Why3: Shepherd Your Herd of Provers*

François Bobot[1,2], Jean-Christophe Filliâtre[1,2],
Claude Marché[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

Abstract. Why3 is the next generation of the Why software verification platform. Why3 clearly separates the purely logical specification part from generation of verification conditions for programs. This article focuses on the former part. Why3 comes with a new enhanced language of logical specification. It features a rich library of proof task transformations that can be chained to produce a suitable input for a large set of theorem provers, including SMT solvers, TPTP provers, as well as interactive proof assistants.

1 Introduction

Why3 is the next generation of the Why software verification platform. In this article, we present it as an environment for logical specification that targets a multitude of automated and interactive theorem provers. It provides a rich syntax based on first-order language and a highly configurable toolkit to convert specifications into proof obligations in various formats.

The development of Why3 is mainly motivated by the necessity to model the behavior of programs (both purely applicative and imperative) and formally prove their properties. It is commonly admitted that verification of non-trivial programs requires designing a pure logical model of the considered programs. In JML [10] or Spec# [2], for instance, such models are described using the pure fragment of the underlying programming language. In the L4.verified project [18], the Haskell language is used to model the C code of a micro-kernel. Proof assistants such as Coq [6], PVS [25], or Isabelle [17] also provide rich specification languages that are convenient to model programs.

Why3 distinguishes itself from the aforementioned approaches in that we want to provide as much automation as possible. Instead of being a theorem prover by itself, Why3 intends to provide a front-end to third-party theorem provers. To this end, we propose a common specification language which aims at maximal expressiveness without sacrificing efficiency of automated proof search

* This work was partly funded by the U3CAT (ANR-08-SEGI-021, `http://frama-c.com/u3cat/`) and DECERT (ANR-08-DEFI-005, `http://decert.gforge.inria.fr/`) projects 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.

(Section 2). Another challenge is modular specification. Our proposal is a notion of reusable theories and an associated mechanism of "cloning" (Section 3). As we target a large set of theorem provers whose language and logic range from mono-sorted first-order logic to many-sorted first-order logic modulo theories to the Calculus of Inductive Constructions, we provide an extensible framework to translate the language of Why3 to these various logic languages (Section 4). Finally, we briefly describe the set of tools in the current distribution of Why3 (Section 5).

2 Logic

The logic of Why3 is a first-order logic with polymorphic types and several extensions: recursive definitions, algebraic data types and inductive predicates.

Types. A type can be non-interpreted, an alias for a type expression or an algebraic data type. For instance, the type of polymorphic binary trees is introduced as follows:

```
type tree 'a = Leaf | Node (tree 'a) 'a (tree 'a)
```

A particular case of algebraic types are enumerations.

```
type answer = Yes | No | MayBe
```

Built-in types include integers (`int`), real numbers (`real`) and polymorphic tuples. The following declaration defines the type `answer_tree` as an alias for a pair:

```
type answer_tree 'a = (tree 'a, answer)
```

Function and Predicate Symbols. Every function or predicate symbol in Why3 has a (polymorphic) type signature. For example, an abstract function that merges two integer trees can be declared as follows:

```
function merge (tree int) (tree int) : tree int
```

Both functions and predicates can be given definitions, possibly mutually recursive. As examples, we can calculate the height of a tree

```
function height (t: tree 'a) : int = match t with
  | Leaf       -> 0
  | Node l _ r -> 1 + max (height l) (height r)
end
```

or test whether the elements of a tree are both sorted and within given bounds

```
predicate sorted (t: tree int) (min: int) (max: int) =
  match t with
    | Leaf -> true
    | Node l x r ->
        sorted l min x /\ min <= x <= max /\ sorted r x max
  end
```

Why3 automatically verifies that recursive definitions are terminating. To do so, it looks for an appropriate lexicographic order of arguments that guarantees a structural descent. Currently, we only support recursion over algebraic types. Other kinds of recursively defined symbols have to be axiomatized. In future versions of Why3, we plan to allow annotating recursive definitions with termination measures. Such definitions would generate proof obligations to ensure termination.

Another extension to first-order language adopted in Why3 is inductive predicates. Such a predicate is the least relation satisfying a set of clauses. For instance, the subsequence relation over finite lists is inductively defined as follows:

```
inductive sub (list 'a) (list 'a) =
  | empty: sub (Nil: list 'a) (Nil: list 'a)
  | cons : forall x: 'a, s1 s2: list 'a.
           sub s1 s2 -> sub (Cons x s1) (Cons x s2)
  | dive : forall x: 'a, s1 s2: list 'a.
           sub s1 s2 -> sub s1 (Cons x s2)
```

Standard positivity restrictions apply to ensure the existence of a least fixed point.

Terms and Formulas. First-order language is extended, both in terms and formulas, with pattern matching, `let`-expressions, and conditional (`if-then-else`) expressions. We have decided to be faithful to the usual distinction between terms and formulas that is made in the first-order logic. Thus we make a difference between a predicate symbol and a function symbol which returns a `bool`-typed value, `bool` being defined with `type bool = True | False`. However, to facilitate writing, conditional expressions are allowed in terms, as in the following definition of absolute value:

```
function abs (x: int) : int = if x >= 0 then x else -x
```

Such a construct is directly accepted by provers not making a distinction between terms and formulas (*e.g.* provers supporting the SMT-LIB V2 format [3]). In order to translate `if-then-else` constructs to traditional first-order language, Why3 lifts them to the level of formulas and rewrites them as conjunctions of two implications.

3 Theories

Why3 input is organized as a list of *theories*. A theory is a list of *declarations*. Declarations introduce new types, functions and predicates, state axioms, lemmas and goals. These declarations can be directly written in the theory or taken from existing theories.

Figure 1 contains an example of Why3 input text, containing four theories. We start with a theory `Order` of partial order, declaring an abstract type `t` and an abstract binary predicate `(<=)`. The next theory, `List`, declares a new

```
theory Order
  type t
  predicate (<=) t t

  axiom le_refl : forall x : t. x <= x
  axiom le_asym : forall x y : t. x <= y -> y <= x -> x = y
  axiom le_trans: forall x y z : t. x <= y -> y <= z -> x <= z
end

theory List
  type list ’a = Nil | Cons ’a (list ’a)

  predicate mem (x: ’a) (l: list ’a) = match l with
    | Nil -> false
    | Cons y r -> x = y \/ mem x r
  end
end

theory SortedList
  use import List
  clone import Order as O

  inductive sorted (l : list t) =
    | sorted_nil :
        sorted Nil
    | sorted_one :
        forall x:t. sorted (Cons x Nil)
    | sorted_two :
        forall x y : t, l : list t.
        x <= y -> sorted (Cons y l) -> sorted (Cons x (Cons y l))

  lemma sorted_mem:
    forall x: t, l: list t. sorted (Cons x l) ->
    forall y: t. mem y l -> x <= y
end

theory SortedIntList
  use import int.Int
  use import List
  clone import SortedList with type O.t = int, predicate O.(<=) = (<=)

  goal sorted123: sorted (Cons 1 (Cons 2 (Cons 3 Nil)))
end
```

Fig. 1. Example of Why3 text.

algebraic type of polymorphic lists, `list 'a`, together with a recursively defined predicate of membership.

Now we want to construct a theory `SortedList` of ordered lists. We want to reuse the definition of polymorphic lists given in theory `List`, as well as the axioms from `Order`. The `use import List` command indicates that this new theory may refer to symbols from theory `List`. These symbols are accessible in a qualified form, such as `List.list` or `List.Cons`. The `import` qualifier additionally allows us to use them without qualification. Then we `clone` theory `Order`. This is pretty much equivalent to copy-pasting every declaration from `Order` to `SortedList`. Finally, we introduce an inductive predicate `sorted` and state as a lemma that the head of a sorted list is smaller or equal to all subsequent elements.

Notice an important difference between `use` and `clone`. If we `use` a theory, say `List`, twice (directly or indirectly), there is no duplication: there is still only one type of lists and a unique pair of constructors. On the contrary, when we `clone` a theory, we create a local copy of every cloned declaration, and the newly created symbols, despite having the same names, are different from their originals.

Now, we can instantiate theory `SortedList` to any ordered type, without having to retype the definition of `sorted`. Let us build a theory `SortedIntList` for sorted lists of integers. We first import the theory of integers `int.Int` from Why3's standard library — the prefix `int` indicates the file in the standard library containing theory `Int`. The next declaration clones `SortedList` (*i.e.* copies its declarations) substituting type `int` for type `O.t` of `SortedList` and the default order on integers for predicate `O.(<=)`. Why3 controls that the result of cloning is well-typed. Notice that, when we instantiate an abstract symbol, its declaration is not copied from the theory being cloned. Thus, we do not create a second declaration of type `int` in `SortedIntList`.

Why should we clone theory `Order` in `SortedList` if we make no instantiation? Couldn't we write `use import Order as O` instead? The answer is no. When we `use` a theory, we mean to share its symbols and declarations with other places where this theory is used. On the other hand, when we `clone` a theory, we obtain a fresh, local copy of its declarations. Therefore, when cloning a theory, we can only instantiate the symbols declared locally in this theory, not the symbols imported with `use`. Therefore, we create a local copy of `Order` in `SortedList` to be able to instantiate `t` and `(<=)` later.

The mechanism of cloning bears some resemblance to modules and functors of ML-like languages. Unlike those languages, Why3 makes no distinction between modules and module signatures, modules and functors. Any Why3 theory can be `use`'d directly or instantiated in any of its abstract symbols.

4 Proof Tasks

The principal activity of Why3 can be described as processing of *proof tasks*. A proof task is basically a sequent: a list of declarations that ends with a goal. A

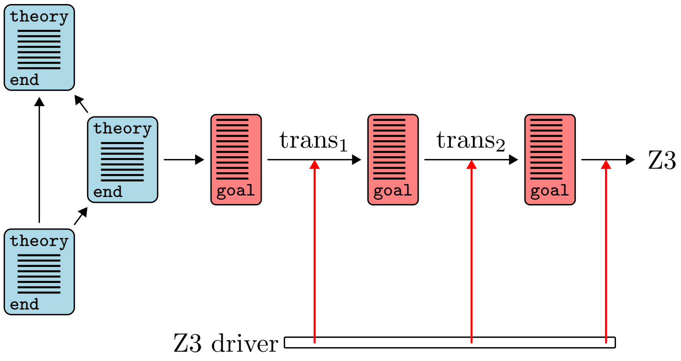


Fig. 2. Task flow in Why3.

proof task is flat: it does not contains any `use` or `clone` anymore. Why3 starts by extracting a set of proof tasks from a given theory.

Suppose we want to send a proof task to a particular prover, say Z3 [13]. Not only is the input syntax of Z3 different from Why3's syntax, there are also significant differences in the logic of the two systems. For instance, Z3 does not support polymorphism or inductive predicates. We need to apply a series of *transformations* that will gradually translate Why3's logic into the prover's logic. This series of transformations is controlled by a configuration file, called a *driver*, associated to any prover supported by Why3. The task flow from theories to provers is illustrated in Figure 2.

Figure 3 contains a simplified driver for Z3. In the driver, we specify a pretty-printer corresponding to the prover's input format (`smtv2` here). We also give regular expressions to interpret the prover output. Next we enumerate the transformations to be applied to a proof task before it can be sent to the pretty-printer. For instance, `inline_trivial` expands "simple" definitions, such as

```
  predicate (>=) (x y : t) = y <= x
```

In the current implementation, we call a definition simple whenever it is non-recursive, right linear and does not contain variables at depth more than one. This might change in future versions of Why3. Transformation `eliminate_algebraic_smt` encodes algebraic data types and pattern-matching expressions in terms of uninterpreted type and function symbols [24]. Finally, `encoding_smt` eliminates polymorphic types from the proof task, converting it to an equivalent monomorphic many-sorted sequent [9]. For description of other transformations, we refer the reader to Why3's manual [8]. Finally, to take into account built-in theories of Z3, we specify the correspondence between Why3 symbols and Z3 interpreted symbols. For instance, integer addition (defined in theory `int.Int`) corresponds to the built-in operation `+` of Z3. Also, we can omit all axioms which are already known to Z3.

```
printer "smtv2"
filename "%f-%t-%g.smt"

valid "^unsat"
invalid "^sat"
unknown "^\\(unknown\\|Fail\\)" "Unknown"
time "why3cpulimit time : %s s"

transformation "inline_trivial"
transformation "eliminate_builtin"
transformation "eliminate_definition"
transformation "eliminate_inductive"
transformation "eliminate_algebraic_smt"
transformation "simplify_formula"
transformation "discriminate"
transformation "encoding_smt"

prelude "(set-logic AUFNIRA)"

theory BuiltIn
  syntax type int   "Int"
  syntax type real  "Real"
  syntax predicate  (=)  "(= %1 %2)"
end

theory int.Int
  prelude ";;; this is a prelude for Z3 integer arithmetic"

  syntax function  zero "0"
  syntax function  one  "1"
  syntax function  (+)  "(+ %1 %2)"
  syntax function  (-)  "(- %1 %2)"
  syntax function  (*)  "(* %1 %2)"
  syntax function  (-_) "(- %1)"
  syntax predicate (<=) "(<= %1 %2)"
  syntax predicate (<)  "(< %1 %2)"
  syntax predicate (>=) "(>= %1 %2)"
  syntax predicate (>)  "(> %1 %2)"

  remove prop CommutativeGroup.Comm.Comm
  remove prop CommutativeGroup.Assoc.Assoc
  remove prop CommutativeGroup.Unit_def
  remove prop CommutativeGroup.Inv_def
  (* etc. *)
end
```

Fig. 3. Driver for Z3.

Users can develop pretty-printers and transformations of their own, dynamically linked to Why3 as plug-ins. They are registered under unique names, which can be subsequently referred to in drivers. As a consequence, a user can easily add support for a new prover or tweak the interface to an existing one. For example, along with the driver in Figure 3, we provide an alternative driver for Z3 with support for built-in theory of arrays. To avoid writing the same driver rules several times, common parts can be put in separate files and included in drivers.

5 Architecture

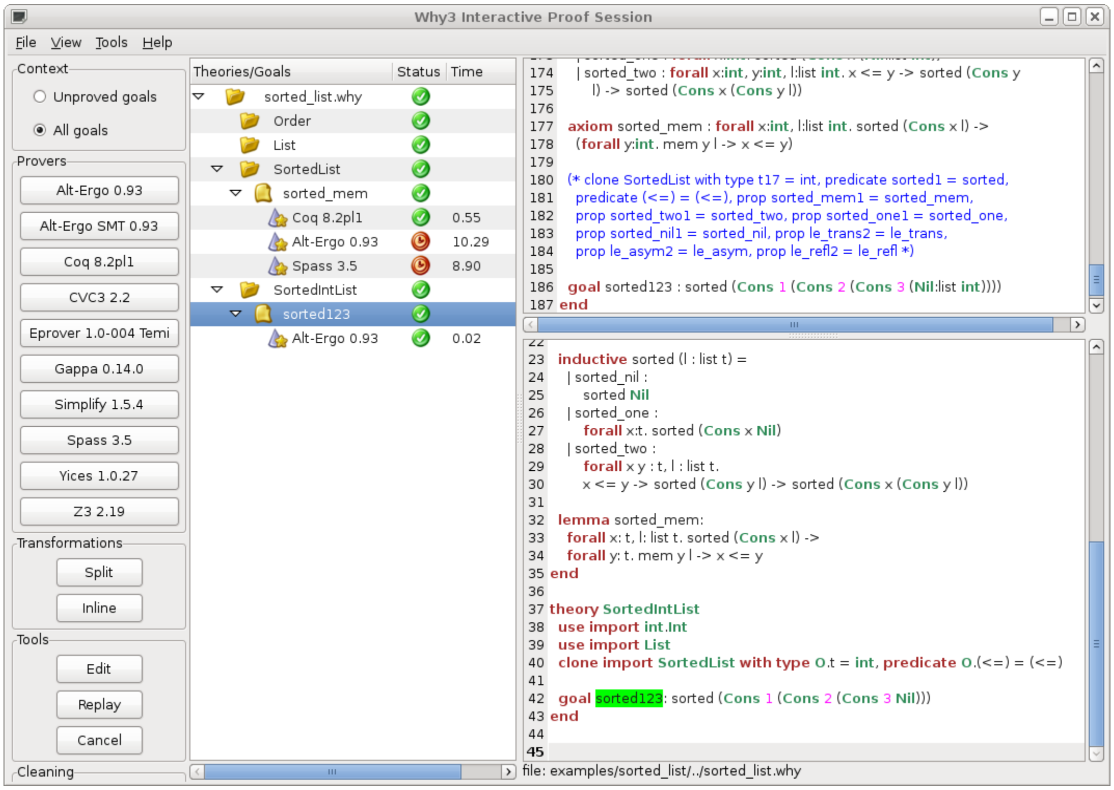
Why3 is implemented as a OCaml programming library. Every functionality (term construction, parsing, proof task transformations, prover calls, etc.) is given in a form of an API. We took a defensive approach in designing this API: Why3 does not allow constructing an ill-formed or ill-typed term, or to use a non-declared symbol in a theory or a proof task. A special effort is made to share the common sub-terms and sub-tasks, and to memoize the intermediate results of transformations on these sub-tasks. In this way, we avoid a lot of redundant work since, in the most common case where proof tasks originate from the same theory, they share the most of their premises.

The tools we provide in Why3 distribution are built on top of this common library. We anticipate other projects to make use of this library. For instance, integrating automated theorem provers in an interactive proof assistant can be naturally done by linking with Why3 (assuming we trust the prover answers). Another way of using Why3 is to supply new parsers, transformations, or pretty-printers in the form of dynamically loadable plug-ins. As an example, we distribute a parser for the TPTP format, allowing us to test Why3 on a vast collection of theorem proving problems [28].

We package three main tools with Why3:

– a simple command-line interface `why3`, to launch a selected prover on a set of goals in a given file;
– an interactive graphical user-interface `why3ide`;
– a tool `why3bench` to benchmark different automated provers (or different configurations of the same prover) on large sets of problems. It is also useful to compare axiomatizations or transformations.

A screenshot of the Why3 GUI is shown in Figure 4. On the left side we see the available provers. We can apply some transformations to a proof task before sending it to the prover; for example, split a goal or unfold a definition in it. The goal in theory `SortedIntList` is quite simple and Alt-Ergo [7] proves it in an instant. The lemma `sorted_mem` in theory `SortedList` is, on the other hand, more difficult for automated prover since it requires induction. We thus resort to an interactive proof assistant, namely Coq, to discharge this proof task. Using the “Edit” button, the user can launch a Coq IDE to edit a proof script. After



Theories/Goals	Status	Time
sorted_list.why	✓	
Order	✓	
List	✓	
SortedList	✓	
sorted_mem	✓	
Coq 8.2pl1	✓	0.55
Alt-Ergo 0.93	⏱	10.29
Spass 3.5	⏱	8.90
SortedIntList	✓	
sorted123	✓	
Alt-Ergo 0.93	✓	0.02

```
  | sorted_two : forall x:int, y:int, l:list int. x <= y -> sorted (Cons y
    l) -> sorted (Cons x (Cons y l))

 axiom sorted_mem : forall x:int, l:list int. sorted (Cons x l) ->
  (forall y:int. mem y l -> x <= y)

 (* clone SortedList with type t17 = int, predicate sorted1 = sorted,
  predicate (<=) = (<=), prop sorted_mem1 = sorted_mem,
  prop sorted_two1 = sorted_two, prop sorted_one1 = sorted_one,
  prop sorted_nil1 = sorted_nil, prop le_trans2 = le_trans,
  prop le_asym2 = le_asym, prop le_refl2 = le_refl *)

 goal sorted123 : sorted (Cons 1 (Cons 2 (Cons 3 (Nil:list int))))
end
```

```
 inductive sorted (l : list t) =
  | sorted_nil :
     sorted Nil
  | sorted_one :
     forall x:t. sorted (Cons x Nil)
  | sorted_two :
     forall x y : t, l : list t.
     x <= y -> sorted (Cons y l) -> sorted (Cons x (Cons y l))

 lemma sorted_mem:
  forall x: t, l: list t. sorted (Cons x l) ->
  forall y: t. mem y l -> x <= y
end

theory SortedIntList
 use import int.Int
 use import List
 clone import SortedList with type O.t = int, predicate O.(<=) = (<=)

 goal sorted123: sorted (Cons 1 (Cons 2 (Cons 3 Nil)))
end
```

file: examples/sorted_list/../sorted_list.why

Fig. 4. Why3 GUI screenshot.

the editing session is finished, Why3 GUI rechecks that the saved proof script is accepted by Coq.

Why3 GUI saves the state of a proof session in a database file. A user can modify the initial Why3 file and then return to GUI and replay the previous proof session. For all proof tasks that have not been automatically discharged in this way, the user has to reconstruct the proof. In the future, we plan to extend Why3 GUI to a full-fledged IDE.

6 Related Work and Perspectives

We presented Why3, a language for specification and a tool to translate it into proof obligations for various interactive and automated theorem provers. Why3 improves upon the former Why 2 platform both in terms of expressiveness, architecture, extensibility (see the manual [8] for an exhaustive list of changes). The Why3 platform can be used by itself, as some kind of standalone "meta" theorem prover, but the main purpose of Why3 is to be used as an intermediate language. For instance, we are currently designing a plug-in for the Coq proof assistant, to extract the first-order part of Coq goals and pass them to automated theorem provers. At the moment, Why3 does not attempt to analyze the output of an automated prover (neither proofs nor counterexamples). A short term perspective would be to translate counterexamples back to Why3 language. This requires not only to parse the output of a particular prover, but also to reverse the effect of various task transformations of Why3. In a long term perspective, we would like to augment the confidence in prover results by producing Coq or Isabelle certificates, in the spirit of Isabelle's Sledgehammer [22].

At the same time, we are re-implementing a verification condition generator for a programming language, WhyML, annotated with Why3 pre/post-conditions. (Interested readers are invited to visit our gallery of verified programs at `http://proval.lri.fr/gallery/why3.en.html`.) We also plan to use WhyML as an intermediate language for program verification, in the spirit of the former Why platform [14] or Boogie [1]. This is one of the two principal approaches to program verification, where a general-purpose programming language is equipped with a specification language and then possibly translated to the intermediate language of a VCGen. This is the case of VCC [12], Spec# [2], Dafny [19], Chalice [20], ESC/Java2 [11], Krakatoa [21], Frama-C [15]. The other principal approach uses a deep embedding of a programming language and its semantics in the logic of a general-purpose proof assistant. This is the case of SunRise [16], KIV [26], Isabelle/Simpl [27], KeY [5], Ynot [23], etc.

While the latter approach benefits from rich specification languages, one can argue that these languages are not close enough to the programming language constructs, creating another entry barrier for a new user. Additionally, proofs are typically performed in an interactive way, since underlying environments do not offer as much automation as state-of-the-art theorem provers (although the Sledgehammer effort strives to fill this gap). On the other hand, the former approach, which Why3 belongs to, offers more automation but poorer specification languages. We believe that Why3 features presented in this paper (inductive predicates, algebraic data types, theories) help to alleviate this drawback. We intend to promote these constructions to existing specification languages, such as ACSL [4] or JML [10].

References

1. M. Barnett, R. DeLine, B. Jacobs, B.-Y. E. Chang, and K. R. M. Leino. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W.-P. de Roever, editors, *Formal Methods for Components and Objects: 4th International Symposium*, volume 4111 of *Lecture Notes in Computer Science*, pages 364–387, 2005.
2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.

3. C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. In A. Gupta and D. Kroening, editors, *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, 2010.
4. P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language, version 1.4*, 2009. `http://frama-c.cea.fr/acsl.html`.
5. 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.
6. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions.* Springer, 2004.
7. 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/`.
8. F. Bobot, J.-C. Filliâtre, C. Marché, and A. Paskevich. *The Why3 platform.* LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.64 edition, Feb. 2011. `http://why3.lri.fr/`.
9. F. Bobot and A. Paskevich. Expressing Polymorphic Types in a Many-Sorted Language, 2011. Preliminary report. `http://hal.inria.fr/inria-00591414/`.
10. L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.
11. D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *CASSIS*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer, 2004.
12. M. Dahlweid, M. Moskal, T. Santen, S. Tobies, and W. Schulte. VCC: Contract-based modular verification of concurrent C. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Companion Volume*, pages 429–430. IEEE Comp. Soc. Press, 2009.
13. L. de Moura and N. Bjørner. Z3, an efficient SMT solver. `http://research.microsoft.com/projects/z3/`.
14. J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, volume 4590 of *Lecture Notes in Computer Science*, pages 173–177, Berlin, Germany, July 2007. Springer.
15. The Frama-C platform for static analysis of C programs, 2008. `http://www.frama-c.cea.fr/`.
16. P. V. Homeier and D. F. Martin. A mechanically verified verification condition generator. *The Computer Journal*, 38(2):131–141, July 1995.
17. The ISABELLE system. `http://isabelle.in.tum.de/`.
18. G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. *Commun. ACM*, 53(6):107–115, June 2010.
19. K. R. M. Leino. Dafny: An Automatic Program Verifier for Functional Correctness. In Springer, editor, *LPAR-16*, volume 6355, pages 348–370, 2010.
20. K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In A. Aldini, G. Barthe, and R. Gorrieri, editors, *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*, volume 5705 of *Lecture Notes in Computer Science*, pages 195–222. Springer, 2009.

21. C. Marché. The Krakatoa tool for deductive verification of Java programs. Winter School on Object-Oriented Verification, Viinistu, Estonia, Jan. 2009. `http://krakatoa.lri.fr/ws/`.
22. J. Meng, C. Quigley, and L. C. Paulson. Automation for interactive proof: first prototype. *Inf. Comput.*, 204(10):1575–1596, 2006.
23. A. Nanevski, G. Morrisett, A. Shinnar, P. Govereau, and L. Birkedal. Ynot: Reasoning with the awkward squad. In *Proceedings of ICFP'08*, 2008.
24. A. Paskevich. Algebraic types and pattern matching in the logical language of the Why verification platform (version 2). Technical Report 7128, INRIA, 2010. `http://hal.inria.fr/inria-00439232/en/`.
25. The PVS system. `http://pvs.csl.sri.com/`.
26. W. Reif, G. Schnellhorn, and K. Stenzel. Proving system correctness with KIV 3.0. In W. McCune, editor, *14th International Conference on Automated Deduction*, Lecture Notes in Computer Science, pages 69–72, Townsville, North Queensland, Australia, july 1997. Springer.
27. N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL*. PhD thesis, Technische Universität München, 2006.
28. G. Sutcliffe, C. Suttner, and T. Yemenis. The TPTP problem library. In A. Bundy, editor, *Proc. 12th Conference on Automated Deduction CADE, Nancy/France*, pages 252–266. Springer-Verlag, 1994.