Preserving User Proofs Across Specification Changes*

François Bobot[3], Jean-Christophe Filliâtre[1,2], Claude Marché[2,1], Guillaume Melquiond[2,1], and Andrei Paskevich[1,2]

[1] Lab. de Recherche en Informatique, Univ. Paris-Sud, CNRS, Orsay, F-91405
[2] INRIA Saclay – Île-de-France, Orsay, F-91893
[3] CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette France

Abstract. In the context of deductive program verification, both the specification and the code evolve as the verification process carries on. For instance, a loop invariant gets strengthened when additional properties are added to the specification. This causes all the related proof obligations to change; thus previous user verifications become invalid. Yet it is often the case that most of previous proof attempts (goal transformations, calls to interactive or automated provers) are still directly applicable or are easy to adjust. In this paper, we describe a technique to maintain a proof session against modification of verification conditions. This technique is implemented in the Why3 platform. It was successfully used in developing more than a hundred verified programs and in keeping them up to date along the evolution of Why3 and its standard library. It also helps out with changes in the environment, *e.g.* prover upgrades.

1 Introduction

The work presented in this paper arose as a part of ongoing development and use of the Why3 system. Though we believe that our methods are applicable and useful in diverse settings of automated deduction, it would be most natural to introduce them in the context of our own project.

Why3 is a platform for deductive program verification. It provides a rich language, called WhyML, to write programs [9] and their logical specifications [4, 8], and it relies on external theorem provers, automated and interactive, to discharge verification conditions. Why3 is based on first-order logic with rank-1 polymorphic types, algebraic data types, inductive predicates, and several other extensions. When a proof obligation is dispatched to a prover that does not support some language features, Why3 applies a series of encoding transformations in order to eliminate, for example, pattern matching or polymorphic types [5]. Other transformations, such as goal splitting or insertion of an induction hypothesis, can be manually invoked by a user upon individual subgoals.

* This work is partly supported by the Bware (ANR-12-INSE-0010, `http://bware.lri.fr/`) project of the French national research organization (ANR), and the Hi-lite project (`http://www.open-do.org/projects/hi-lite/`) of the System@tic ICT cluster of Paris-Région Île-de-France.

To keep track of verification progress and to ensure that a once attained proof can be rechecked later, Why3 records applied transformations and proof attempts (calls to interactive and automated theorem provers). Maintaining this record against changes in proof obligations (that may ensue from changes in specification, program, or VC generation algorithm) is a difficult task, which, fortunately, can be automated to a certain degree. This paper deals with mechanisms of such automation.

Let us consider a typical user workflow in Why3. The user, an enlightened programmer named Alice, desires to formally verify an intricate algorithm. She starts by writing down the program code, fixes one or two simple mistakes (readily spotted by static typing), and, before annotating the program with any pre- or postconditions, runs the interactive verifier to perform the safety checks against out-of-bounds array accesses, arithmetic overflows, division by zero, etc. Why3 presents the whole verification condition for a given function as a single goal, and thus the first step is to split it down to a number of simple proof obligations and then to launch the provers, say, Alt-Ergo [3], CVC4 [1], or Z3 [7] on each of them. Let each safety condition be satisfied, except for one, which requires an integer parameter to be positive. Alice writes a suitable precondition, effectively putting the whole VC under an implication. It is now the verifier's job to detect that the once proved goals changed their shape and have to be reproved. Besides, since Alice's algorithm is recursive, a new proof obligation appears at each recursive call.

Alice continues to work on the specification; she adds the desired functional properties and regularly runs the verifier. The program's VC grows and, in addition to the first split, other interactive interventions are required: more splits, an occasional definition expansion, a call to an SMT solver with ten times the default time limit, a call to an interactive proof assistant to prove an auxiliary lemma by triple induction. The verified program now carries a complex proof script, which we call a *session*: a tree of goal transformations and a history of individual proof attempts at the leaves of that tree.

Almost every modification in the code or in the specification changes (even if slightly) almost every verification condition, requiring the proofs to be redone. Keeping the automated provers running on almost the same goals is only part of the bother. What could quickly make the verification process unmanageable is reconstructing, manually and every time, the proof session: finding those particular subgoals that required an increased time limit, a definition expansion, a Coq proof. Subgoals do not have persistent names, they may appear and vanish, and their respective order may change. Thus the only way to rebuild a proof is to look for similarities between the new goals and the old ones in order to find out where to re-apply the transformations and to re-launch the provers. This is a task where computer assistance would be highly appreciated.

Meanwhile, Alice finishes her program and puts it—code, specification, and proof—on her web page, so that it can be read and rechecked by other enlightened programmers. Three weeks later, a new version of an SMT solver used in the session is released, and the session file must be updated. Five weeks later, a

new release of Why3 comes out: it features an improved VC generator as well as numerous additions and corrections in the standard library. The latter affects the premises of proof obligations, the former, conclusions, so that the proof session has to be updated again.

Just like Alice, we prefer to be busy developing and proving new programs. Therefore, we have devised and implemented in Why3 a set of algorithms that maintain proof sessions, keep them up to date across prover upgrades and, most importantly, across changes in verification conditions. In the best course of events, Why3 is able to rebuild the proof session fully automatically, leaving to the user just the new subgoals (for which no previous proof attempts were made) or the ones that cannot be reproved without user intervention (typically when a Coq proof script requires modifications). These algorithms are the subject of the current paper.

In Section 2, we give a formal description of a Why3 proof session. Section 3 contains the algorithm of goal pairing that is used to rebuild proof sessions. In Section 4, we discuss additional measures to maintain proof scripts for interactive proof assistants like Coq or PVS. In Section 5, we explain how to configure and use Why3 in an environment of multiple automated and interactive provers.

2 Proof Sessions: Static Model

Transformations and proof attempts applied to proof obligations are stored in a tree-like structure, called *proof session*. We describe it in this section.

Proof Attempts. A prover is characterized by a name, a version number, and a string field that is used to discriminate different ways to call the same prover.

$$\mathit{prover} \ ::= \ \langle \mathit{name}, \mathit{version}, \mathit{options} \rangle$$

A proof attempt describes a call to an external prover.

$$\begin{aligned} \mathit{proof\_attempt} &::= \langle \mathit{prover}, \mathit{timelimit}, \mathit{memlimit}, \mathit{result} \rangle \\ \mathit{result} &::= \langle \mathit{time}, \mathit{status} \rangle \\ \mathit{status} &::= \texttt{valid} \mid \texttt{invalid} \mid \texttt{unknown} \mid \\ &\qquad \texttt{timeout} \mid \texttt{outofmemory} \mid \texttt{failure} \end{aligned}$$

Information is the prover, the maximal amount of CPU time and memory given to the prover, and the result of that call. A result is a pair: the time of the execution of the external process, and the prover outcome (*status*). Such a status is obtained by matching the prover output using regular expressions or by looking at its exit code. A status has six possible values: one for a successful proof attempt (`valid`), and five unsuccessful ones. Status `invalid` means that the prover declared the goal to be invalid; `unknown` means an inconclusive outcome (neither `valid` nor `invalid`) before the time limit is reached; `timeout` (resp. `outofmemory`) means the prover had been stopped because it exhausted the given resources; and `failure` means any other reason for an unsuccessful execution. This is similar to the SZS *no-success* ontology [14].

Proofs and Transformations. The entities for proof tasks (*proof_task*) and transformations (*transf*) have the following structure:

$$\begin{aligned} \mathit{proof\_task} &::= \langle \mathit{name}, \mathit{expl}, \mathit{goal}, \mathit{proof\_attempt}*, \mathit{transf}*, \mathit{verified} \rangle \\ \mathit{transf} &::= \langle \mathit{name}, \mathit{proof\_task}*, \mathit{verified} \rangle \\ \mathit{verified} &::= \texttt{true} \mid \texttt{false} \end{aligned}$$

A proof task is characterized by a name, an explanation (a text describing its origin *e.g.* "loop invariant preservation"), and a goal. A proof task contains a collection of proof attempts, as well as a collection of transformations. There is no contradiction in having a proof task with both proof attempts and transformations. A transformation has a name (as registered in Why3 kernel) and a collection of sub-tasks. A proof task has status *verified* if and only if there is at least one proof attempt with status `valid` or one transformation with status *verified*. A transformation has status *verified* if and only if all its sub-tasks have status *verified*.

Theories, Files, and Sessions. A theory has a name and a collection of proof tasks. A file has a pathname (relative to the session file) and a collection of theories. A proof session is a set of files:

$$\begin{aligned} \mathit{theory} &::= \langle \mathit{name}, \mathit{proof\_task}*, \mathit{verified} \rangle \\ \mathit{file} &::= \langle \mathit{pathname}, \mathit{theory}*, \mathit{verified} \rangle \\ \mathit{proof\_session} &::= \mathit{file}* \end{aligned}$$

A theory has status *verified* if and only if all its tasks are verified. A file has status *verified* if and only if all its theories are verified.

Example. In Fig. 1, we show an example of a simple session. It consists of one file, `f_puzzle.why`, which contains one theory, `Puzzle`. This theory, whose WhyML source is shown in the bottom right corner, introduces an uninterpreted function symbol `f` and two axioms:

```
function f int: int
axiom H1: forall n: int. 0 <= n → 0 <= f n
axiom H2: forall n: int. 0 <= n → f (f n) < f (n+1)
```

Our final goal consists in revealing that `f` is the identity on natural numbers:

```
goal G: forall n: int. 0 <= n → f n = n
```

To that purpose, we use four simple lemmas and two instances of the induction scheme on natural numbers, provided by the Why3 standard library. We also apply a transformation called `split_goal_wp` to split a conjunction into two separate subgoals (see lemma `L3` on Fig. 1). Each subgoal is successfully verified due to the combined effort of three automated provers.

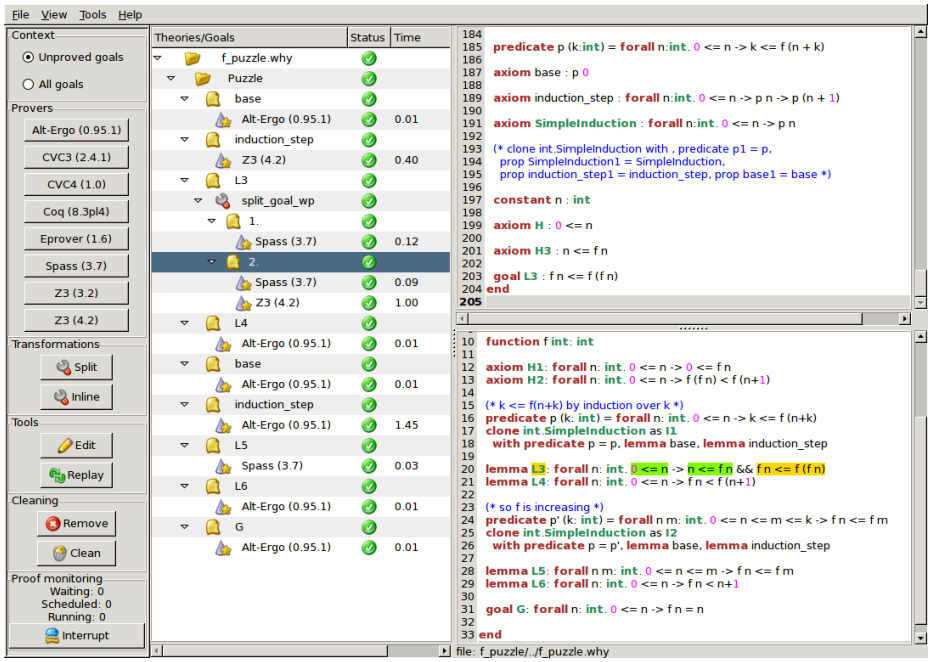

Fig. 1. An example of Why3 session.

3 Session Updates

The problem we address in this section is to update a proof session when the set of goals changes. There are many possible reasons for such a change: a user modification of a goal statement, a modification of a goal context (*e.g.* the introduction of additional hypotheses), a modification of a program and/or its specifications resulting in a different VC, etc. Wherever the change comes from, the problem boils down to matching an old proof session (typically stored on disk during a previous verification process) with a new collection of files, theories, and goals. Such a matching is performed on a file and theory-basis, where files and theories are simply identified by names.[4]

This matching process is performed recursively over the tree structure of a session. Given the collection of (old) proof tasks and a collection of new goals, we match each new goal g either to an old task t or to a freshly created task with no proof attempt and no transformation. In the former case, each transformation Tr of the old task t is applied to the new goal g, resulting into a collection of new goals. Then we proceed recursively, matching these new goals with the sub-tasks of Tr.

[4] We could provide refactoring tools to rename files and/or theories, but this is not what is discussed in this paper.

$$
\begin{aligned}
\overline{n} &= \texttt{'c'} + n \\
\overline{x} &= \texttt{'V'} + \mathit{unique}(x) \quad \text{if } x \text{ is a local variable} \\
&= \texttt{'V'} + x \quad \text{otherwise} \\
\overline{\texttt{true}} &= \texttt{'t'} \\
\overline{\texttt{false}} &= \texttt{'f'} \\
\overline{f(t_1, \ldots, t_n)} &= \texttt{'a'} + f + \overline{t_1} + \ldots + \overline{t_n} \\
\overline{\forall x : \tau.\, t} &= \overline{t} + \texttt{'F'} \\
\overline{t_1 \Rightarrow t_2} &= \overline{t_2} + \texttt{'I'} + \overline{t_1} \\
\overline{\neg\, t} &= \texttt{'N'} + \overline{t} \\
\overline{\texttt{let } x = t_1 \texttt{ in } t_2} &= \overline{t_2} + \texttt{'L'} + \overline{t_1} \\
\overline{\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3} &= \texttt{'i'} + \overline{t_3} + \overline{t_2} + \overline{t_1}
\end{aligned}
$$

Fig. 2. Shapes of terms and formulas.

We are now left with the sub-problem of pairing a collection of old goals (and their associated tasks) and a collection of new goals. We first pair goals that are exactly the same.[5] In a second step, we pair remaining goals using a heuristic measure of similarity based on a notion of goal *shape*.

3.1 Goal Shape

The shape of a goal is a character string. The similarity between two goals is defined as the length of the common prefix of their shapes. To match our intuition of logical similarity, we adopt the following principles for computing shapes:

- shapes should take explanations into account, so that only goals with same explanations are paired;
- shapes are invariant by renaming of bound variables;
- conclusion is more important than hypotheses, *e.g.* the shape of an implication $A \Rightarrow B$ is built from the shape of B first, and then the shape of A.

Declarations, definitions, and axioms are disregarded when computing shapes. There are two reasons: first, it keeps shapes reasonably small; second, it is unlikely that two goals differ only in their contexts. The shape of a term or formula t, written $\overline{t}$, is recursively defined over the structure of t, as given in Fig. 2. The shape computation is not injective: two formulas may have the same shape. It is not an issue for us, as we only use shapes as an heuristic for pairing.

Let us consider the goal

```
forall x: int. f x = x
```

Its shape is the string `a=afV0V0F`. If we modify it into the following goal

[5] Technically, since the old goal is not stored on disk, we detect identical goals using MD5 checksums.

```
forall n: int. 0 <= n → f n = n
```

then its shape becomes the string `a=afV0V0Ia<=c0V0F`. These two shapes share a common prefix of length 8, that is `a=afV0V0`. As illustrated on this example, bound variables are mapped to unique integers, numbered from zero for a given goal.

3.2 Matching Algorithm

We are given a collection of N "old" shapes and a collection of M "new" shapes. This section describes an algorithm that tries to map each new shape to an old one. We note $lcp(u, v)$ the length of the longest common prefix of strings u and v, *i.e.* the largest k such that $u_i = v_i$ for all $0 \leq i < k$. We choose the following greedy algorithm, which repeatedly picks up the pair that maximizes the length of the common prefix.

new ← new shapes
old ← old shapes
while *new* $\neq \emptyset$ **and** *old* $\neq \emptyset$
  find o in *old* and n in *new* such that $lcp(o, n)$ is maximal
  pair o and n
  old ← *old* $- \{o\}$
  new ← *new* $- \{n\}$

Notice that goal o is removed from set *old* as soon as it is paired with a new goal. We could have chosen to keep it, in order to pair it later with another new goal. However, the purpose of our algorithm is to reassign former successful proofs to new goals, and not to discover new proofs.

Given as such, this algorithm is inefficient. Let shapes have maximal length L. Assuming $N = M$, the algorithm has complexity $O(LN^3)$, since finding the pair that maximizes *lcp* is $O(LN^2)$. One can turn this into a more efficient algorithm, by making a list of all shapes (be they old or new) and then sorting it in lexicographic order. In that case, the pair (o, n) that maximizes $lcp(o, n)$ is necessarily composed of two shapes n and o that are consecutive in the list. So finding the pair becomes linear and the overall complexity is now $O(LN^2)$. It is even possible to reduce this complexity using a priority queue containing all pairs (o, n) of consecutive shapes (either old/new or new/old), ordered according to $lcp(o, n)$. As long as the priority queue is not empty, we extract its maximal element (o, n), we pair the corresponding shapes whenever both o and n are not yet already paired to another shape, and we (possibly) insert a new pair in the priority queue. The cost of sorting is $O(LN \log N)$ and, assuming a priority queue with logarithmic insertion and extraction, the cost of repeated extractions and insertions is also $O(N(L+\log N))$ (there is at most one insertion for each extraction, and thus the priority queue never contains more than $2N-1$ elements). Whenever $N \neq M$, the cost of sorting is dominating and thus we have a total cost $O(L(N + M) \log(N + M))$.

A property of the algorithm above is that, whenever there is at least as many new shapes as old ones, each old shape gets paired with a new one. Said otherwise, no former proof task is lost. When there are less new shapes, however, some shapes cannot be paired and, subsequently, former proof tasks are lost.[6]

4 Script Updates for Interactive Provers

Updating sessions is sufficient for handling transformations and calls to automated provers, since their inputs are just Why3 goals which are parts of sessions. For interactive proof assistants, the situation is slightly different. Indeed, an interactive proof script is the mix of a skeleton generated from a goal and an actual proof written by the user. Currently, Why3 supports two proof assistants: Coq and PVS. Yet the ideas presented in this section apply to any interactive proof assistant that supports a textual input.

In a nutshell, Why3 outputs a theorem statement, together with definitions and axioms corresponding to the Why3 context for that goal. Then the user writes commands for guiding the proof assistant towards a proof of that statement. The user may introduce auxiliary lemmas and definitions for proving the main theorem.

In Coq, proof commands are part of the same file as the definitions and theorem statements, while in PVS, they are usually stored in a separate file. Still, in both cases, user statements and Why3-generated statements are intermingled in the proof script. When a session is updated, the context and the statement of the main theorem might change, so the proof script needs to be regenerated. There are two main issues though, which are not present for automated provers. First, it is important not to lose any part of the script the user might have painstakingly written. Second, while preserving user parts, it is important to discard parts previously generated by Why3, since they are now obsolete.

As far as Why3 is concerned, a proof script is simply a sequence of definitions and facts, each of them possibly followed by its proof (that is, a sequence of commands). Why3 makes the following assumptions: axioms were generated by Why3 itself, while proof commands, if any, were written by the user. For definitions and theorem statements, there is an ambiguity, so Why3 annotates them with a comment when they are generated. These comments have a low impact on readability, since most entries do not need any disambiguation. The following Coq script is the one generated for the running example; all the `Axiom` and `Parameter` statements are generated by Why3; this is also the case for the theorem statement, and it is prefixed by an annotation so that it is not mistaken for some user content; finally, proof commands such as `intros` are written by the user.

```
Parameter f: Z -> Z.
Axiom H1 : forall (n:Z), 0 <= n -> 0 <= f n.
```

[6] We could devise some kind of `lost+found` pool of abandoned proofs, to be used in subsequent rounds of the matching algorithm or to be manually selected by the user.

```
Axiom H2 : forall (n:Z), 0 <= n -> f (f n) < f (n + 1).
... (* other Why3 statements *)

(* Why3 goal *)
Theorem G : forall (n:Z), 0 <= n -> f n = n.
intros n h1.
... (* other user commands *)
Qed.
```

Regarding script regeneration, Why3 takes the following approach. Whenever it needs to output a statement, if a statement with the same name is already present in the old script, it first outputs any user content that was preceding it. If the user had attached commands to that statement, they are output. This seemingly simple process is actually quite effective in practice.

Note that, while this mechanism is currently applied only to interactive proof assistants, it might also make sense for automated provers. Indeed, some of them accept user hints for guiding proof search. For instance, an SMT solver may require some facts to be removed from the context, or some triggers to be modified. Alt-Ergo supports such user hints. Another example is Gappa, which only performs branch-and-bound when instructed, so the user should have a chance of modifying the Gappa script beforehand to add this kind of hint.

5 Environment Changes

The environment is composed by the installed version of Why3 and the installed provers. This environment changes when users upgrade Why3 or one of the provers.

For the first case, Why3's developers try to keep backward compatibility in every aspect of Why3. Unsurprisingly, that encompasses backward compatibility of the application programming interface, but also backward compatibility of the session on-disk format. More indirectly, modifications of the weakest precondition calculus, of the simplifications, and of the transformations, are done so as to keep provability whenever possible. This is checked during the nightly regression test which verifies that all the program examples from the Why3 gallery are still proved. Moreover, this test suite also exercises the mechanism of session update, since pairing breakage would cause some goals to become unproved.

For the second case, Why3 offers a tool for auto-detection of provers, called `why3config`. According to the name and version of the prover, it selects the configuration file, called *driver*, that specifies the built-in functions and the transformations to apply before sending a goal to the prover. When a new version of the prover is released, a new version of the driver is created. Old drivers are kept, so that older versions can still be used. If the user upgrades a prover and runs `why3config --detect`, then Why3 asks, when old sessions are open, whether to copy or move proof attempts to the newer version of the prover.

In order to compare the results of different provers or in order to update proofs incrementally, a user can install different versions of the same prover

at the same time. The `why3session` command-line tool allows to copy/move proof attempts done with one prover to another prover or to compare results of different provers.

6 Conclusions, Related Work, and Perspectives

We described in this paper the way we designed a proof session system in Why3. The technical choices were guided by the general need of maintaining proofs across specification changes. The same technique is also useful in case of changes in the system itself: upgrade of Why3's kernel, upgrade of the standard library, upgrade of external provers. Our session system allowed us to maintain, for more than 2 years now, a gallery of verified programs (`http://toccata.lri.fr/gallery/index.en.html`) containing more than 100 examples. Several versions of Why3 and of external provers were released during this period. Moreover, session files are available on that website, so that anyone should be able to replay the proofs.

The contributions of this paper are mainly technical, not much scientific in the noble sense. Nevertheless we believe that our design of proof sessions are worth publicizing, hoping that some ideas can be useful to others. Indeed, writing this paper allowed us to discover a few subtle bugs in our implementation.

We found few related works in the literature. An early work by Reif and Stenzel in 1993 [12] aimed at managing changes in the context of the KIV verification system. Some ideas were reused by V. Klebanov in 2009 [10] for managing changes in proof scripts made inside the KeY system [2]. They both introduce a notion of similarity of goals, although different from ours. Indeed, their aim was to manage changes in interactive proof scripts, which is only a part of our own aim. It is not really meaningful to compare these works with our own approach, since in their case, they have a whole proof object at hand, performed by a single prover, in which they can search for example if a lemma is used or not. We are instead dealing with small pieces of proof made by different provers.

Note that some deductive verification systems rely on a single automated prover and express proof skeletons at the source level only (*e.g.* lemmas, ghost code, but no Why3-like transformations). Thus they do not have a need for proof management, as all the proof obligations will be handled the same way. This is the case for VCC, Dafny, Verifast, and so on.

We also found some attempts at designing large shared databases for bookkeeping proofs. The *Evidential Tool Bus* is a first step towards this idea by J. Rushby in 2005 [13]. Recently, Cruanes, Hamon, Owre, and Shankar [6] presented a formal setting on how several independent tools can cooperate and exchange proofs on such a tool bus. A similar effort is the goal of D. Miller's ProofCert project[7], where a general framework for a common proof format is proposed [11]. As far as we understand, the issue of maintaining proofs across specification changes is not yet addressed in these settings. We hope that our techniques could be useful in these contexts.

[7] `http://www.lix.polytechnique.fr/Labo/Dale.Miller/ProofCert.html`

Acknowledgments We would like to thank Rustan Leino for suggesting the example of this paper. We also thank the anonymous reviewers for their useful comments and suggestions.

References

1. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.
2. B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*, volume 4334 of *Lecture Notes in Computer Science*. Springer, 2007.
3. F. Bobot, S. Conchon, E. Contejean, M. Iguernelala, S. Lescuyer, and A. Mebsout. The Alt-Ergo automated theorem prover, 2008. `http://alt-ergo.lri.fr/`.
4. F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. Why3: Shepherd your herd of provers. In *Boogie 2011: First International Workshop on Intermediate Verification Languages*, pages 53–64, Wrocław, Poland, August 2011.
5. F. Bobot and A. Paskevich. Expressing Polymorphic Types in a Many-Sorted Language. In C. Tinelli and V. Sofronie-Stokkermans, editors, *Frontiers of Combining Systems, 8th International Symposium, Proceedings*, volume 6989 of *Lecture Notes in Computer Science*, pages 87–102, Saarbrücken, Germany, Oct. 2011.
6. S. Cruanes, G. Hamon, S. Owre, and N. Shankar. Tool integration with the evidential tool bus. In R. Giacobazzi, J. Berdine, and I. Mastroeni, editors, *Verification, Model Checking, and Abstract Interpretation, 14th International Conference, VMCAI 2013*, volume 7737 of *Lecture Notes in Computer Science*, pages 275–294, Rome, Italy, 2013. Springer.
7. L. de Moura and N. Bjørner. Z3, an efficient SMT solver. In *TACAS*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.
8. J.-C. Filliâtre. Combining Interactive and Automated Theorem Proving in Why3 (invited talk). In K. Heljanko and H. Herbelin, editors, *Automation in Proof Assistants 2012*, Tallinn, Estonia, April 2012.
9. J.-C. Filliâtre and A. Paskevich. Why3 — where programs meet provers. In M. Felleisen and P. Gardner, editors, *Proceedings of the 22nd European Symposium on Programming*, volume 7792 of *Lecture Notes in Computer Science*, pages 125–128. Springer, Mar. 2013.
10. V. Klebanov. *Extending the Reach and Power of Deductive Program Verification*. PhD thesis, Universität Koblenz-Landau, 2009. `http://formal.iti.kit.edu/~klebanov/pubs/vstte09.pdf`.
11. D. Miller and E. Pimentel. A formal framework for specifying sequent calculus proof systems. *Theoretical Computer Science*, pages 98–116, 2013.
12. W. Reif and K. Stenzel. Reuse of proofs in software verification. In R. K. Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science, 13th Conference*, volume 761 of *Lecture Notes in Computer Science*, pages 284–293, Bombay, India, 1993. Springer.
13. J. M. Rushby. An evidential tool bus. In K.-K. Lau and R. Banach, editors, *ICFEM*, volume 3785 of *Lecture Notes in Computer Science*, pages 36–36. Springer, 2005.
14. G. Sutcliffe. The SZS ontologies for automated reasoning software. In P. Rudnicki, G. Sutcliffe, B. Konev, R. A. Schmidt, and S. Schulz, editors, *Proceedings of the LPAR 2008 Workshops, Knowledge Exchange: Automated Provers and Proof*

Assistants, and the 7th International Workshop on the Implementation of Logics, volume 418 of *CEUR Workshop Proceedings*, pages 38–49, Doha, Qatar, 2008.