename}}.h"`, where the Handlebars expression `{{filename}}` contains the `filename` parameter that will be replaced by the actual name of the file to be included.

The code generator produces a header file, an implementation file, a makefile, a simple test driver file, and a documentation manual.

The structure of the header file is defined by the grammar in Table 1. The header file contains, among other items, the declarations (*typedef_definitions* in the grammar) for types with explicit representation of size and sign, e.g., `UC_8`, for *eight-bit unsigned char*, the declaration (*state_labels_enum*) for an enumeration type defining the node labels, and the declaration (*state_structure*) for the *state* structure type representing the state of the Emucharts model. This structure contains one *context* field for each variable defined in the Emucharts context, and two more fields (*curr_node* and *prev_node*) contain the labels of the current and the previous node.

The declarations are followed by the function prototypes of the two utility functions *enter* and *leave*, the *init* function, and, for each trigger, one permission and one transition function. The functions receive a pointer to a structure of type *state* passed by a calling program. The *enter* and *leave* functions, called by the *init* and transition functions, update the *curr_node* and *prev_node* fields, respectively, with the target and

⟨ *headerfile* ⟩ ::= ⟨ *preprocessor_directives* ⟩
[⟨ *constant_definitions* ⟩]
⟨ *typedef_definitions* ⟩
⟨ *state_labels_enum* ⟩
⟨ *state_structure* ⟩
⟨ *utility_functions* ⟩
⟨ *init_function* ⟩
⟨ *permission_functions* ⟩
⟨ *transition_functions* ⟩

Table 1. Structure of a header file. Non-terminal symbols are enclosed between angle brackets and square brackets enclose optional symbols.

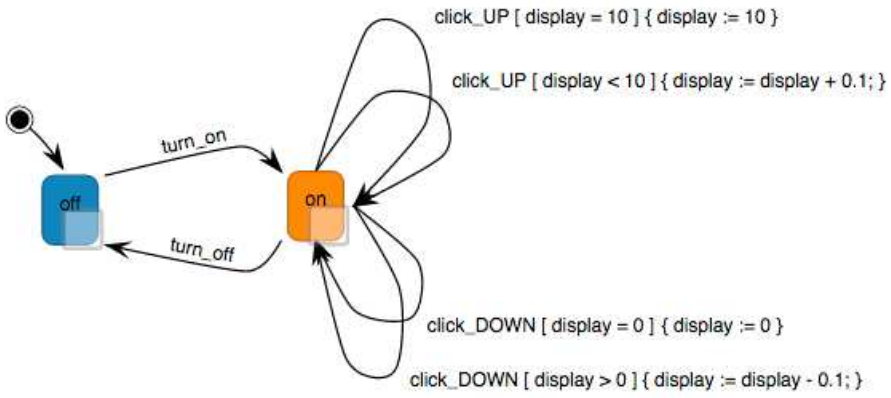


Fig. 3. Emucharts diagram for the Medtronic MiniMed 530G data entry system.

source node label of the executed transition. The *leave* function has been introduced to allow future versions to implement checkpointing algorithms. The *init* function initialises the state's context fields with the values of the context variables specified in the Emucharts diagram, and the *curr_node* field with the label of the initial node. As mentioned above, each permission function checks if the current node has a transition labelled by the respective event. Then, the matching transition function chooses among the transitions triggered by that event, according to the respective guards (assumed to be mutually exclusive).

The implementation file contains the function definitions. For example, consider the Emucharts diagram of the data entry system of the Medtronic MiniMed 530G System shown in Figure 3. The diagram has a context variable *display* of type *double* represented on 64 bits, which holds the value shown on the device's display. The node labels and the *state* type are defined as

```
typedef enum { off, on } node_label;
typedef struct {
    D_64 display;
    node_label curr_node;
```

```
    node_label prev_node; } state;
```

The code for the permission function associated with the *click_UP* trigger is

```
UC_8 per_click_UP(const state* st) {
    if (st->current_state ==  on) {
        return true;
    }
    return false;
}
```

where the return type UC_8 (eight-bit unsigned character) is used to represent the Boolean type. The transition function is

```
state click_UP(state* st) {
    assert(st->current_state ==  on);
    assert(st->display < 10 || st->display == 10);
    if (st->display < 10 && st->current_state == on) {
        leave(on, st);
        st->display = st->display + 0.1f;
        enter(on, st);
        assert(st->current_state == on);
        return *st;
    }
    if (st->display == 10 && st->current_state == on) {
        leave(on, st);
        st->display = 10.0f;
        enter(on, st);
        assert(st->current_state == on);
        return *st;
    }
    return *st;
}
```

A proof of the correctness of this translation schema is shown in Appendix A.

5 Case study

The Alaris GP, made by Becton Dickinson and Company, was used as a case study for the MISRA C code generator.

This volumetric infusion pump is a medical device used for controlled automatic delivery of fluid medication or blood transfusion to patients, with an infusion rate range between 1 ml/h and 1200 ml/h. It has a monochrome dot matrix display with three significant digits, and has 14 buttons for operating the device (see Figure 4). The pump has a rather complex user interface, with different modes of operation and ways of entering data, including the possibility of choosing from a list of preloaded treatments. For simplicity, in this paper only the essential part of the data entry interface, concerning numerical input and display, is considered.

Fig. 4. Front panel of the Alaris GP infusion pump.

Numerical input is done through the chevrons buttons: upward and downward chevrons increase and decrease, respectively, the displayed value. The amount by which the value is increased or decreased depends on whether a single or double chevron is pressed, and on the current displayed value. More precisely, the displayed value is changed as follows: (*i*) If the displayed value is below 100, the value changes by 0.1 units for a single chevron, and steps up or down to the next decade for a double chevron (e.g., from 9.1 to 10.0); (*ii*) if the displayed value is between 100 and 1,000, the value changes by 1 unit for a single chevron, and steps up or down to a value equal to the next hundred plus the decade of the displayed value for a double chevron (e.g., from 310 or 315 to 410); (*iii*) if the displayed value is 1,000 or above, the value changes by 10 units for a single chevron, and steps up or down to a value equal to the next hundred for a double chevron (e.g., from 1,010 or 1,080 to 1,100).

The Emucharts diagram for the numeric data entry is shown in Fig. 5. Triggers *click_alaris_up* and *click_alaris_dn* represent clicks on the upward and downward single-chevron buttons, respectively, and triggers *click_alaris_UP* and *click_alaris_DN* represent clicks on the double-chevron ones. For each event, combinations of guards and actions specify the rules described above.

The PVS code generator translates the diagram into an executable logic theory, and the C code generator produces permission and transition functions for each trigger, as explained previously.

5.1 Mobile applications

The PVSio-web framework uses a standard web interface to integrate its tools: this approach offers a uniform interface that a developer can access with any web browser.

Our framework has been extended by providing the possibility to run simulations on a mobile device. Smartphones and tablets improve usability and help make user

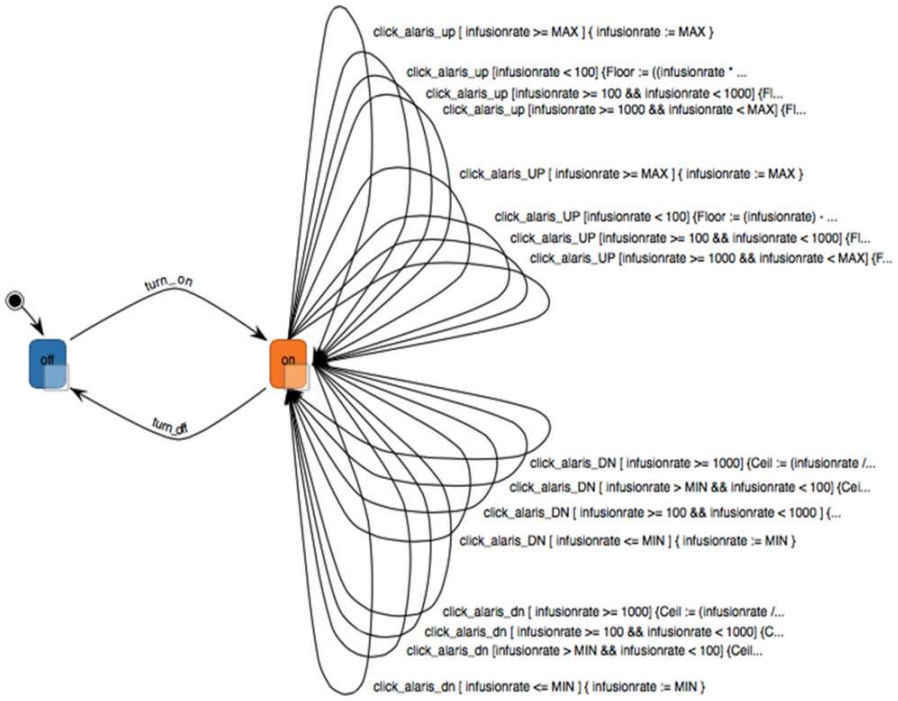


Fig. 5. Emucharts diagram for numeric data entry.

interaction similar to actual device operation. For example, mobile devices could be used in a hospital environment to train medical personnel and patients.

An interactive device can be simulated using the C source code produced by the PVSio-web generator, compiled and linked with a mobile device-specific application. For example, the code for the user interface of the Alaris infusion pump has been ported to the Android [10] platform using the Android NDK [25] toolset, which can embed C code in a Java project, relying on the Java Native Interface (JNI) [18].

6 Conclusions

We presented the implementation of our MISRA C code generator for the PVSio-web prototyping toolkit. Automatic code generation significantly reduces project development time. Our approach eliminates a human-performed step in the development process: user interface software engineers no longer need to convert the design specifications into executable target code.

Our tool improves the development of safe and dependable user interfaces, as it greatly facilitates using formal methods easily and reliably with real UIs, which we demonstrated with the medical device examples in this paper.

Current and future directions include improving this initial integration with other features of C, still conformant to MISRA C under the most recent 2012 rules. We plan to develop code generators for programming languages such as C++, Java and ADA.

Acknowledgements

This work was partially supported by the PRA 2016 project "Analysis of Sensory Data: from Traditional Sensors to Social Sensors" funded by the University of Pisa.

References

1. Motor Industry Software Reliability Association (1998): *Guidelines for the Use of the C Language in Vehicle Based Software.* Motor Industry Research Association.
2. Ayan Banerjee & Sandeep K. S. Gupta (2014): *Model Based Code Generation for Medical Cyber Physical Systems.* In: *1st Workshop on Mobile Medical Applications (MMA '14)*, pp. 22–27, doi:10.1145/2676431.2676646.
3. M. Beine, R. Otterbach & M. Jungmann (2004): *Development of safety-critical software using automatic code generation.* Technical Report, SAE Technical Papers, doi:10.4271/2004-01-0708.
4. C. Bernardeschi, L. Cassano, A. Domenici & L. Sterpone (2013): *Unexcitability Analysis of SEUs Affecting the Routing Structure of SRAM-based FPGAs.* In: *Proc. of the 23rd ACM Great Lakes Symposium on VLSI*, GLSVLSI '13, pp. 7–12, doi:10.1145/2483028.2483050.
5. Cinzia Bernardeschi, Paolo Masci & Holger Pfeifer (2008): *Early Prototyping of Wireless Sensor Network Algorithms in PVS*, pp. 346–359. Springer Berlin Heidelberg, Berlin, Heidelberg, doi:10.1007/978-3-540-87698-4_29.
6. J. Bowen & A. Hinze (2011): *Supporting Mobile Application Development with Model-Driven Emulation.* In: *Formal Methods for Interactive Systems 2011*, *Electr. Comm. EASST* 45.
7. J. Bowen & S. Reeves (2012): *Modelling User Manuals of Modal Medical Devices and Learning from the Experience.* In: *4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '12)*, pp. 121–130, doi:10.1145/2305484.2305505.
8. J. Bowen & S. Reeves (2015): *Design Patterns for Models of Interactive Systems.* In: *24th Australasian Software Engineering Conference (ASWEC)*, IEEE, pp. 223–232, doi:10.1109/ASWEC.2015.30.
9. A. Cerone, P. Curzon, J. Bowen & S. Reeves (2007): *Formal Models for Informal GUI Designs.* *Electronic Notes in Theoretical Computer Science* 183, pp. 57–72, doi:10.1016/j.entcs.2007.01.061.
10. Guiran Chang, Chunguang Tan, Guanhua Li & Chuan Zhu (2010): *Developing Mobile Applications on the Android Platform.* In: *Mobile Multimedia Processing*, pp. 264–286, doi:10.1007/978-3-642-12349-8_15.
11. T. Erkkinen & M. Conrad (2007): *Safety-critical software development using automatic production code generation.* Technical Report, SAE Technical Papers, doi:10.4271/2007-01-1493.
12. J. Fitzgerald, P. G. Larsen, P. Mukherjee, N. Plat & M. Verhoef (2005): *Validated Designs For Object-oriented Systems.* Springer-Verlag TELOS, Santa Clara, CA, USA.
13. J. D. Foley & P. Noi Sukaviriya (1994): *History, Results, and Bibliography of the User Interface Design Environment (UIDE), an Early Model-based System for User Interface Design and Implementation.* In: *Proceedings of Design, Verification and Specification of Interactive Systems (DSVIS'94)*, pp. 3–14.
14. (2016): *Handlebars Semantic Template.* Available at http://handlebarsjs.com.
15. M. D. Harrison, J. C. Campos & P. Masci (2015): *Reusing models and properties in the analysis of similar interactive devices.* *Innovations in Systems and Software Engineering* 11(2), pp. 95–111, doi:10.1007/s11334-013-0201-3.

16. MD. Harrison, JC. Campos, R. Rimvydas & P. Curzon (2016): *Modelling information resources and their salience in medical device design.* In: *8th ACM SIGCHI Symposium on Engineering Interactive Computing Systems (EICS '16)*, doi:10.1145/2933242.2933250.

17. Campos JC & Harrison MD (2001): *Model checking interactor specifications.* *Automated Software Engineering* 8(3–4), pp. 5275–310, doi:10.1023/A:1011265604021.

18. (2016): *Java Native Interface.* http://docs.oracle.com/javase/8/docs/technotes/guides/jni/.

19. P. Masci, A. Ayoud, P. Curzon, MD. Harrison, I. Lee & H. Thimbleby (2013): *Verification of interactive software for medical devices: PCA infusion pumps and FDA regulation as an example.* In: *5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems, (EICS '13)*, doi:10.1145/2494603.2480302.

20. P. Masci, P. Mallozzi, F. L. De Angelis, G. Di Marzo Serugendo & P. Curzon (2015): *Using PVSio-web and SAPERE for rapid prototyping of user interfaces in Integrated Clinical Environments.* In: *Verisure2015, Workshop on Verification and Assurance, co-located with CAV2015.*

21. P. Masci, P. Oladimeji, P. Curzon & H. Thimbleby (2014): *Tool demo: Using PVSio-web to demonstrate software issues in medical user interfaces.* In: *4th International Symposium on Foundations of Healthcare Information Engineering and Systems (FHIES2014).*

22. P. Masci, P. Oladimeji, P. Curzon & H. Thimbleby (2015): *PVSio-web 2.0: Joining PVS to Human-Computer Interaction.* In: *27th International Conference on Computer Aided Verification (CAV2015)*, Springer, doi:10.1007/978-3-319-21690-4_30. Tool and application examples available at http://www.pvsioweb.org.

23. P. Masci, Yi Zhang, P. Jones, P. Oladimeji, E. D'Urso, C. Bernardeschi, P. Curzon & H. Thimbleby (2014): *Combining PVSio with Stateflow.* In: *6th NASA Formal Methods Symposium (NFM2014)*, doi:10.1007/978-3-319-06200-6_16.

24. C. Muñoz (2003): *Rapid prototyping in PVS.* Technical Report NIA 2003-03, NASA/CR-2003-212418, National Institute of Aerospace, Hampton, VA, USA.

25. (2016): *NDK.* Available at http://developer.android.com/ndk.

26. P. Oladimeji, P. Masci, P. Curzon & H. Thimbleby (2013): *PVSio-web: a tool for rapid prototyping device user interfaces in PVS.* In: *FMIS2013, 5th International Workshop on Formal Methods for Interactive Systems.*

27. S. Owre, J. M. Rushby & N. Shankar (1992): *PVS: A Prototype Verification System.* In: *Automated Deduction—CADE-11: 11th International Conference on Automated Deduction*, pp. 748–752, doi:10.1007/3-540-55602-8_217.

28. M. Pajic, Zhihao Jiang, Insup Lee, O. Sokolsky & R. Mangharam (2014): *Safety-critical Medical Device Development Using the UPP2SF Model Translation Tool.* *ACM Trans. Embed. Comput. Syst.* 13(4s), pp. 127:1–127:26, doi:10.1145/2584651.

29. F. Paternò, C. Santoro & L. D. Spano (2009): *MARIA: A Universal, Declarative, Multiple Abstraction-level Language for Service-oriented Applications in Ubiquitous Environments.* *ACM Trans. Comput.-Hum. Interact.* 16(4), pp. 19:1–19:30, doi:10.1145/1614390.1614394.

30. Raymond Merrill Smullyan (1995): *First-order logic.* Dover publications, New York.

31. Mandayam Srivas, Harald Rueß & David Cyrluk (1997): *Hardware Verification Using PVS.* In Thomas Kropf, editor: *Formal Hardware Verification: Methods and Systems in Comparison*, *Lecture Notes in Computer Science* 1287, Springer-Verlag, pp. 156–205.

$$\textbf{arc}\ \frac{\varepsilon,(p,q,e,g,v');\ \varepsilon = e \wedge n = p \wedge v \models g}{\langle n,v\rangle \rightarrow \langle q,v'\rangle}$$

$$\textbf{idle}\ \frac{\varepsilon,(p,q,e,g,v');\ \varepsilon \neq e \vee n \neq p \vee v \not\models g}{\langle n,v\rangle \rightarrow \langle n,v\rangle}$$

Fig. 6. Emucharts operational semantics.

Appendix A Correctness of code generation

In order to assess the correctness of the generated code, the Emucharts diagram is taken as the reference model, and a correspondence is established between the evolution of the model and that of the executed code.

A.1 Transition system for an Emucharts diagram

As discussed above (section 4), an Emucharts diagram is a graph of nodes and labelled transitions, extended with a set of typed context variables, each one with an initial value. Its semantics is given by a transition system. Let the following be defined:

- A set $N = \{n_1, \ldots, n_i\}$ of nodes;
- a set $X = \{x_1, \ldots, x_j\}$ of context variables (for simplicity, assumed to be typeless);
- a set $\mathbb{V}$ of values;
- a set $E = \{\varepsilon_1, \ldots, \varepsilon_k\}$ of events;
- a set $G = \{g_1, \ldots, g_l\}$ of guards, i.e., Boolean expressions involving variables, constants from $\mathbb{V}$, arithmetic and relational operators;
- a denumerable set V of *valuations*, i.e., functions from X to $\mathbb{V}$;
- a set $A = \{a_1, \ldots, a_l\}$ of arcs, i.e., 5-tuples of the form (s,t,e,g,v), where $s,t \in N$ are the arc's source and target node, $e \in E$, $g \in G$, and $v \in V$ is the valuation defined by the action labelling the corresponding transition in the diagram; more precisely, v is the valuation obtained by overriding the previous valuation with the assignments in the action associated with the arc;
- a set $\mathbb{Q}$ of states of the form $\langle n, v\rangle$, with $n \in N$ and $v \in V$;
- a transition relation $\rightarrow \subseteq Q \times Q$, defined by the semantic rules in Figure 6, where the premises contain an event ε, an arc label, and a logical condition, and the consequences contain a member of the transition relation that is enabled if the condition holds.

With the above definitions, the associated transition system T is the tuple $(\mathbb{Q}, \rightarrow, q_0)$, where $q_0 = \langle n_0, v_0\rangle$ is the initial state. Since the diagram is deterministic, given a sequence of event occurrences $e_1, \ldots, e_k, \ldots$, the transition system has only one sequential path. If an event cannot affect a state (either it is not permitted or no guard prefixed by the event is satisfied), the system does not change state. The operational semantics are given in Figure 6.

$$\mathbf{arc}_P \quad \frac{\varepsilon, (p,q,e,g,v'_c);\ \varepsilon = e \wedge v_n(x_{\text{curr}}) = p \wedge v_c \models g}{\langle v_n, v_c\rangle \xrightarrow{P} \langle q, v'\rangle}$$

$$\mathbf{idle}_P \quad \frac{\varepsilon, (p,q,e,g,v'_c);\ \varepsilon \neq e \vee v_n(x_{\text{curr}}) \neq p \vee v_c \not\models g}{\langle v_n, v_c\rangle \xrightarrow{P} \langle n, v_c\rangle}$$

Fig. 7. Generated code operational semantics.

A.2 Transition system for the generated code

The generated functions are used within a more complex system, which is responsible for catching events at the real or simulated user interface and for calling the respective functions according to an appropriate protocol: the *init* function must have been called previously, then, when an event is catched, the permission function of the corresponding trigger is called, and only if it returns *true* can the respective transition function be executed.

Assume that the data entry subsystem of the device is controlled by a program P that responds to input events by calling the respective functions. The execution of these function will take the device to the next state.

Also the program P can be modelled as a transition system T_P based on the following sets, each one being isomorphic ($\cong$) to the corresponding set in T, or an extension to that set: (*i*) A set $N_P \cong N$ of node labels, each represented by an enumerator of the *node_label* type in P; (*ii*) a set $X_P = X_c \cup \{x_{\text{curr}}\}$ of variables, where $X_c \cong X$, each variable in X_c represents a context field of the *state* structure in P, and x_{curr} represents the *curr_node* of the *state* structure; (*iii*) a set $\mathbb{V}_P = \mathbb{V}_c \cup N_P$ of values, where $\mathbb{V}_c = \mathbb{V}$; (*iv*) a set $E_P \cong E$ of events, each one associated with one permission function and one transition function in P; (*v*) a set $G_P \cong G$ of guards, each implemented as the condition of an *if* statement in P; (*vi*) a denumerable set $V_P = V_c \cup V_n$ of valuations from X_P to $\mathbb{V}_P$, where $V_c \cong V$ and V_n: $\{x_{\text{curr}}\} \to N_P$; (*vii*) a set $A_P \cong A$ of arcs, where each arc has the form $(v_n(x_{\text{curr}}), v'_n(x_{\text{curr}}), e, g, v'_c)$, and each arc represents an *if* statement in the transition function for event e having guard g as its condition and valuation $v' = v'_n \cup v'_c$ as its controlled statement, with v'_n implemented by the *enter* function and v'_c by the assignments specified in the Emucharts diagram.

With the above definitions, let Q_P be a set of states where each state is a pair $\langle v_n, v_c\rangle$, with $v_n \in V_n$, $v_c \in V_c$. The transition relation $\xrightarrow{P} \subseteq Q_P \times Q_P$ is defined by the semantic rules in figure 7 applied to elements of the above sets, and implemented by the permission functions, which check for each event e if the condition $v_n(x_{\text{curr}}) = p$ holds or not, and by the transition functions, which check if the current values of the variables satisfy the guards, and update node and variables accordingly. The associated transition system T_P is the tuple $(Q_P, \xrightarrow{P}, q_{P0})$, where q_{P0} is the state defined by the initial values of x_{curr} and of the context variables, set by the *init* function. The operational semantics are given in Fig. 7.

A.3 Equivalence of the transition systems

To prove the correctness of the generated code, we introduce the definition of equivalence between Emucharts states and the program states.

Definition 1. *A member m of one of the sets N, X, $\mathbb{V}$, E, defined in T, is equivalent ($\sim$) to the member m_P paired to m by the isomorphism between the set containing m and the corresponding set in T_P.*

Definition 2. *A state $q = \langle n, v \rangle$, $q \in Q$, is equivalent ($\sim$) to a state $q_P = \langle v_n, v_c \rangle$, $q_P \in Q_P$ iff $n \sim v_n(x_{\text{curr}})$ — so the value of x_{curr} is equivalent to node n, and $\forall_{x \in X} v(x) = v_c(x_P)$ (i.e., matching variables in q and q_P have the same values).*

The proof of correctness for the generated code is by induction on the length of computation. We assume that T and T_P are the transition systems modelling, respectively, an Emucharts diagram and a program that uses the generated code, respecting the previously introduced protocol, and accepts a sequence of input events.

Theorem 1. *Let T and T_P be the transition systems introduced in the above paragraphs, and $e = e_1, e_2 \ldots$ be a sequence of input event sequences. Let $\sigma = q_0, q_1, \ldots$ and $\sigma_P = q_{P0}, q_{P1}, \ldots$ be sequences of states, with $q_i \rightarrow q_{(i+1)}$ and $q_{Pi} \xrightarrow{P} q_{P(i+1)}$.*

We prove that, at each step of the computation, $q_i \sim q_{Pi}$:
Induction base. $q_0 \sim q_{P0}$ by construction.
Induction step. Let $q_j \sim q_{Pj}$ at step j. On the occurrence of an event e, let $q_j \rightarrow q_{(j+1)}$ and $q_{Pj} \xrightarrow{P} q_{P(j+1)}$. We can prove that $q_{(j+1)} \sim q_{P(j+1)}$ by case analysis: (1) e not permitted in q_j; (2) e permitted and guard not satisfied; and (3) e permitted and guard satisfied.
Case 1: e not permitted. If the event is not permitted in the current state, rules **idle** and **idle**$_P$ apply to T and T_P, respectively, so that $q_{(j+1)} = q_j$ and $q_{P(j+1)} = q_{Pj}$, equivalent by induction hypothesis. Recall that the permission function for e returns *false* in this case, and by hypothesis program P does not call the corresponding transition function.
Case 2: e permitted and guard not satisfied. Also in this case, rules **idle** and **idle**$_P$ apply to the transition systems. The *if* statements in P check that the guard does not hold, and the respective controlled statements are not executed.
Case 3: e permitted and guard satisfied. In this case, Rules **arc** and **arc**$_P$ apply to both transition systems, therefore (*i*) T moves from state $q_j = \langle n, v \rangle$ to state $q_{(j+1)} = \langle n', v' \rangle$, or (*ii*) T_P moves from state $q_{Pj} = \langle v_n, v_c \rangle$ to state $q_{P(j+1)} = \langle v'_n, v'_c \rangle$. Valuation v'_n maps x_{curr} to a node label equivalent by definition to n', and v'_c maps the context variables in T_P to values equivalent by definition to those assigned by v' to the context variables in T'.

The new states in the two transition systems are therefore equivalent.