Journal of Universal Computer Science, vol. 13, no. 5 (2007), 629-638
submitted: 7/5/07, accepted: 25/5/07, appeared: 28/5/07 © *J.UCS*

Floating-Point Verification

John Harrison
(Intel Corporation, JF1-13, 2111 NE 25th Avenue,
Hillsboro OR 97124, USA
johnh@ichips.intel.com)

Abstract: This paper overviews the application of formal verification techniques to hardware in general, and to floating-point hardware in particular. A specific challenge is to connect the usual mathematical view of continuous arithmetic operations with the discrete world, in a credible and verifiable way.

Key Words: Formal Methods, Hardware Verification
Category: F.3.1, B.2

1 Introduction

Formal verification has become a well-established practice in parts of the hardware industry. A particularly notable success story has been in the area of floating-point arithmetic. Floating-point algorithms are often subtle and difficult to get right, and floating-point bugs have led to spectacular or costly problems in the past. On the other hand, they seem to lend themselves to relatively precise mathematical specification, and real industrial designs can be proved correct using current verification tools. We present a short survey of work in the area and outline how it fits into the wider world of formal methods.

2 Progams, Bugs and Verification

As most programmers know to their cost, writing programs that function correctly in all circumstances —or even saying what that means— is difficult. Most large programs contain 'bugs'. In the past, hardware has been substantially simpler than software, but this difference is eroding, and current leading-edge microprocessors are also extremely complex and usually contain errors. It has often been noted that mere testing, even on clever sets of test cases, is usually inadequate to guarantee correctness on *all* inputs, since the number of possible inputs and internal states, while finite, is usually astronomically large. For example [Dijkstra (1976)]:

> As I have now said many times and written in many places: program testing can be quite effective for showing the presence of bugs, but is hopelessly inadequate for showing their absence.

The main alternative to testing is formal verification, where it is rigorously *proved* that the system functions correctly on all possible inputs. This involves forming mathematical models of the system and its intended behaviour and linking the two:

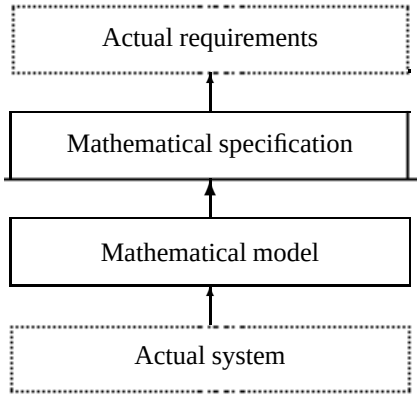

Figure 1: Linking the actual requirements to the actual system.

The facts that (i) getting a mathematical proof right is also difficult [DeMillo et al. (1979)], and (ii) correctness of formal models does not necessarily imply correctness of the actual system [Fetzer (1988)] caused much controversy in the 70s. But it is now widely accepted that formal verification, with the proof itself checked by machine, gives a much greater degree of confidence than traditional techniques. The main impediment to greater use of formal verification is not these generalist philosophical objections, but just the fact that it's rather difficult.

3 A panorama of formal verification

The introductory remarks may have suggested that verification and testing are antithetical approaches to correctness, but this is not so at all. Moreover, there are a number of approaches to formal verification that have quite different characteristics.

3.1 Verification versus testing

The advantages of formal verification over testing are so clear that it hardly seems worth enumerating them. Set against that, however, is the great difficulty of performing formal correctness proofs. Even arriving at a precise mathematical specification of how a computer artifact *should* behave, let alone proving that it *does* so, can be remarkably difficult. Performing a complete formal verification is usually challenging, often demanding high levels of expertise in mathematics and programming as well as detailed understanding of the target.

However, there need not be such a black-and-white separation of verification and testing. One can consider intermediate possibilities where although a full formal correctness proof is not performed, a more than usually rigorous logical analysis, often using similar technology, is undertaken. Simple static checking of programs for type errors or uninitialized variables can be seen as the first step from a purely dynamic approach, testing particular values at runtime, to a logical analysis of the code itself. This can be gradually extended to cover, for example, array bounds checking, null pointer dereferencing and failure to subsequently deallocate all allocated resources.

Such intermediate levels of verification have achieved some notable successes, in finding bugs and giving partial correctness guarantees. For example, the Static Driver Verifier at Microsoft has helped to find non-trivial bugs in Windows device drivers [Ball et al. (2006)]. Static analysis of the avionics software of the Airbus A380 has verified that it is impossible, under reasonable environmental assumptions, for a floating-point overflow exception to be raised.[1] In neither case has full correctness been verified, but in both cases the level of assurance has been substantially increased.

Moreover, verification and testing can sometimes support each other. For example, one effective methodology for verifying certain floating-point square root algorithms [Cornea-Hasegan (1998)] combines a relatively clean and simple analytical proof based on worst-case error bounds with the isolation of the relatively small number of possible inputs that the simple proof may not cover. A full correctness proof combines the high-level mathematical verification with explicit checking (albeit in a purely mathematical sense) of the exceptional cases. These cases, by design, are where floating-point rounding is particularly difficult because the decision between rounding up or down to the closest floating-point number is particularly fine. Thus, as well as being the foundation of a verification method, they make a very good set of test cases too.

3.2 Hardware versus software

Only in a few isolated safety-critical niches of the software industry is any kind of formal verification widespread, e.g. in avionics. But in the hardware industry, formal verification is widely practised, and increasingly seen as necessary. We can identify at least three reasons:

• Hardware is designed in a more modular way than most software. Constraints of interconnect layering and timing means that one cannot really design ‘spaghetti hardware’.

• More proofs in the hardware domain can be largely automated, reducing the need for intensive interaction by a human expert with the mechanical theorem-proving system.

• The potential consequences of a hardware error are greater, since such errors often cannot be patched or worked around, and may *in extremis* necessitate a hardware replacement.

[1] See `http://www.di.ens.fr/~cousot/COUSOTtalks/AcadSci06.shtml` for a recent presentation of this work.

3.3 The spectrum of formal verification techniques

There are many formal verification techniques in widespread use including:

- Propositional logic [Stålmarck and Säflund (1990), Biere et al. (1999), Sheeran and Stålmarck (2000), Abdulla et al. (2000), Bryant (1986), Moskewicz et al. (2001), Goldberg and Novikov (2002)]
- Symbolic simulation [Carter et al. (1979), Bryant (1985)]
- Symbolic trajectory evaluation [Seger and Bryant (1995), Yang (2000)]
- Temporal logic model checking [Clarke and Emerson (1981), Queille and Sifakis (1982)]
- Decidable subsets of first order logic [Burch and Dill (1994), Velev and Bryant (1999)]
- First order automated theorem proving [Schumann (2001)]
- Interactive theorem proving [Kaufmann et al. (2000b), Kaufmann et al. (2000a)]
- Higher-order logic theorem proving [Gordon and Melham (1993)]

Such a range of competing approaches may seem surprising, but can be explained by the fact that, generally speaking, techniques with a more limited range of applicability have the compensatory advantage of permitting fuller and/or more efficient automation, or simply of fitting better into the traditional design flow. (For example, symbolic simulation is a natural generalization of standard testing methods.) The above list has been arranged on this basis, with the 'limited but efficient' methods first and the 'general but inefficient' ones at the end.

Actually, they need not be considered as *competing* approaches; on the contrary there is considerable interest in combining them [Seger and Joyce (1991), Joyce and Seger (1993), Rajan et al. (1995)]. Intel is actively pursuing the development of tools combining general higher-order theorem proving and other techniques such as symbolic trajectory evaluation and temporal logic model checking [Aagaard et al. (1999)]. Such combinations have proved invaluable in several real hardware verification projects [Leary et al. (1999), Kaivola and Aagaard (2000)]. Another 'hybrid' application of formal verification is to prove correctness of some of the CAD tools used to produce hardware designs [Aagaard and Leeser (1994)] or even of the abstraction and reduction algorithms used to model-check large or infinite-state systems [Chou and Peled (1999)].

4 Floating-point verification

Representation of real numbers on the computer is fundamental to much of applied mathematics, from aircraft control systems to weather forecasting. Most applications use floating-point approximations, though this raises significant mathematical difficulties because of rounding and approximation errors. Even if rounding is properly controlled, 'bugs' in software using real numbers can be particularly subtle and insidious.

Yet because real-number programs are often used in controlling and monitoring physical systems, the consequences can be catastrophic. A spectacular example is the destruction of the Ariane 5 rocket shortly after takeoff in 1996, owing to an uncaught floating-point exception. Less dramatic, but very costly and embarrassing to Intel, was an error in the FDIV (floating-point division) instruction of some early Intel® Pentium® processors in 1994 [Pratt (1995)]. Intel set aside approximately $475M to cover costs arising from this issue.

So it is not surprising that a considerable amount of effort has been applied to formal verification in the floating-point domain, not just at Intel [Leary et al. (1999), Kaivola and Aagaard (2000)], but also at AMD [Moore et al. (1998), Rusinoff (1998)] and IBM [Sawada (2002)], as well as in academia [Jacobi (2002), Boldo (2004)]. Floating-point algorithms are in some ways an especially natural and appealing target for formal verification. It is not hard to come up with widely accepted formal specifications of how basic floating-point operations *should* behave. In fact, many operations are specified almost completely by the IEEE Standard governing binary floating-point arithmetic [IEEE (1985)]. This gives a clear specification that high-level algorithms can rely on, and which implementors of instruction sets and compilers need to realize. In some other respects though, floating-point operations present a difficult challenge for formal verification.

In many other areas of verification, significant success has been achieved using highly automated techniques, usually based on a Boolean or other finite-state model of the state of the system. But it is less easy to verify non-trivial floating-point arithmetic operations using such techniques. The natural specifications, including the IEEE Standard, are based on real numbers, not bit-strings. While simple adders and multipliers can be specified quite naturally in Boolean terms, this becomes progressively more difficult when one considers division and square root, and seems quite impractical for transcendental functions. So while model checkers and similar tools are of great value in dealing with low-level details, at least some parts of the proof must be constructed in general theorem proving systems that enable one to talk about high-level mathematics. For verification of the arithmetic hardware itself, most work at Intel is based on a tool that effectively combines a general theorem prover and a rich array of automated model checking techniques. As a paradigmatic example of how it may be used, a divider circuit might be verified by an inductive proof that certain invariants are maintained, while the circuit-level details are verified by symbolic trajectory evaluation [Leary et al. (1999), Kaivola and Aagaard (2000)]. As the floating-point algorithms become higher-level, e.g. implementing transcendental functions like *sin* on top of the hardware primitives, automated enumerative techniques become less useful, and the key is a programmable environment for general mathematical proofs.

There are many theorem proving programs with different strengths and weaknesses [Wiedijk (2006)]. Several of these, including at least ACL2, Coq, HOL Light and PVS, have been applied to floating-point verification. Our own work is conducted using the

HOL Light system [Harrison (1996)].[2] This is a general framework for verification of mathematical proofs, based on higher-order logic, implemented using a very simple kernel of basic logical inference rules, to provide a high level of assurance.

5 A typical high-level verification

To give a flavour of what the verification of a typical high-level floating-point algorithm involves, consider the sin / cos algorithm described in [Harrison (2000)]. As usual in modern transcendental function implementations [Tang (1991), Muller (1997)], the algorithm can be considered as three phases: (i) initial range reduction, (ii) core computation, (iii) reconstruction.

For our trigonometric functions, the initial argument x is reduced modulo $\pi/2$. Mathematically, for any real x we can always write:

$$x = N(\pi/2) + r$$

where N is an integer (the closest to $x \cdot \frac{2}{\pi}$) and $|r| \leq \pi/4$. The core approximation is then a polynomial approximation to $sin(r)$ or $cos(r)$ as appropriate, similar to a truncation of the familiar Taylor series:

$$sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$
$$cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$$

but with the pre-stored coefficients computed numerically to minimize the maximum error over r's range, using the so-called *Remez algorithm* [Remes (1934)]. Finally, the reconstruction phase: to obtain either $sin(x)$ and/or $cos(x)$, just return one of $sin(r)$, $cos(r)$, $-sin(r)$ or $-cos(r)$ depending on N modulo 4, e.g. $sin((4M+3)(\pi/2)+r) = -cos(r)$. In order to verify this algorithm we need to prove:

• The range reduction to obtain r is done accurately. This is by no means trivial since large floating-point numbers can come very close to multiples of $\pi/2$. We need to formalize some elementary theorems from diophantine approximation.

• The mathematical facts used to reconstruct the result from components are applicable. In this case, this just amounts to a few trigonometric identities, but is sometimes more involved.

• Stored constants such as the approximations to π and the polynomial coefficients are sufficiently accurate. This involves a combination of high-level mathematics and actual numerical approximation.

[2] See also `http://www.cl.cam.ac.uk/~jrh13/hol-light/`

• The power series approximation does not introduce too much error in approximation. The power series used are not Taylor expansions with a simple analytical error bound, so more general means must be employed.

• The rounding errors involved in computing with floating point arithmetic are within bounds. Every computation in the machine may deviate from its exact mathematical counterpart, and all the errors must be checked, bounded, and approved.

Most of these parts are non-trivial. Moreover, some of them require more pure mathematics than might be expected. Some parts (e.g. accumulating and bounding rounding errors) are unbearably tedious to do without programmability and machine checking, yet much of the high-level mathematics is beyond simple automated verification tools. Thus the use of a general framework such as HOL Light seems essential.

6 Concluding remarks

Formal verification is becoming established as best practice in several important niches of the hardware industry; this paper has just given a brief overview of the rich variety of methods in use. For more detail on hardware verification see [Kropf (1999), Peled (2001)] as well as the recent lectures in [Bernardo and Cimatti (2006)].

References

[Aagaard et al. (1999)] M. D. Aagaard, R. B. Jones, and C.-J. H. Seger. *Lifted-FL*: A pragmatic implementation of combined model checking and theorem proving. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin, and Laurent Théry, editors, *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*, volume 1690 of *Lecture Notes in Computer Science*, pages 323–340, Nice, France, 1999. Springer-Verlag.

[Aagaard and Leeser (1994)] Mark Aagaard and Miriam Leeser. Verifying a logic synthesis tool in Nuprl: A case study in software verification. In G. v. Bochmann and D. K. Probst, editors, *Computer Aided Verification: Proceedings of the Fourth International Workshop, CAV'92*, volume 663 of *Lecture Notes in Computer Science*, pages 69–81, Montreal, Canada, 1994. Springer Verlag.

[Abdulla et al. (2000)] Parosh Aziz Abdulla, Per Bjesse, and Niklas Eén. Symbolic reachability analysis based on SAT-solvers. In Susanne Graf and Michael Schwartzbach, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00)*, volume 1785 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[Ball et al. (2006)] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. Rajamani, and A. Ustuner. Thorough static analysis of device drivers. In *Proceedings of EuroSys'06, the European Systems Conference*, 2006.

[Bernardo and Cimatti (2006)] Marco Bernardo and Alessandro Cimatti, editors. *Formal Methods for Hardware Verification, 6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2006*, volume 3965 of *Lecture Notes in Computer Science*, Bertinoro, Italy, 2006. Springer-Verlag.

[Biere et al. (1999)] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1999.

[Boldo (2004)] Sylvie Boldo. *Preuves formelles en arithmétiques à virgule flottante*. PhD thesis, ENS Lyon, 2004. Available on the Web from `http://www.ens-lyon.fr/LIP/Pub/Rapports/PhD/PhD2004/PhD2004-05.pdf`.

[Bryant (1985)] R. E. Bryant. Symbolic verification of MOS circuits. In H. Fuchs, editor, *Proceedings of the 1985 Chapel Hill Conference on VLSI*, pages 419–438. Computer Science Press, 1985.

[Bryant (1986)] Randall E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, 1986.

[Burch and Dill (1994)] J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In David L. Dill, editor, *Computer Aided Verification, 6th International Conference, CAV '94*, volume 818 of *Lecture Notes in Computer Science*, pages 68–80, Stanford CA, USA, 1994. Springer-Verlag.

[Carter et al. (1979)] W. C. Carter, W. H. Joyner, and D. Brand. Symbolic simulation for correct machine design. In *Proceedings of the 16th ACM/IEEE Design Automation Conference*, pages 280–286. IEEE Computer Society Press, 1979.

[Chou and Peled (1999)] Ching-Tsun Chou and Doron Peled. Formal verification of a partial-order reduction technique for model checking. *Journal of Automated Reasoning*, 23:265–298, 1999.

[Clarke and Emerson (1981)] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In Dextrer Kozen, editor, *Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71, Yorktown Heights, 1981. Springer-Verlag.

[Cornea-Hasegan (1998)] Marius Cornea-Hasegan. Proving the IEEE correctness of iterative floating-point square root, divide and remainder algorithms. *Intel Technology Journal*, 1998-Q2:1–11, 1998. Available on the Web as `http://developer.intel.com/technology/itj/q21998/articles/art_3.htm`.

[DeMillo et al. (1979)] R. DeMillo, R. Lipton, and A. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22:271–280, 1979.

[Dijkstra (1976)] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[Fetzer (1988)] James H. Fetzer. Program verification: The very idea. *Communications of the ACM*, 31:1048–1063, 1988.

[Goldberg and Novikov (2002)] Evgueni Goldberg and Yakov Novikov. BerkMin: a fast and robust Sat-solver. In Carlos Delgado Kloos and Jose Da Franca, editors, *Design, Automation and Test in Europe Conference and Exhibition (DATE 2002)*, pages 142–149, Paris, France, 2002. IEEE Computer Society Press.

[Gordon and Melham (1993)] Michael J. C. Gordon and Thomas F. Melham. *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press, 1993.

[Harrison (1996)] John Harrison. HOL Light: A tutorial introduction. In Mandayam Srivas and Albert Camilleri, editors, *Proceedings of the First International Conference on Formal Methods in Computer-Aided Design (FMCAD'96)*, volume 1166 of *Lecture Notes in Computer Science*, pages 265–269. Springer-Verlag, 1996.

[Harrison (2000)] John Harrison. Formal verification of floating point trigonometric functions. In Warren A. Hunt and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design: Third International Conference FMCAD 2000*, volume 1954 of *Lecture Notes in Computer Science*, pages 217–233. Springer-Verlag, 2000.

[IEEE (1985)] IEEE. Standard for binary floating point arithmetic. ANSI/IEEE Standard 754-1985, The Institute of Electrical and Electronic Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, 1985.

[Jacobi (2002)] C. Jacobi. *Formal Verification of a Fully IEEE Compliant Floating Point Unit*. PhD thesis, University of the Saarland, 2002. Available on the Web as `http://engr.smu.edu/~seidel/research/diss-jacobi.ps.gz`.

[Joyce and Seger (1993)] Jeffrey J. Joyce and Carl Seger. The HOL-Voss system: Model-checking inside a general-purpose theorem-prover. In Jeffrey J. Joyce and Carl Seger, editors, *Proceedings of the 1993 International Workshop on the HOL theorem proving system*

and its applications, volume 780 of *Lecture Notes in Computer Science*, pages 185–198, UBC, Vancouver, Canada, 1993. Springer-Verlag.

[Kaivola and Aagaard (2000)] Roope Kaivola and Mark D. Aagaard. Divider circuit verification with model checking and theorem proving. In M. Aagaard and J. Harrison, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 338–355. Springer-Verlag, 2000.

[Kaufmann et al. (2000a)] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer, 2000.

[Kaufmann et al. (2000b)] Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer, 2000.

[Kropf (1999)] Thomas Kropf. *Introduction to Formal Hardware Verification*. Springer-Verlag, 1999.

[Moore et al. (1998)] J Strother Moore, Tom Lynch, and Matt Kaufmann. A mechanically checked proof of the correctness of the kernel of the $AMD5_K86$ floating-point division program. *IEEE Transactions on Computers*, 47:913–926, 1998.

[Moskewicz et al. (2001)] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*, pages 530–535. ACM Press, 2001.

[Muller (1997)] Jean-Michel Muller. *Elementary functions: Algorithms and Implementation*. Birkhäuser, 1997.

[Leary et al. (1999)] John O'Leary, Xudong Zhao, Rob Gerth, and Carl-Johan H. Seger. Formally verifying IEEE compliance of floating-point hardware. *Intel Technology Journal*, 1999-Q1:1–14, 1999. Available on the Web as `http://developer.intel.com/technology/itj/q11999/articles/art_5.htm`.

[Peled (2001)] Doron A. Peled. *Software Reliability Methods*. Springer-Verlag, 2001.

[Pratt (1995)] Vaughan R. Pratt. Anatomy of the Pentium bug. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *Proceedings of the 5th International Joint Conference on the theory and practice of software development (TAPSOFT'95)*, volume 915 of *Lecture Notes in Computer Science*, pages 97–107, Aarhus, Denmark, 1995. Springer-Verlag.

[Queille and Sifakis (1982)] J. P. Queille and J. Sifakis. Specification and verification of concurrent programs in CESAR. In *Proceedings of the 5th International Symposium on Programming*, volume 137 of *Lecture Notes in Computer Science*, pages 195–220. Springer-Verlag, 1982.

[Rajan et al. (1995)] S. Rajan, N. Shankar, and M. K. Srivas. An integration of model-checking with automated proof-checking. In Pierre Wolper, editor, *Computer-Aided Verification: CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, 1995. Springer-Verlag.

[Remes (1934)] M. Eugène Remes. Sur le calcul effectif des polynomes d'approximation de Tchebichef. *Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences*, 199:337–340, 1934.

[Rusinoff (1998)] David Rusinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998. Available on the Web at `http://www.onr.com/user/russ/david/k7-div-sqrt.html`.

[Sawada (2002)] Jun Sawada. Formal verification of divide and square root algorithms using series calculation. In Dominique Borrione, Matt Kaufmann, and J Moore, editors, *3rd International Workshop on the ACL2 Theorem Prover and its Applications*, pages 31–49. University of Grenoble, 2002.

[Schumann (2001)] Johann M. Schumann. *Automated Theorem Proving in Software Engineering*. Springer-Verlag, 2001.

[Seger and Joyce (1991)] Carl Seger and Jeffrey J. Joyce. A two-level formal verification methodology using HOL and COSMOS. Technical Report 91-10, Department of Computer

Science, University of British Columbia, 2366 Main Mall, University of British Columbia, Vancouver, B.C, Canada V6T 1Z4, 1991.

[Seger and Bryant (1995)] Carl-Johan H. Seger and Randal E. Bryant. Formal verification by symbolic evaluation of partially-ordered trajectories. *Formal Methods in System Design*, 6:147–189, 1995.

[Sheeran and Stålmarck (2000)] Mary Sheeran and Gunnar Stålmarck. A tutorial on Stålmarck's proof procedure for propositional logic. *Formal Methods in System Design*, 16:23–58, 2000.

[Stålmarck and Säflund (1990)] Gunnar Stålmarck and M. Säflund. Modeling and verifying systems and software in propositional logic. In B. K. Daniels, editor, *Safety of Computer Control Systems, 1990 (SAFECOMP '90)*, pages 31–36, Gatwick, UK, 1990. Pergamon Press.

[Tang (1991)] Ping Tak Peter Tang. Table-lookup algorithms for elementary functions and their error analysis. In Peter Kornerup and David W. Matula, editors, *Proceedings of the* 10^{th} *Symposium on Computer Arithemtic*, pages 232–236, 1991.

[Velev and Bryant (1999)] M. N. Velev and R. E. Bryant. Superscalar processor verification using efficient reduction of hte logic of equality with uninterpreted functions to propositional logic. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99*, volume 1703 of *Lecture Notes in Computer Science*, pages 37–53, Bad Herrenalb, Germany, 1999. Springer-Verlag.

[Wiedijk (2006)] Freek Wiedijk. *The Seventeen Provers of the World*, volume 3600 of *Lecture Notes in Computer Science*. Springer-Verlag, 2006.

[Yang (2000)] Jin Yang. A theory for generalized symbolic trajectory evaluation. In *Proceedings of the 2000 Symposium on Symbolic Trajectory Evaluation*, Chicago, 2000. Available via `http://www.intel.com/research/scl/stesympsite.htm`.