Rodin Platform Why3 plug-in

Alexei Iliasov, Paulius Stankaitis,
David Adjepon-Yamoah, Alexander Romanovsky

Newcastle University, UK

Abstract. We briefly present the motivation, architecture and usage experience as well as proof statistics for a new Rodin Platform proof back-end based on the Why3 umbrella prover. Why3 offers a simple and versatile notation as a common interface to a large number of automated provers including all the leading SMT-LIB and TPTP compliant tools. The plug-in can function either in a local mode when all the provers are installed locally, or remotely as a cloud service. We discuss the experience of building the tool, the current status and the potential advantages of a cloud-hosted proof infrastructure.

1 Overview

The Rodin Platform offers a fairly capable development and proof support for the Event-B specification language. Some of the automated provers are a part of the Platform and there is a number of add-on provers that significantly improve proof success. Two more important one are the Atelier-B ML prover and the SMT plug-in [3] that offers a bridge to a number of SMT-LIB compliant provers.

In addition to SMT-LIB interface, the majority of the prominent automated provers support the TPTP [6] interface that originated as a common notation for prover competitions.

Recently some important work has been done to bring a large number of TPTP and SMT-LIB provers under the roof of a common, versatile notation - the Why3 verification platform [1]. At the basic level Why3 offers a common interface to over a dozen of automated provers; it also has its own high-level specification notation to reason about software correctness though we do not make any use of it in this work and rather rely on Why3 to offer a bridge to tools like Z3 [4], SPASS [7], Vampire [5] and Alt-Ergo [2].

A theorem prover is a computationally and memory intensive program typically run for rather short periods of time (the vast majority of proofs is done within two seconds) with long idling periods in between. Proof success and perceived usability depend on the capability of an execution platform. Such requirement is best met by the cloud technology.

Doing proofs on a cloud opens possibilities that we believe were previously not explored, outside, perhaps, prover contests. The cloud service keeps a detailed record of each proof attempt along with (possibly obfuscated) proof obligations, supporting lemmas and translation rules. There is a fairly extensive library of Event-B models constructed over the past 15 years and these are a ready of source

Model	Total POs	Open, built-in	Open, built-in + SMT	Open, Why3
Multi-core runtime	625	281	62	18
Paxos	348	121	9	4
Fisher's alg.	82	14	2	0
Train Control System	133	36	5	32

Table 1. Performance benchmark for the cloud-based proving service.

of proof obligations. Some of these come from academia and some from industry. We are now starting to put models through our prover plug-in in order to collect some tens of thousands of proof obligations. One immediate point of interest is whether one can train a classification algorithm to make useful prediction of relative prover performance. If such a prediction can yield statistically significant results, prover call order may be optimized to minimize resource utilization while retaining or improving average proof time.

In order to convert Event-B into Why3 verification we had provide a mapping for various Event-B operators, especially its set-theoretic treatment of functions and relations. Unlike say, Isabelle/HOL, Why3 does not rely on a small proof kernel and allows one to make axiomatic definitions. It is a much quicker way to define an embedding of a logic but there is always a danger of making it unsound. In a simpler case, an unsound axiomatisation may be detected by proving of a tautological falsity but there are more intricate situations where unsound definitions show up only in specific circumstances (that is, unsound part is guarded by an implication or requires instantiation of some bound variables). A database of proof attempts makes detecting suspect changes much easier as we can go through historic proofs at any time to see if the outcome changes. We also perform negated proofs on thousands of saved proof efforts.

Table 1 shows the comparative performance of the plug-in for some models. Why3 plug-in at the moment is slower than the SMT plug-in but generally more capable though we have one example where its performance is much inferior to the SMT plug-in. The plug-in is open source and is available from the authors on request; we plan to release it with a public cloud service in the coming few months.

2 Translation

The translation of an Event-B proof sequent into a Why3 theory is split into following four activities: a lightweight syntactic translation; construction of a theory from an Event-B sequent and translated formulas; filtering of sequent hypotheses and support lemmas, Why3 axiomatisation of the Event-B mathematical language. Most of the effort goes in the last part so that the programmatic bit of translation is relatively lightweight and generic.

The syntactic translator is written in Tom/Java and simply pretty-prints an AST of an Event-B formula as an S-expression (which is, in essence, the input syntax of Why3) with a static mapping of Event-B operator names. Thus a formula $f \setminus \{t\}$ becomes (`diff` f (`singleton` t)). There two non-trivial mappings:

the folding of left- or right- associative multi-operators into equivalent binary forms, and the detection of enumerated set definitions (a native, algebraic definition of enumerated sets significantly improves prover performance).

An Event-B proof sequent is mapped into a Why3 theory. The Why3 language has a different treatment of types - type variables are explicit and are separate from the notion of a set - hence every carrier sets is defined twice: as a type variable and as a maximal set. For instance, carrier set CORES and enumerated set STATUS are translated into:

```
type tp_CORES
type tp_STATUS = T_ON | T_OFF

constant id_STATUS : (set tp_STATUS)
constant id_CORES : (set tp_CORES)

axiom hyp1 :(maxset id_STATUS)
axiom hyp2 :(maxset id_CORES)
```

where

```
 axiom maxset_axm:
     forall s:set 'a, x:'a .  (maxset s) -> mem x s
```

Free identifiers occurring in a sequent become constants of a Why3 theory; hypotheses are theory axioms and the sequent goal is mapped into a theory goal (i.e., a lemma).

The filtering of hypotheses and support conditions is essential to enable proof within reasonable time. It is discussed in Section3.

Most of the translation effort goes into the construction and fine-tuning of Why3 support theories. We define each Event-B operator in a separate theory and give the bare minimum axiomatic definition that must be checked by hand. For instance, the following is the cardinality operator defined inductively:

```
theory Cardinality
  use import Set
  ..
  use import ElementAddition

  function card (set 'a) : int

  axiom card_def1:
    forall s: set 'a.
      finite s /\ is_empty s -> (card s) = 0

  axiom card_def2:
    forall x : 'a, s : set 'a.
      finite s /\ not mem x s -> card (add x s) = 1 + card s
end
```

This is all one needs to know but not really enough to carry out proofs. Thus we construct and prove a fairly long list of support conditions. These are deposited in a separate theory (to facilitate filtering). The following gives an example of such support lemmas:

```
theory Cardinality_support
  use import Set
  ...
  use import Union
  ...
  lemma lemma_card_def5:
    forall s, t: set 'a.
      finite t /\ (forall x : 'a. mem x s -> mem x t) ->
        card s <= card t
  ...
  lemma lemma_card_def10:
    forall s : set 'a, t : set 'b, f : rel 'a 'b.
      finite t /\ (mem f s >->> t) ->
        card s = card t
  ...
  lemma lemma_card_def11:
    forall s : set 'a, t : set 'b, f : rel 'a 'b.
      card s = card t /\ (mem f s >-> t) ->
        mem f s -->> t
  ...
end
```

3 Hypotheses and lemma filtering

The initial experiments have shown that a minimal axiomatisation support is not sufficient to discharge a sizeable proportion of proof obligations. Provable lemmas were added to assist with specific cases but then it become clear that a large number of support conditions slow down or even preclude a proof. On top of that, the auto tactic language of Rodin offers a very crude hypotheses selection mechanism that for larger models tends to include tens if not hundreds of irrelevant statements. It was thus deemed essential to attempt to filter out unnecessary axiomatisation definitions, Why3 support lemmas and proof obligation hypotheses.

The Rodin mechanism for hypotheses is based on matching conditions with common free identifiers. To complement this mechanism we do filtering on the structure of a formula. It is also a natural choice since support lemmas do not have any free identifiers.

Directly comparing some two formulae is expensive: a straightforward algorithm (tree matching) is quadratic unless memory is not an issue. We use a computationally cheap proxy measure known as the Jaccard similarity which, as the first approximation, is defined as $JS(P,Q) = \mathrm{card}(P \cap Q) / \mathrm{card}(P \cup Q)$.

The key is in computing the number of overall and common elements and, in fact, defining what an "element" means for a formula. One immediate issue is that P and Q are sets and a formula, at a syntactic level, is a tree. One common way to match some two sequences (e.g., bits of text) using the Jaccard similarity is to use *shingles* of elements to attempt to capture some part of the ordering information. A shingle is a tuple preserving order of original elements but seen as an atomic element. Thus sequence $[a, b, c, d]$ could be characterised by two 3-shingles $P = \{[a, b, c], [b, c, d]\}$ (here $[b, c, d]$ is but a structured name) and matching based on these shingles would correctly show that $[a, b, c, d]$ is much closer to $[a, b, c, d, e]$ than to $[d, c, b, a]$. Trees are slightly more challenging. On one hand, a tree may be seen (but not defined uniquely) as a set of paths from a root to leaves and we could just do matching on a set of sequences and aggregate the result. This is not completely satisfactory as tree structure is not accounted for. So we add another characterisation of tree as a set of sequences of the form $[p, c_1, \ldots, c_2]$ where p is a parent element and $c_1, \ldots, c_2$ are children. This immediately gives a set of n-shingles that might need to be converted into shorter m-shingles to make things practical.

As an example, consider the following expression $a * (b + c/d) + e * (f - d * 2)$. We are not interested in identifiers and literals so we remove them to obtain tree $+(*(+/))(*(-*))$ which has the following 3-shingles based on paths, $[*, +, /], [+, *, +], [+, *, -], [*, -, *]$, and only 1 3-shingle, $[+, *, *]$, based on the structure. The shingles are quite cheap to compute (linear to formula size) and match (fixed cost if we disregard low weight shingles, see below). Let $\mathrm{sd}(P)$ and $\mathrm{sw}(P)$ be set of depth and structure shingles of formula P. Then the similarity between some P and Q is computed as

$$s(P, Q) = \sum_{i \in I_1} \mathrm{w_d}(i) + c \sum_{i \in I_2} \mathrm{w_w}(i) \qquad I_1 = \mathrm{sd}(P) \cap \mathrm{sd}(Q), I_2 = \mathrm{sw}(P) \cap \mathrm{sw}(Q)$$

where $w_*(i) = \mathrm{cnt}(i)^{-1}$ and $\mathrm{cnt}(i)$ is number of times i occurs in all hypotheses and support lemmas. Very common shingles contribute little to the similarity assessment and may be disregarded so that there is some k such that $\mathrm{card}(I_1) < k, \mathrm{card}(I_2) < k$.

4 Prover scenarious

The cloud service accepts as inputs sequents $\mathcal{S}$ of the form: (p, t) where p defines a proof scenario stipulating which provers need to be run and t is a Why3 theory containing a single goal.

The server executes a proof scenario p to obtain a proof result. A proof scenario is a function from an input sequent to a proof result: $q \in \mathcal{S} \to \{\mathsf{unknown}, \mathsf{valid}, \mathsf{invalid}\}$ and is defined via the following proof scenario primitives:

$$\begin{array}{rll} p(t) := & \mathsf{pr}(t) & \text{(prover call, positive)} \\ & \mid \mathsf{pr}(\neg t) & \text{(prover call, negative)} \\ & \mid p(t) \, \triangleright \, w & \text{(deadline)}, w \in \mathbb{R}_+ \\ & \mid p_1(t) \, \wedge \, p_2(t) & (\textit{and} \text{ composition}) \\ & \mid p_1(t) \, \vee \, p_2(t) & (\textit{or} \text{ composition}) \\ & \mid \neg p(t) & \text{(result negation)} \end{array}$$

The negation operator $\neg$ on proof results turns valid into invalid, invalid into

valid and does not affect $\mathsf{unknown}$ and failed.

The *or* composition $(p_1 \vee p_2)(t)$ is opportunistic: it may return any result r such that $r \in \{p_1(t), p_2(t)\} \setminus \{\mathsf{unknown}\}$, and when no such result exists, returns $\mathsf{unknown}$. The *and* composition $(p_1 \wedge p_2)(t)$ evaluates $\max(\{p_1(t), p_2(t)\})$ where $\mathsf{unknown} < \mathsf{valid} < \mathsf{invalid}$.

The compositions are distributive and commutative so that provers invocations may be scheduled rather flexibly or invoked at the same time. In practical terms, the *or* composition runs until any prover returns a definite results and the *and* composition runs all the provers until it sees $\mathsf{invalid}$ result.

The multiplicity of (independently developed) back-end verification tools may be relied on to increase the confidence in a proof result by applying adjudicating on the results of prover calls: $SA(t) \equiv \mathsf{pr}_1(t) \wedge \mathsf{pr}_2(t) \wedge \mathsf{pr}_3(t) \wedge \ldots$.

An important case is proving both positive and negative forms of an input sequent: $\mathsf{pr}_1(t) \wedge \neg \mathsf{pr}_1(\neg t)$. Negation may also be employed opportunistically with the parallel composition: $\mathsf{pr}_1(t) \vee \neg \mathsf{pr}_1(\neg t)$.

Provers may be run with a timeout. A practical example is to run a less capable but often faster prover in parallel with a slower prover: $\mathsf{pr}_1(t) \triangleright w_1 \vee \mathsf{pr}_2(t) \triangleright (w_1 + w_2), w_1, w_2 \in \mathbb{R}_+$.

An efficient implementations of both sequential and parallel compositions must rely on concurrent invocation of some or all of the composed prover calls.

References

1. François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, August 2011.
2. Sylvain Conchon, Évelyne Contejean, Johannes Kanig, and Stéphane Lescuyer. CC(X): Semantical combination of congruence closure with solvable theories. In *Post-proceedings of the 5th International Workshop on Satisfiability Modulo Theories (SMT 2007)*, volume 198(2) of *Electronic Notes in Computer Science*, pages 51–69. Elsevier Science Publishers, 2008.
3. Y. Guyot L. Voisin D. Deharbe, P. Fontaine. Integrating SMT solvers in Rodin. *Science of Computer Programming*, 94, Part 2:130 – 143, 2014.
4. N. Bjørner L. De Moura. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08. Springer, 2008.
5. A. Voronkov L. Kovács. First-order theorem proving and vampire. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV)*, 2013.
6. TPTP. Thousands of Problems for Theorem Provers. Available at www.tptp.org/.
7. Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22 : 22nd International Conference on Automated Deduction*, volume 5663 of *LNCS*, pages 140–145. Springer, 2009.