Verification by Abstract Interpretation, Soundness and Abstract Induction

Patrick COUSOT

Courant Institute of Mathematical Sciences, New York University
pcousot@cims.nyu.edu, cims.nyu.edu/~pcousot

Abstract

Automatic program verification tools have to cope with programming language and machine semantics, undecidability, and mathematical induction, and so are all complex and imperfect. The ins and outs of automatic program verification will be discussed in light of the theory and practice of abstract interpretation [18, 19, 22].

Programming language semantics

If Edsger W. Dijkstra could claim in his 1972 Turing lecture [29, p. 863] that "When FORTRAN has been called an infantile disorder, full PL/1, with its growth characteristics of a dangerous tumor, could turn out to be a fatal disease.", PL/1 and its 1000 pages formal operational semantics defined in VDL [7] now appears as a marvel of simplicity compared to present-day programming languages [57] and their informal definitions [1].

The formal specification of the semantics of programming languages is now a well-established and well-taught subject in computer science, with many proposals at various levels of abstraction such as small/big step operational, denotational, relational and predicate transformer, axiomatic, and algebraic semantics. Surprisingly, very few languages have received a formal definition of their semantics (with few exceptions such as ML [47, 48]). None of the most popular programming languages has a formal and complete definition of its semantics that can reasonably be used as a basis for formal verification. This may come from the high complexity of present-day programming languages.

This problem is often solved by the verification community by considering models or programs that are never checked to be valid or toy languages hardly ever used for programming. Typically there are reals no floats, arrays but infinite, lists or trees but no pointers, etc.

Considering actual languages is a huge and long effort [24, 46], not necessarily more highly rewarded.

Machine semantics

For a long time, computers have been miraculously correct (whenever bugs were found the corresponding pages of the user manual had just to be teared up). Since the exponential improvement of the power of computers by Moore's Law started to slow down significantly a decade ago, computer industry technology moved to multiprocessors. Because memory access is about 100 times slower than CPU, modern microprocessors optimize memory accesses by reordering memory operations using caches and memory banks. On most modern multiprocessors memory operations are not executed in the order specified by the program code. In multi-threaded environments (or when interfacing with other hardware via memory buses) this out-of-order execution may lead to serious problems. Barriers or fences have been introduced to synchronize memory accesses. Their functioning is far from being well-documented, sometimes wrong, and their semantics is complex [2]. Of course different machines have different semantics of concurrency so programs and program verification methods are no longer portable.

This problem is often solved by the verification community by considering sequential consistency [44] (which was valid with classical memory models [20] but no longer on modern architectures [6, 31]), or by putting severe restrictions on acceptable implementations (such as multi-programming on a single processor [50]) with potential terrible slowdowns (potentially by factors of tenth of thousands) resulting from naïve compilation.

Soundness

Soundness refers to the possibility of proving that any automatically verified program specification for any program of a programming language is valid with respect to a formal definition of the semantics of the programming language. This does not necessarily mean absolute correctness when putting restrictions (such as no parameter aliasing in Clousot [10] or soundness up to the first error with unpredictable effects in Astrée [23]), preferably machine checkable ones (such as the former restrictions on dynamic memory allocation in Astrée [23], now dropped).

The most common method to circumvent the problem is to ignore it completely [58]. For example unsound static analyzers are easier, indeed much much easier, to develop than sound ones, and they have commercial successes [55], based on heuristic organization of the reported errors. This does not extend to subtle errors and high quality software as required *e.g.* in aerospace [56] which market is unfortunately much smaller than that of easily analyzable dirty software.

On the contrary, the ultimate objective is that of automatically verified automatic verification methods. Unfortunately, automatic soundness verifications are still is their infancy [9] since the mechanization of abstract interpretation theory is a difficult problem.

Undecidability

Assuming that what is to be verified is well-defined, undecidability is the next challenge. In full generality, mathematical induction, is required to verify programs, and this induction may be non-trivial

PPDP '15, June 13–16, 2015, Siena, Italy..
 ISBN 978-1-4503-3516-4/15/07…$15.00.
http://dx.doi.org/10.1145/2790449.2790451

(mathematicians sometimes take centuries to solve apparently simple problems [59]).

First attempts in data-flow analysis [3] considered finite abstractions to get fixpoint equations in finite abstract domains. There was no soundness consideration. Only informal explanations where provided to justify the correctness of the static analysis, without formal connection to a clearly worded program semantics [18].

Unfortunately, finite abstractions, although complete for a given program and a given property [15], are always surpassed by infinitary abstraction for a programming language, as proved in [21].

A similar and even more radical and trivial approach has been to consider finite state systems [11, 53] thus avoiding all problems but combinatorial explosion. After enormous but fruitless efforts of the computer aided verification community, this simplistic approach is progressively recognized as a useful toy with no hope of scalability. The simplest solution is then to claim that scalability is unnecessary. This is a bit in contradiction with the software everywhere perspectives of modern societies.

Another easy approach to undecidability, is to completely forget about termination. This is the case *a.o.* of finite abstraction refinements [12], even when refined to avoid useless refinements [26]. There is even a best way to refine abstractions for a given program and a property [34, 35] which has never been exploited in practice, probably because it is more complicated and does not terminate either. Most engineers are patient and can wait for hours or days if the results are useful, provided they know an upper bound on how long they have to wait. Of course time-out, which is a too trivial widening, is unsatisfactory. Moreover, refinement is based on the idea than the first imprecise analyses may be able to prove results quickly without resorting to complex abstract domains. This might work on small programs, although the difference is not significative, but has not been shown to scale to very large programs which inevitably have their intricacies requiring complex abstractions [32] beyond naïve refinements towards popular abstract domains *e.g.* [27, 49]. A refinement per false alarm will hardly scale.

Undecidability can also be handled by looking for specific decidable cases. The most trivial case is programs without loops. Decidability requires restrictions on the properties (such as no type union in ML [14]) and the programming language (such as both branches of a conditional must have the same type [14]). For more general properties, the problem is that the considered cases are almost never applicable in practice and, despite that, the algorithms are of very high complexity so do not scale.

When all these biases will have shown to be too limited in practice, the next step is to use mathematical induction.

Mathematical induction

Induction is the basic mathematical method for reasoning on computer systems, most often on their structure (like structural induction in operational semantics [52]) or the progress of the computations [33], or inversely [41], or both [38]). Apart from the fact that program properties may not be computer-representable and that implication is undecidable (which is solved by considering abstract domains in SMT solvers [25, 28, 30] or by using proof checkers [51] instead of theorem provers [42]), the main problem is to find an inductive argument for the proof by induction, which is nothing but an infinite fixpoint computation [18]. Convergence must definitely be accelerated (unless the solution can be computed symbolically once and for all, in which case the widening just plugs in the appropriate solution [36]).

The first work-around is to ask the programmer to provide the inductive argument (such as all loop invariants [45]). This is a trivial widening to the proposed solution and to unknown if it is not valid. This is great for tiny programs but more difficult for larger ones, in particular because the inductive argument is often much larger than the program itself. This is also costly at the development time (even for projects that greatly simplify real life software[9, 40]). This is even more costly in the maintenance phase since any modification of the program requires a manual modification of the corresponding inductive argument (and even of the proof itself for interactive verification).

The second work-around is to ask the programmer not for the inductive argument but for a pattern of the inductive argument [37]. This always works when the pattern is per program since the limit of this approach is to ultimately provide the inductive argument itself. This can become rapidly very complex when the pattern is general enough (*e.g.* for non-linear invariants [16]) and very large programs. Moreover this does the solve the analysis termination problem so that convergence acceleration is necessary [4].

Abstract mathematical induction

Abstract domains generalize patterns of properties essentially by providing infinitely many patterns. Infinite abstract domains require convergence acceleration by extrapolation (widening/dual-widening) and interpolation operators (narrowing, dual-narrowing, which are equivalent up to the exchange of the parameters) to automate mathematical induction in the abstract domain [17, 27].

Trivial extrapolations include bounding computations [43] or abstracting to finite domains [39]. This is subject to the finite abstraction limitations [21]. It cannot be used to prove anything, but the presence of bugs at the beginning of the considered program executions.

The power of extrapolation/interpolation operators lies in the ability to enforce convergence in infinitely many different ways for infinitely many different programs.

In general, an increasing iterative static analysis using extrapolation of successive iterates by widening followed by a decreasing iterative static analysis using interpolation of successive iterates by narrowing (both bounded by the specification) can be further improved by a increasing iterative static analysis using interpolation of iterates with the specification by dual-narrowing until reaching a fixpoint and checking whether it is inductive for the specification. This can be done locally [5]. The soundness and termination properties are independent [13]. An example is dual-narrowing which generalizes Craig interpolation in first order logic pre-ordered by implication to arbitrary abstract domains and may not terminate.

Conclusion

A large fraction of research on program verification has been devoted to avoiding hard and difficult fundamental problems. We know, by undecidability, that full verification will always be automatically unsolvable. So the game is to anticipate how far the limit can be pushed. From what we learned from the past, we anticipate that abstract interpretation will be able to explain how it works [22]. It is another challenge to find what will work.

Moreover predicting the future is well-known to be error-prone. For example, [13] concluded that “Les techniques de mise au point encore largement utilisées dans l’industrie informatique du logiciel peuvent être en partie évitées (du moins pour les fautes de programmation si ce n’est pour les fautes de conception), en utilisant nos propositions d’analyse sémantique automatique des programmes, et ce, sans attendre les dix ans (ou plus) qui seront nécessaires pour que les techniques de vérification des programmes utilisant des démonstrateurs de théorèmes soient opérationnelles. Il est d’ailleurs certain que les méthodes que nous proposons sont complémentaires et offrent pour certains types d’analyses un rapport coût/bénéfice

très rentable."[1]. Replacing years by decades might still be a valid conclusion.

References

[1] 14882:2014, I.: Information technology — Programming languages — C++. ISO. (14 January 2014)

[2] Alglave, J.: A formal hierarchy of weak memory models. Formal Methods in System Design 41(2), 178–210 (2012)

[3] Allen, F.E.: A basis for program optimization. In: IFIP Congress (1). pp. 385–390 (1971)

[4] Amato, G., Maio, S.D.N.D., Meo, M.C., Scozzari, F.: Narrowing operators on template abstract domains. In: Bjørner and de Boer [8], pp. 57–72

[5] Amato, G., Scozzari, F., Seidl, H., Apinis, K., Vojdani, V.: Efficiently intertwining widening and narrowing. CoRR abs/1503.00883 (2015)

[6] Berry, D., Milner, R., Turner, D.N.: A semantics for ML concurrency primitives. In: Sethi, R. (ed.) Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 19-22, 1992. pp. 119–129. ACM Press (1992)

[7] Bjørner, D., Jones, C.B. (eds.): The Vienna Development Method: The Meta-Language, Lecture Notes in Computer Science, vol. 61. Springer (1978)

[8] Bjørner, N., de Boer, F.D. (eds.): FM 2015: Formal Methods - 20th International Symposium, Oslo, Norway, June 24-26, 2015, Proceedings, Lecture Notes in Computer Science, vol. 9109. Springer (2015)

[9] Boldo, S., Jourdan, J., Leroy, X., Melquiond, G.: A formally-verified C compiler supporting floating-point arithmetic. In: Nannarelli, A., Seidel, P., Tang, P.T.P. (eds.) 21st IEEE Symposium on Computer Arithmetic, ARITH 2013, Austin, TX, USA, April 7-10, 2013. pp. 107–115. IEEE Computer Society (2013)

[10] Christakis, M., Müller, P., Wüstholz, V.: An experimental evaluation of deliberate unsoundness in a static program analyzer. In: D'Souza, D., Lal, A., Larsen, K.G. (eds.) Verification, Model Checking, and Abstract Interpretation - 16th International Conference, VMCAI 2015, Mumbai, India, January 12-14, 2015. Proceedings. Lecture Notes in Computer Science, vol. 8931, pp. 336–354. Springer (2015)

[11] Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Logics of Programs, Workshop, Yorktown Heights, New York, May 1981. Lecture Notes in Computer Science, vol. 131, pp. 52–71. Springer (1981)

[12] Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1855, pp. 154–169. Springer (2000)

[13] Cousot, P.: Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French). Thèse d'État ès sciences mathématiques, Université Joseph Fourier, Grenoble, France (21 March 1978)

[14] Cousot, P.: Types as abstract interpretations. In: Lee, P., Henglein, F., Jones, N.D. (eds.) Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997. pp. 316–331. ACM Press (1997)

[15] Cousot, P.: Partial completeness of abstract fixpoint checking. In: Choueiry, B.Y., Walsh, T. (eds.) Abstraction, Reformulation, and Approximation, 4th International Symposium, SARA 2000, Horseshoe Bay, Texas, USA, July 26-29, 2000, Proceedings. Lecture Notes in Computer Science, vol. 1864, pp. 1–25. Springer (2000)

[16] Cousot, P.: Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In: Cousot, R. (ed.) Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3385, pp. 1–24. Springer (2005)

[17] Cousot, P., Cousot, R.: Static determination of dynamic properties of programs. In: Proc. Secont Int. Symp. on Programming. pp. 106–130. Dunod, Paris, France (1976)

[18] Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (eds.) POPL. pp. 238–252. ACM (1977)

[19] Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Aho, A.V., Zilles, S.N., Rosen, B.K. (eds.) POPL. pp. 269–282. ACM Press (1979)

[20] Cousot, P., Cousot, R.: Invariance proof methods and analysis techniques for parallel programs. In: Biermann, A., Guiho, G., Kodratoff, Y. (eds.) Automatic Program Construction Techniques, chap. 12, pp. 243–271. Macmillan, New York, New York, United States (1984)

[21] Cousot, P., Cousot, R.: Comparing the galois connection and widening/narrowing approaches to abstract interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP'92, Leuven, Belgium, August 26-28, 1992, Proceedings. Lecture Notes in Computer Science, vol. 631, pp. 269–295. Springer (1992)

[22] Cousot, P., Cousot, R.: Abstract interpretation: past, present and future. In: Henzinger, T.A., Miller, D. (eds.) Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria, July 14 - 18, 2014. pp. 2:1–2:10. ACM (2014)

[23] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The astreé analyzer. In: Sagiv, S. (ed.) Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings. Lecture Notes in Computer Science, vol. 3444, pp. 21–30. Springer (2005)

[24] Cousot, P., Cousot, R., Feret, J., Miné, A., Mauborgne, L., Monniaux, D., Rival, X.: Varieties of static analyzers: A comparison with ASTREE. In: First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering, TASE 2007, June 5-8, 2007, Shanghai, China. pp. 3–20. IEEE Computer Society (2007)

[25] Cousot, P., Cousot, R., Mauborgne, L.: Theories, solvers and static analysis by abstract interpretation. J. ACM 59(6), 31 (2012)

[26] Cousot, P., Ganty, P., Raskin, J.: Fixpoint-guided abstraction refinements. In: Nielson, H.R., Filé, G. (eds.) Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4634, pp. 333–348. Springer (2007)

[27] Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Aho, A.V., Zilles, S.N., Szymanski, T.G. (eds.) Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978. pp. 84–96. ACM Press (1978)

[28] Deters, M., Reynolds, A., King, T., Barrett, C.W., Tinelli, C.: A tour of CVC4: how it works, and how to use it. In: Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014. p. 7. IEEE (2014)

[29] Dijkstra, E.W.: The humble programmer. Commun. ACM 15(10), 859–866 (1972)

[30] D'Silva, V., Haller, L., Kroening, D.: Abstract satisfaction. In: Jagannathan, S., Sewell, P. (eds.) The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL

[1] The debugging techniques still in common use in the software industry can be avoided in part (at least for programming errors if not for design errors) using our proposed automatic semantic analysis techniques, without waiting for the ten years (or more) that will be needed for program verification techniques using theorem provers to be operational. It is also certain that the methods we offer are complementary and provide for certain types of analyzes a very profitable cost/benefit ratio.

'14, San Diego, CA, USA, January 20-21, 2014. pp. 139–150. ACM (2014)

[31] Farzan, A., Kincaid, Z., Podelski, A.: Proof spaces for unbounded parallelism. In: Rajamani and Walker [54], pp. 407–420

[32] Feret, J.: Static analysis of digital filters. In: Schmidt, D.A. (ed.) Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings. Lecture Notes in Computer Science, vol. 2986, pp. 33–48. Springer (2004)

[33] Floyd, R.: Assigning meaning to programs. In: Schwartz, J. (ed.) Proc. Symposium in Applied Mathematics, vol. 19, pp. 19–32. Amer. Math. Soc. (1967)

[34] Giacobazzi, R., Quintarelli, E.: Incompleteness, counterexamples, and refinements in abstract model-checking. In: Cousot, P. (ed.) Static Analysis, 8th International Symposium, SAS 2001, Paris, France, July 16-18, 2001, Proceedings. Lecture Notes in Computer Science, vol. 2126, pp. 356–373. Springer (2001)

[35] Giacobazzi, R., Ranzato, F., Scozzari, F.: Making abstract interpretations complete. J. ACM 47(2), 361–416 (2000)

[36] Gonnord, L., Halbwachs, N.: Combining widening and acceleration in linear relation analysis. In: Yi, K. (ed.) Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings. Lecture Notes in Computer Science, vol. 4134, pp. 144–160. Springer (2006)

[37] Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings. Lecture Notes in Computer Science, vol. 1254, pp. 72–83. Springer (1997)

[38] Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969), http://doi.acm.org/10.1145/363235.363259

[39] Hunt, S., Hankin, C.: Fixed points and frontiers: A new perspective. J. Funct. Program. 1(1), 91–120 (1991)

[40] Jourdan, J., Laporte, V., Blazy, S., Leroy, X., Pichardie, D.: A formally-verified C static analyzer. In: Rajamani and Walker [54], pp. 247–259

[41] Jr., J.H.M., Wegbreit, B.: Subgoal induction. Commun. ACM 20(4), 209–222 (1977)

[42] Kaufmann, M., Moore, J.S.: Enhancements to ACL2 in versions 5.0, 6.0, and 6.1. In: Gamboa, R., Davis, J. (eds.) Proceedings International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2013, Laramie, Wyoming, USA, May 30-31, 2013. EPTCS, vol. 114, pp. 5–12 (2013)

[43] Kroening, D., Lewis, M., Weissenbacher, G.: Proving safety with trace automata and bounded model checking. In: Bjørner and de Boer [8], pp. 325–341

[44] Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers 28(9), 690–691 (1979)

[45] Leino, K.R.M., Wüstholz, V.: The dafny integrated development environment. In: Dubois, C., Giannakopoulou, D., Méry, D. (eds.) Proceedings 1st Workshop on Formal Integrated Development Environment, F-IDE 2014, Grenoble, France, April 6, 2014. EPTCS, vol. 149, pp. 3–15 (2014)

[46] Logozzo, F.: Practical specification and verification with code contracts. In: Boleng, J., Taft, S.T. (eds.) Proceedings of the 2013 ACM SIGAda annual conference on High integrity language technology, HILT 2013, Pittsburgh, Pennsylvania, USA, November 10-14, 2013. pp. 7–8. ACM (2013)

[47] Milner, R., Tofte, M.: Commentary on standard ML. MIT Press (1991)

[48] Milner, R., Tofte, M., Harper, R.: Definition of standard ML. MIT Press (1990)

[49] Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)

[50] Miné, A.: Relational thread-modular static value analysis by abstract interpretation. In: McMillan, K.L., Rival, X. (eds.) Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings. Lecture Notes in Computer Science, vol. 8318, pp. 39–58. Springer (2014)

[51] Paulin-Mohring, C.: Introduction to the coq proof-assistant for practical software verification. In: Meyer, B., Nordio, M. (eds.) Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures. Lecture Notes in Computer Science, vol. 7682, pp. 45–95. Springer (2011)

[52] Plotkin, G.D.: The origins of structural operational semantics. J. Log. Algebr. Program. 60-61, 3–15 (2004)

[53] Queille, J., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) International Symposium on Programming, 5th Colloquium, Torino, Italy, April 6-8, 1982, Proceedings. Lecture Notes in Computer Science, vol. 137, pp. 337–351. Springer (1982)

[54] Rajamani, S.K., Walker, D. (eds.): Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015. ACM (2015)

[55] Ramos, D.A., Engler, D.R.: Practical, low-effort equivalence verification of real code. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 669–685. Springer (2011), http://dx.doi.org/10.1007/978-3-642-22110-1

[56] Randimbivololona, F.: Orientations in verification engineering of avionics software. In: Wilhelm, R. (ed.) Informatics - 10 Years Back. 10 Years Ahead. Lecture Notes in Computer Science, vol. 2000, pp. 131–137. Springer (2001)

[57] Stroustrup, B.: Foundations of C++. In: Seidl, H. (ed.) Programming Languages and Systems - 21st European Symposium on Programming, ESOP 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings. Lecture Notes in Computer Science, vol. 7211, pp. 1–25. Springer (2012)

[58] Wang, X., Zeldovich, N., Kaashoek, M.F., Solar-Lezama, A.: A differential approach to undefined behavior detection. ACM Trans. Comput. Syst. 33(1), 1:1–1:29 (2015)

[59] Wiles, A.: Modular elliptic curves and Fermat's last theorem. Annals of Mathematics 141(3), 443–551 (1995)