Int J Softw Tools Technol Transfer (2011) 13:397–403
DOI 10.1007/s10009-011-0211-0

INTRODUCTION

Deductive software verification

Jean-Christophe Filliâtre

Published online: 20 August 2011

Abstract Deductive software verification, also known as program proving, expresses the correctness of a program as a set of mathematical statements, called verification conditions. They are then discharged using either automated or interactive theorem provers. We briefly review this research area, with an emphasis on tools.

Keywords Software verification · Specification language · Hoare logic · Proof assistant · Automated theorem prover

1 Introduction

Deductive software verification is the process of turning the correctness of a program into a mathematical statement and then proving it. The idea is not new. It is actually as old as programming. One of the very first proof of program was performed by Alan Turing in 1949. His three pages paper *Checking a large routine* (Morris and Jones in IEEE Ann Hist Comput 6(2):139–143, 1984; Turing in Checking a large routine. In: Report of a Conference on High Speed Automatic Calculing Machines. Mathematical Laboratory, Cambridge, pp 67–69, 1949) contains a rigorous proof a of program computing the factorial by repeated additions. Mathematical assertions about variables (storage locations, actually) are associated with the various points of the code flowchart. The paper even includes a proof of termination. Another example is Hoare's *Proof of a program: FIND* (Commun ACM 14:39–45, 1971). In this 1971 paper, Hoare shows the correctness of a program that rearranges the elements of an array so that the kth element ends up at position k, with smaller elements to the left and greater elements to the right. Hoare exhibits the program annotations and carefully states and proves the 18 lemmas that entail their validity. Later, a mechanized proof of this program (Filliâtre in Sci Comput Program 64:332–340, 2006) following Hoare's assertions showed that this proof did not contain a single error. On a theoretical point of view, the method applies to any code and any specification, the limit only being the mathematical skills of the user. In practice, however, this is not that simple. Proving a program requires to check the validity of many conditions, most of them being boring and error-prone statements. Fortunately, the situation has evolved a lot in the last two decades. Languages and tools have flourished, which allow the mechanization of the verification process. In the following, we give a quick overview of the difficulties inherent to deductive software verification and of the techniques and tools that are tackling them.

2 Challenges

The correctness of a program is not uniquely defined. It obviously depends on a given specification, being it informal or written in a formal language. A specification can be limited to the mere *safety*: the program will execute without ever running into a fatal error, such as dividing by zero or accessing an invalid memory location. Proving safety can be already quite challenging. A few years ago, a bug was found in the Java standard library which was related to the computation of the average of two integers in some binary search [8]. With array indices `l` and `u` large enough, the computation of `(l + u)/2` may return a negative value if the addition overflows,

J.-C. Filliâtre
CNRS, Gif-sur-Yvette, France

J.-C. Filliâtre (✉)
LRI/Univ Paris-Sud 11, 91405 Orsay, France
e-mail: Jean-Christophe.Filliatre@lri.fr

J.-C. Filliâtre
INRIA Saclay–Île-de-France, 91893 Orsay, France

resulting in some fatal access out of array bounds[1]. This sole example shows that proving safety may require to prove the absence of arithmetic overflows, which in turn may require a lot of other properties to be checked.

Another example is termination. Most of the time, termination is a desired property and should naturally be associated with safety. It is worth pointing out that early papers on program verification, in particular the ones cited above [25,52], already included proving termination as part of the proof process. In principle, proving termination is easy: each loop or recursive function is associated with some entity, called a *variant*, which decreases at each step and cannot decrease indefinitely [25]. Typically, a natural number will do. But exactly as for safety above, proving termination can be challenging anyway. A historical example is McCarthy's "91 function" [38]. It is a function over integers defined as follows:

$$f(n) = \begin{cases} f(f(n+11)) & \text{if } n \leq 100, \\ n - 10 & \text{otherwise.} \end{cases} \tag{1}$$

Figuring out a variant for this function is not that difficult: it is as simple as $101 - n$. But proving that it is indeed a variant, that is proving that it is non-negative and decreases for each recursive call, is another matter. Indeed, it requires to establish a postcondition for the function, relating its output value to its parameter n (namely, that $f(n)$ returns 91 whenever $n \leq 100$, and $n - 10$ otherwise). As for binary search above, proving a property as simple as termination actually requires us to prove much more about the program.

Another example of program whose sole termination is difficult to establish is Floyd's cycle detection algorithm, also known as "tortoise and hare" algorithm [33, ex. 6 p. 7]. To check whether a singly linked list contains a cycle, this algorithm traverses it at speeds 1 and 2 at the same time, using only two pointers. Whenever a pointer reaches the end of the list, the outcome is negative. Whenever the two pointers are equal, the outcome is positive. The beauty of this algorithm is that it necessarily terminates, the hare not being able to overtake the tortoise within the cycle, if any. Turning that into a proof is non-trivial. One has to state the finiteness of the memory cells of the list, to introduce the inductive notion of path from one list cell to another, and finally to come up with a variant which is quite involved.

As a last example, let us consider the verification of a sorting algorithm. The algorithmic literature contains so many sorting algorithms that it is natural to consider proving them as well, at least, as an exercise. Expressing that a routine sorting an array ends up with the array being sorted is quite easy. It amounts to a postcondition looking like

$$\forall ij,\ 0 \leq i \leq j < n \Rightarrow a[i] \leq a[j]. \tag{2}$$

But this is only half of the specification, and that is the easy one. We also need to state that the final contents of the array are a rearrangement of its initial contents. Otherwise a code that simply fills the array with a single value would obviously fulfill the specification above. When it comes to specify the rearrangement property, there are several solutions but none is easy to define. In a higher-order logic, one can state the existence of a bijection. Though elegant, it is not amenable to proof automation. Another solution is to introduce the notion of multiset, to turn the contents of an array into a multiset, and then to state the equality of the multisets for the initial and final contents of the array. A third possibility is to define the notion of permutation as the smallest equivalence relation containing transpositions, which is adapted to sorting algorithms that swap elements [23].

These few verification examples show the necessity of expressive specification languages, even to prove simple properties such as safety. In particular, we cannot expect deductive program verification to be conducted within a decidable logic.

3 Specifications

Roughly speaking, a specification language is made up of two parts: first, a mathematical language in which components of the specification (pre- and postconditions, invariants, etc.) is written, as well as verification conditions *in fine*; second, its integration with a programming language. Obviously, the choice of a specification language is tightened to both the methodology used to generate the verification conditions and the technique used to prove them. These two aspects will be discussed in the next two sections.

As far as the mathematical language is concerned, first-order predicate calculus with equality is a natural choice, immediately picked up without discussion in pioneer work [25, p. 21]. Today, many systems use a flavor of first-order logic [3,5,6,11,17,36]. Beside equality, a handful of built-in theories are typically considered. Linear arithmetic, purely applicative arrays, and bit vectors are the common ones; others such as non-linear or real arithmetic may be available as well. Other extensions to first-order logic include recursive functions and algebraic data types—in Dafny [36] and Why3 [11] for instance—as well as polymorphism [12] and inductive predicates [11]. Set theory is an alternative, considered for instance in the B method [1].

Another route is to consider using the logic of an existing, general-purpose proof assistant. Examples of such systems successfully used in software verification include Coq [50], PVS [43], Isabelle [44], and ACL2 [31]. This approach has obvious benefits. First, it provides a powerful specification language, as these proof assistants typically provide

[1] Interestingly, binary search was precisely taken as an example of program verification in Jon Bentley's *Programming Pearls* [7] more than 25 years ago. Unfortunately, his proof does not consider the possibility of an arithmetic overflow.

a rich and higher-order logic. Second, the logic is already implemented and, in particular, interactive proof tools are ready to be used. Finally, proof assistants come with libraries developed over years, which eases the specification process. But this approach has drawbacks as well. One is that a richer logic comes at a cost: proof automation is typically more difficult to achieve. This is discussed later in Sect. 5.

Finally, one can also introduce a new logic, dedicated for program verification. Undoubtedly the best example is separation logic [45]. Among other things, it allows local reasoning on heap fragments, being a tool of choice for proving programs involving pointer data structures. Similar proofs with traditional first-order predicate calculus require cumbersome and explicit manipulations of set of pointers.

Once the mathematical language for specification and verification conditions is set, we are left with its integration with a programming language. Pioneer works intend to bind logical assertions to the program flowchart [25,52]. A key step was then Hoare's axiomatic basis [27] introducing the concept now known as *Hoare triple*. In modern notation, such a triple $\{P\}s\{Q\}$ binds together a precondition P, a program statement s, and a postcondition Q. Its validity means that execution of s in any state satisfying P must result, on completion, in a state satisfying Q. Requiring s to be terminating defines total correctness, as opposed to partial correctness otherwise. The beauty of Hoare logic is not really that it proposes a methodology to derive the correctness of a program—weakest preconditions are more efficient, as we will see in next section—but that it opens the way to a *modular* reasoning about programs. Software components can be enclosed in Hoare triples, with preconditions that are proof requirements for the caller and, conversely, postconditions that are ensured to be valid by the callee. In some hypothetical language, such a triple for a function f could look like

$$\begin{array}{l}\textbf{requires } P\\ \textbf{ensures } Q\\ \textbf{function } f(x_1, \ldots, x_n)\end{array} \tag{3}$$

where P may refer to the current state and to function parameters x_i, and Q may refer to the state prior to the call, to the final state, to function parameters, and to the returned value, if any. This idea has been popularized by Meyer's *design by contract*, a key feature of the Eiffel programming language [40], and later by specification languages such as JML [34] and its derivatives [3,5,24]. Modern specification languages include some variations, such as the ability to have several contracts for a given function, to distinguish side effects from the postcondition, to allows exceptional behaviors, etc. Anyway, they all clearly derive from Hoare logic.

This is not the only way to go, though. A specification language can be tightened to a programming language within a dedicated *program logic*, which mixes programs and logical statements in a more intricate way. The aforementioned B method is built on top of a notion of generalized substitution, which infuses programming constructs among logical ones within a single language. Similarly, when software verification is conducted within a general-purpose proof assistant, there is only one language, that is the logic language. A program f is simply a purely applicative and terminating function, already part of the logic. Proving it correct amounts to establishing a statement like

$$\forall x,\ P(x) \Rightarrow Q(x, f(x)). \tag{4}$$

The user is then left with a traditional proof or assisted with some "tactic" dedicated to program verification.

So far we have discussed the specification language and its integration with the programming language. We now turn to methodologies to extract verification conditions from a program and its associated specification.

4 Methodologies

Hoare logic is undoubtedly the most famous of all techniques, Hoare's paper [27] being the most cited among pioneer works. It proposes rules to establish the validity of Hoare triples. The rule for assignment, for instance, reads

$$\{P[x \leftarrow E]\}\ x := E\ \{P\}. \tag{5}$$

It assumes E to be an expression without side effect, thus safely shared between program and logic, and also x not to be aliased with another program variable. More generally, each program construct is given a deductive rule. Some additional rule allows to strengthen a precondition and to weaken a postcondition. It is a neat system, from which you can derive the validity of arbitrarily complex programs. In practice, however, one quickly finds it painful to figure out intermediate assertions for Hoare rules to compose nicely. The natural approach is rather to assign meaningful assertions at key places in the code and let the system figure out the intermediate assertions. This is exactly what Dijkstra's calculus of weakest precondition [21] is doing. Given a program statement s and a postcondition Q, it computes a precondition $wp(s, Q)$, recursively over the structure of s, that precisely captures the weakest requirement over the initial state such that the execution of s will end up in a final state satisfying Q. As a consequence, the validity of the Hoare triple $\{P\}s\{Q\}$ is equivalent to

$$P \Rightarrow wp(s, Q). \tag{6}$$

Computing weakest preconditions is the approach taken in most modern verification condition generators. Recent work has even improved over Dijkstra's calculus to provide greater efficiency [35]. There is still the limitation of programs being free of aliasing, though. As a consequence, the semantics of the program under consideration must be encoded as a

set of types, symbols, and axioms, known as the *memory model*, together with an alias-free version of this program performing actions on this memory model. Within the last decade, two intermediate languages appeared, whose sole purpose is to perform the computation of weakest preconditions, independently of the memory model: Why [10,24] and Boogie [2]. Many tools now employ this route which consists in building a memory model and then reusing an existing intermediate language (Krakatoa, SPEC#, VCC, Frama-C, Dafny, among others).

Building memory models is an activity in itself. Some key ideas come from the 70s [14]. They could be successfully applied at the turn of the century when practice finally caught up with theory [13] and then even improved [29]. Other models include ideas from state-of-the-art recent research. An impressive example is the model used in the L4. verified project [32]. This model [51] combines low-level details allowing the verification of high-performance C code with higher-level concepts of separation logic.

We already explained that proof assistants, such as Coq, PVS, Isabelle, or ACL2, are tools of choice to write program specifications, and even to write programs when they happen to be purely applicative. Then, proving a program f correct, that is a statement such as (4), is a matter of following the definition of f, splitting the goal into smaller verification conditions. This can even be automated, using a dedicated tactic (either built-in or developed for the purpose of program verification). The user may have to provide some intermediate assertions, such as a generalized specification for a recursive function. This can be compared to the computation of weakest preconditions. When done, the user is left with a bunch of purely logical statements and all the power of the proof assistant can be exploited to discharge them. An impressive example of such a program verification is Leroy's verified C compiler CompCert [37], developed and proved correct within the Coq proof assistant.

But the use of general-purpose proof assistants is not limited to the verification of purely applicative programs. One can use the logic of a proof assistant to define an imperative programming language and its semantics. Then, a proof methodology can be carried out within the proof assistant and particular programs can be verified. This is called *deep embedding*. A successful example is Schirmer's work in Isabelle/HOL [46]. It provides a Hoare logic-like language with procedures, operating over a polymorphic state space. It was used in the L4. verified project. An additional benefit of deep embedding is that it allows meta-theoretical proofs, such as soundness of the proof method with respect to the semantics of the language. An alternative to deep embedding is *shallow embedding*. Concepts related to programs are mere notations for semantic definitions, which can be tackled by unfolding or, better, by relevant axioms and theorems. An example is Ynot [42], a Coq-embedded system to verify higher-order and effectful programs using a combination of monads, separation logic, and Hoare triples.

When a program is given a specification and processed through a suitable deductive verification method, one is left with a set of mathematical statements, the so-called *verification conditions*. The last step is to prove them.

5 Proofs

The L4. verified and CompCert projects are two examples of large program verification conducted within a proof assistant (Isabelle/HOL and Coq, respectively). At least, they show the relevance of using a general-purpose proof assistant to discharge verification conditions. Even when the latter is computed completely independently, for instance using a weakest precondition calculus, it is still possible to translate them to the native format of a proof assistant. This is always possible, as the logic of the proof assistant is typically richer than the one used to express the verification conditions.

However, there are several reasons why we may want to avoid this path, at least in a first step. One reason is that deductive verification typically generates numerous, yet simple, goals. These include array bounds checking, variant decreasing, absence of arithmetic overflows, non-nullity of pointers, etc. They are better handled directly by automated theorem provers, as discussed below. Processing them within a proof assistant would incur an extra cost, such as one manipulation at least per verification condition, if not more. Another reason to avoid interactive proof in the first step is proof maintenance. In the process of figuring out the right specification, or even the right code, it is likely that verification conditions will change a lot before stabilizing. Any manipulation within the proof assistant, as small as it is, will slow the whole specify/prove cycle.

It is better to turn to automated theorem provers when it comes to the task of discharging verification conditions. Automated theorem provers have a long history, as demonstrated by the CASC competition [48] and the huge corresponding library of problems TPTP [49]. However, all the automated theorem provers taking part in this competition were never really involved in deductive program verification. The main reason must be the lack of support for arithmetic, a must-have in program verification[2]. Independently, another line of automated theorem provers emerged, called *SMT solvers*. Initiated by Shostak's decision procedure [47] in the early 1980s, it focuses on the combination of first-order logic, congruence closure, and built-in theories. The latter typically involves linear arithmetic, at least. One of the earliest prover in that line was Nelson's Simplify [20]. Developed in the 1990s, in the context of program verification, it still

[2] Things may change, as there are rumors that next versions of Vampire, the state-of-the-art TPTP prover, will have support for arithmetic.

performs impressively. Other SMT solvers followed, such as CVC3 [4], Yices [19], Z3 [18], and Alt-Ergo [9]. Today, they have reached such a level of efficiency that a program such as binary search can be proved correct fully automatically within a second. This includes proving safety, termination, absence of arithmetic overflow and full behavioral correctness.

Obviously, we are also left from time to time with verification conditions that cannot be proved automatically. A possibility is then to insert more intermediate assertions in the code, to reduce the cost of each individual deduction. In other words, the user inserts logical *cuts* to ease the proof. If this is unsuccessful, or if this is not even an option, then there is no other choice than turning to a proof assistant and starting an interactive proof. A definitive improvement in this process is the recent introduction of automated theorem provers within proof assistants. Isabelle/HOL is leading the way with its Sledgehammer tool [39].

Finally, another approach is to use a dedicated prover. It can be motivated by a need for a close connection between verification conditions as displayed during an interactive proof process and the input code and specification. Examples include the system KeY [6] and the B method, for instance. Regarding the latter, it may also be motivated by the poor support for set theory in existing automated theorem provers. One obvious drawback of a dedicated prover is that it will hardly benefit from state-of-the-art results in automated or interactive theorem proving, at least not for free.

6 Conclusion

This quick survey of deductive software verification shows the vitality of this domain. Techniques and tools have reached a maturity which allows the verification of complex programs in limited amount of time. The emergence of competitions related to software verification (VERIFYTHIS website, VSTTE and FoVeOOS competitions, etc.) leaves no doubt about that. Let us hope that these competitions will boost deductive software verification, exactly as CASC/SMT competitions boosted the development of automated theorem provers.

We can hardly say that deductive software verification scales though. There are few examples of large verified programs, such as the aforementioned L4. verified and CompCert projects. But, generally speaking, deductive software verification is still a time-consuming activity requiring experts. Perspectives for improvement include large libraries of verified components and support for even more theories in automated theorem provers, among other things.

Deductive software verification is only one aspect of formal methods. The whole area is showing an intense activity, exemplified by the VSTTE 2009 post-proceedings which we now introduce.

Verified software: theory, tools, and experiments (VSTTE) 2009 post-proceedings

The second edition of the workshop *Verified Software: Theory, Tools, and Experiments* was held in Eindhoven, The Netherlands on November 2, 2009, as part of the *Formal Methods Week FM2009*[3]. The focus of this workshop was on *tools*. This special issue gathers five papers resulting from a post-workshop selection. We briefly introduce them.

As we explained earlier in this paper, formal method tools were long awaited and it is only in the last 20 years that we saw a true emergence of *tools*. Nowadays, it is almost the opposite. There are so many techniques and tools related to formal methods that one can quickly get lost. In the five papers of this special issue [15,16,26,30,53], I could enumerate not less than 65 different tools and languages being cited. At least, half of them were indeed used by the authors, not mentioning the design and implementation of new tools [16,30] or languages [53]. This shows the extraordinary vitality of formal methods. In this wide soup of tools, some natural selection process is taking place: tools are created, modified, abandoned, combined into new ones, etc. It is hard to predict what will emerge—and I will not try. But with no doubt I can say that this special issue is representative of the creativity and high quality of today's formal methods.

In *Functional Dependencies of C Functions via Weakest Pre-Conditions* [16], the authors consider the problem of expressing and checking how a C function interferes with its environment when it is called. The function's memory footprint, that is the set of memory locations possibly modified by the call, is made part of the specification. Additionally, the specification expresses *dependencies*: for each memory location in the footprint, the set of memory locations which influence its final value is also made explicit. The novelty of this work is to consider a non-trivial fragment of C with full pointer aliasing. To do so, the authors devise weakest preconditions for functional dependencies and a suitable notion of loop invariants. This is implemented in the Frama-C toolbox for analysis of C programs.

In *SMT Solvers: New Oracles for the HOL Theorem Prover* [53], the author is integrating SMT solvers in the HOL4 proof assistant. The purpose is to bring state-of-the-art automated theorem provers to HOL4 users, relieving them from tedious proofs involving a combination of propositional reasoning, congruence closure and a few decidable theories supported by SMT solvers such as linear arithmetic, tuples, bit vectors, etc. Though the author is mostly focusing on the SMT solver Yices, to make the best possible use of its native input language, he is also integrating all SMT solvers

[3] VSTTE was organized as a full conference in 2006 and 2008, and as a workshop in 2006, 2009 and 2010. VSTTE 2012 will be a conference, co-located with POPL.

supporting the SMT-LIB input format (Z3, CVC3, Alt-Ergo, etc.). This paper is only using the SMT solvers as trusted oracles, but the author already started working on proof reconstruction. There is no doubt that a carefully crafted integration of automated theorem provers into proof assistants will change our way of doing proofs.

Scenario-Based Testing from UML/OCL Behavioral Models—Application to POSIX Compliance [15] is about generating test cases for a file system manager written in UML/OCL, as part of the POSIX mini-challenge. The authors identify the limitations of fully automated test generation and propose instead a scenario-based testing approach. They design a language for that purpose, where sequences of operations are described using regular expressions. Experiments are conducted in a plug-in for the Hydra platform and consider two different implementations of the POSIX file system. Results show a definitive improvement over fully automated test generation.

An Interval-based SAT Modulo ODE Solver for Model Checking Nonlinear Hybrid Systems [30] is introducing `hydlogic`, a new bounded model checker for hybrid systems. Such systems combine discrete and continuous models, to describe programs which interact with physical environments. An example provided by the authors is a controller steering a car on a road along a canal. To verify such systems, the authors are using the framework of satisfiability modulo theories (SMT) to combine a SAT solver and an interval-based solver for hybrid constraint systems. Their implementation, `hydlogic`, outperforms existing tools such as PHAVer and HSolver on several examples, showing the benefits of this new approach. It is a nice example of a technique, namely satisfiability modulo theories, being successfully transposed into a different domain than the one it was originally designed for.

In *Formal Methods for Security in the Xenon Hypervisor* [26], the authors are reporting on the use of Circus to give a formal specification to the Xenon hypervisor. In particular, it means defining security and specifying the hypercall interface. The challenge is to devise a notion of information flow security which is preserved by refinement, which is achieved in this paper. The development is truly impressive (4,500 pages of Circus) and shows that formal methods do scale up when they are smartly used.

I hope you will enjoy the five papers from this special issue as much as I did.

References

1. Abrial, J.-R.: The B-Book, Assigning Programs to Meaning. Cambridge University Press, Cambridge (1996)
2. Barnett, M., DeLine, R., Jacobs, B., Evan Chang, B.-Y., Leino, K.R.M.: Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In: de Boer F.S., Bonsangue, M.M., Graf S., de Roever W.-P. (eds.) Formal Methods for Components and Objects: 4th International Symposium. Lecture Notes in Computer Science, vol. 4111, pp. 364–387 (2005)
3. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# Programming System: An Overview. pp. 49–69. Springer, Berlin (2004)
4. Barrett, C., Tinelli, C.: CVC3. In: Damm, W., Hermanns, H. (eds.) Proceedings of the 19th International Conference on Computer Aided Verification (CAV'07), Berlin, Germany. Lecture Notes in Computer Science. Springer, Berlin (2007)
5. Baudin, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, version 1.4 (2009). http://frama-c.cea.fr/acsl.html
6. Beckert, B., Hähnle, R. P.H. Schmitt (eds.) Verification of Object-Oriented Software: The KeY Approach. Lecture Notes in Computer Science, vol. 4334. Springer, Berlin (2007)
7. Bentley, J.L.: Programming Pearls. Addison-Wesley, Reading (1986)
8. Bloch, J.: Nearly all binary searches and mergesorts are broken (2006). http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html
9. Bobot, F., Conchon, S., Contejean, Iguernelala, E., Lescuyer, M., Mebsout, S., Alain, M.: The Alt-Ergo automated theorem prover (2008). http://alt-ergo.lri.fr/
10. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, A.: The Why3 platform. LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.64 edn, February (2011). http://why3.lri.fr/
11. Bobot, F., Filliâtre, J.-C., Marché, C., Paskevich, Andrei: Why3: Shepherd your herd of provers. In: Boogie 2011: First International Workshop on Intermediate Verification Languages. Wrocław, Poland, August (2011)
12. Bobot, F., Paskevich, A.: Expressing Polymorphic Types in a Many-Sorted Language (2011). Preliminary report. http://hal.inria.fr/inria-00591414/
13. Bornat, R.: Proving pointer programs in Hoare logic. In: Mathematics of Program Construction, pp 102–126 (2000)
14. Burstall, R.: Some techniques for proving correctness of programs which alter data structures. Mach. Intell. **7**, 23–50 (1972)
15. Castillos, K.C., Dadeau, F., Julliand, J.: Scenario-Based Testing from UML/OCL Behavioral Models—Application to POSIX Compliance. Special Section on VSTTE (2009). doi:10.1007/s10009-011-0189-7
16. Cuoq, P., Monate, B., Pacalet, A., Prevosto, V.: Functional Dependencies of C Functions via Weakest Pre-Conditions. Special Section on VSTTE (2009). doi:10.1007/s10009-011-0192-z
17. Dahlweid, M., Moskal, M., Santen, T., Tobies, S., Schulte W.: 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, pp 429–430. IEEE Comp. Soc. Press, New York (2009)
18. de Moura, L., Bjørner, N.: Z3, an efficient SMT solver. http://research.microsoft.com/projects/z3/
19. de Moura, L., Dutertre, B.: Yices: An SMT Solver. http://yices.csl.sri.com/
20. Detlefs, D., Nelson, G., Saxe, J.B.: Simplify: a theorem prover for program checking. J. ACM **52**(3), 365–473 (2005)
21. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**, 453–457 (1975)
22. Filliâtre, J.-C.: Formal Proof of a Program: Find. Sci. Comput. Program. **64**, 332–340 (2006)
23. Filliâtre, J.-C.: Magaud Nicolas Certification of Sorting Algorithms in the System Coq. In: Theorem Proving in Higher Order Logics: Emerging Trends. Nice, France (1999)
24. Filliâtre, J.-C., Marché, C.: The Why/Krakatoa/Caduceus platform for deductive program verification. In: Damm, W., Hermanns, H., (eds.) 19th International Conference on Computer Aided Verifi-

cation. Lecture Notes in Computer Science, vol. 4590. Springer, Berlin, pp. 173–177 (2007)
25. Floyd, R.W.: Assigning meanings to programs. In: Schwartz, J.T. (ed.) Mathematical Aspects of Computer Science. Proceedings of Symposia in Applied Mathematics, vol. 19. American Mathematical Society, Providence, Rhode Island, pp. 19–32 (1967)
26. Freitas, L., Mcdermott, J., Woodcock, J.: Formal Methods for Security in the Xenon Hypervisor. Special Section on VSTTE (2009). doi:10.1007/s10009-011-0195-9
27. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10):576–580, 583 (1969)
28. Hoare, C.A.R.: Proof of a program: Find. Commun. ACM **14**, 39–45 (1971)
29. Hubert, T., Marché, C.: Separation analysis for deductive verification. In: Heap Analysis and Verification (HAV'07). Braga, Portugal, pp. 81–93 (2007)
30. Ishii, D., Ueda, K., Hosobe, H.: An Interval-based SAT Modulo ODE Solver for Model Checking Nonlinear Hybrid Systems. Special Section on VSTTE (2009). doi:10.1007/s10009-011-0193-y
31. Kaufmann, M., Moore, J.S., Manolios, P.: Computer-Aided Reasoning: An Approach. Kluwer Academic Publishers, Norwell (2000)
32. Klein, G., Andronick, J., Elphinstone, K., Heiser, G., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: Formal verification of an OS kernel. Commun. ACM **53**(6), 107–115 (2010)
33. Knuth, D.E.: The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms. Addison-Wesley Longman Publishing Co., Inc., Reading (1997)
34. Leavens, G.T., Leino, K.R.M., Poll, E., Ruby, C., Jacobs, B.: JML: notations and tools supporting detailed design in Java. In: OOPSLA 2000 Companion, Minneapolis, Minnesota, pp. 105–106 (2000)
35. Leino, K.R.M.: Efficient weakest preconditions. Inform. Process. Lett. **93**(6), 281–288 (2005)
36. Leino, K.R.M.: Dafny: An Automatic Program Verifier for Functional Correctness. In: Springer, editor, LPAR-16, vol. 6355, pp. 348–370 (2010)
37. Leroy, X.: A formally verified compiler back-end. J. Autom. Reason. **43**(4), 363–446 (2009)
38. Manna, Z., Mccarthy, J.: Properties of programs and partial function logic. In: Machine Intelligence, Edinburgh Uni. Press, vol. 5. Edinburgh (1970)
39. Meng, J., Paulson, L.: Translating higher-order clauses to first-order clauses. J. Autom. Reason. **40**, 35–60 (2008)
40. Meyer, B.: Eiffel: The Language. Prentice Hall, Hemel Hempstead (1992)
41. Morris, F.L., Jones, C.B.: An early program proof by alan turing. IEEE Ann. Hist. Comput. **6**(2), 139–143 (1984)
42. Nanevski, A., Morrisett, G., Shinnar, A., Govereau, P., Birkedal, L.: Ynot: Reasoning with the awkward squad. In: Proceedings of ICFP'08 (2008)
43. Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Language Reference. Computer Science Laboratory, SRI International, Menlo Park, CA (1999)
44. Paulson, L.C.: Introduction to isabelle. Technical report, University of Cambridge (1993)
45. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: 17h Annual IEEE Symposium on Logic in Computer Science. IEEE Comp. Soc. Press, New York (2002)
46. Schirmer, N.: Verification of Sequential Imperative Programs in Isabelle/HOL. PhD thesis, Technische Universität München (2006)
47. Shostak, R.E.: Deciding combinations of theories. J. ACM **31**, 1–12 (1984)
48. Sutcliffe, G., Suttner, C.: The State of CASC. AI Commun. **19**(1), 35–48 (2006)
49. Sutcliffe, G., Suttner, C.: The TPTP Problem Library: CNF Release v1.2.1. J. Autom. Reason. **21**(2), 177–203 (1998)
50. The Coq Development Team: The Coq Proof Assistant Reference Manual—Version V8.2 (2008). http://coq.inria.fr
51. Tuch, H., Klein, G., Norrish, M.: Types, bytes, and separation logic. In: Hofmann, M., Felleisen, M. (eds.) Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07), pp. 97–108, Nice, France, January (2007)
52. Turing, A.M.: Checking a large routine. In: Report of a Conference on High Speed Automatic Calculing Machines. Mathematical Laboratory, Cambridge, pp. 67–69 (1949)
53. Weber, T.: SMT Solvers: New Oracles for the HOL Theorem Prover. Special Section on VSTTE (2009). doi:10.1007/s10009-011-0188-8