A Computing Medley on Program Verification, Specification and Automated Reasoning

John A. van der Poll
School of Computing, UNISA, Pretoria, South Africa
vdpolja@unisa.ac.za

Abstract

A brief overview of the science of formal program verification is presented, a topic close to the heart of Derrick Kourie to whom this article is dedicated in honour of his sixtieth birthday. No account would do justice to this topic without referring to the well-known Floyd-Hoare axiomatic approach to the verification and construction of programs. The specification of a program in terms of a precondition, program statement and postcondition is touched on and is followed by specification methods employed during the earlier phases of system development. Reasoning about the properties of a specification is a rewarding exercise since it may lead to useful insights. Modern specification languages often support set-theoretic constructs and these pose demanding challenges to automated reasoning programs. To this end the science of Automated Reasoning has made remarkable progress as far as tool usage is concerned.

Keywords: *Automated reasoning, formal specification, heuristics, OTTER, Prover9, resolution, set theory, Z*

Computing Review Categories: *D.2.4, F.3.1, F.4.1, I.2.3*

1 Introduction

The work of Robert Floyd [11] and C A R Hoare [12] together probably constitute some of the earliest writings on the formal verification of computer programs. Essentially Floyd defined rules for the construction of flowcharts while Hoare defined a number of similar deduction rules, based on the notions of preconditions, postconditions and program statements for proving the correctness of programming constructs.

Traditionally verification principles are presented in terms of compilable programming language constructs, e.g. if statements, while statements, etc. Mathematical rigour may, however, be employed much earlier in the design phase and this has led to the development of a multitude of formal specification techniques like VDM [14], Z [6, 15] and B [1] to name but a few. In essence a specification of a system defines *what* the resultant system must do rather than saying how it is to be achieved. One of the benefits to be realised through the use of a formal specification is that the specifier may reason about the properties of the system to be built at a very early stage in the development process. Any errors or omissions discovered in the user requirements could, therefore, be rectified earlier, leading to reduced costs [21].

Mathematical set theory [10] is a basic, yet deep, underlying commodity in Computing, e.g. any good text on relational databases tells us that databases are based on functions — given a person's ID number (a key), the system returns the record for the person. It is no coincidence, therefore, that many formal specification languages (e.g. Z, B, etc.) are in part based on set theory. This brings about significant challenges when one tries to reason about the properties of specifications written in these languages. Set theory is highly hierarchical, since an object (e.g. a set) at a very fine level of granularity may be an element of another (coarser) object, which may in turn be a member of another, even coarser object. Iterations through these levels often give rise to much activity and the generation of much irrelevant information [26]. If one, therefore, embarks on the use of an automated reasoner to prove properties of these set-theoretic specifications, then one quickly encounters a number of time- and space complexity problems. [7, 22, 26].

Automated reasoning tools have steadily improved over the last couple of decades and in this paper I will trace some of these advances made.

1.1 Layout of this Paper

In Section 2 a small coding example of the utility of Hoare's verification rules is presented. Following this I show in Section 3 an example of a Z specification which, as mentioned above, is a document produced early on in the development of a system. The utility of such a formal viewpoint is illustrated when omissions in the specification are identified through a manual proof. Thereafter I take a very short trip through reasoning with set theory since the 1970s. In Section 4 I introduce, amongst others, the OTTER theorem prover [17] which I often used in reasoning about the properties of specifications. It is shown how this reasoner easily discharges some proofs, yet has difficulty with more complicated proofs. Newer, more sophisticated provers emerge all the time and in the same section I mention briefly two other state of the art reasoners, namely, Vampire [27] and Prover9 [18], the latter being the successor of OTTER.

2 Hoare Logic: An Example

An expression of the form $\{P\}S\{Q\}$ where P and Q are properties of the program variables and S is a program (a single statement or large code fragment), is called a Hoare triple [12, 2]. P is called the precondition and Q is known as the postcondition of S. In this paper $\{P\}S\{Q\}$ is interpreted as follows: If statement S and all its associated variables are defined in context and precondition P is satisfied before the execution of S, execution of S is guaranteed to terminate, and afterwards, the program variables will satisfy Q [2]. This property is defined as *total correctness* by Baber [4]. Over time proof rules for the verification of assignment statements, conditionals, sequential composition, looping constructs, procedure calls, etc. have been defined.

Verification theory involving pre- and postconditions allows us to solve a rather common problem in Computing, namely, if the postcondition to a code fragment is known, how does one determine the precondition, i.e. where does one have to start to ensure the end result? The answer to this question is of value in requirements engineering [20] as well; if a user states some requirements (postcondition) and a software engineer suggests a solution, then what should be in place beforehand to ensure that the solution proposed will satisfy the requirements of the user, i.e. what is the precondition? For example, suppose we require a balance to be above a certain minimum after a user has withdrawn an amount of money. If the minimum balance is R50 and the user withdraws R130, then ignoring transaction costs, it should be easy to see that the user should start with at least R180 in the account.

The above example is relatively simple, but for more complicated calculations we need formal rules. Example 1 illustrates how two of the rules of Hoare logic may be used to calculate a precondition for a more complex system.

Example 1

Suppose we want to calculate the precondition, P for the sequence of statements $x := x - 1$ and $y := y - 1$, given a postcondition $\{z-1 \leq y < x \leq w\}$. We employ the assignment axiom (1) as well as the proof rule for sequential composition (2) defined below.

$$\{\, Q[x := e]\}\, x := e\, \{Q\} \tag{1}$$

$$\{P\}S1;\ S2\{Q\} \leftarrow \{P\}S1\{R\} \wedge \{R\}S2\{Q\} \tag{2}$$

First we apply the assignment axiom and replace y with $(y-1)$ in the postcondition Q to obtain an intermediate precondition, say, $P1 = z - 1 \leq (y-1) < x \leq w$. Now we apply the rule of sequential composition and equate $P1$ as the postcondition for the first assignment statement $x := x - 1$. Applying the assignment axiom again in $P1$ gives us the final precondition $P = z-1 \leq (y-1) < (x-1) \leq w$ which can be simplified to $z \leq y < x \leq w+1$. Therefore: $\{z \leq y < x \leq w+1\}\, x := x - 1;\ y := y - 1\, \{z - 1 \leq y < x \leq w\}$. The triple $(\{\mathit{precondition}\}, \mathit{program\ fragment}, \{\mathit{postcondition}\})$ is often referred to as a specification for the particular program fragment. The reader is referred to any of [4] or [2] for further details.

In the next section I introduce a specification mechanism applicable to the earlier phases of system development.

3 Formal Specification Using Z

Z specifications are essentially set-theoretic specifications. Z is based on a strongly-typed fragment of Zermelo-Fraenkel set theory [10], expressed in suitable first-order languages augmented by schema notation. The schema notation arose, according to Hoare, as a mechanism to separate visually the formed parts of a specification from the semi-formal and informal documentation around them [13, 26].

A schema is generally divided into two parts: a declaration part in which variables to be used in the specification are declared, and a predicate part in which constraints are placed on the variables. Schemas may be combined using the schema calculus [21]. Z as a specification tool is different from the style presented

in Section 2, since its notation employs first-order logic rather than programming constructs which emerge further on in the development cycle. Z may, therefore, be viewed as a front end to Hoare logic.

Next we give an example of a partial Z specification, starting with a requirements definition of what needs to be specified. The development below stems essentially from [28] and [25].

3.1 An oil terminal control system

Design a Z specification to enable a sea port authority to keep track of oil tankers arriving and docking at its berths. If a tanker arrives, and there is an open berth, the berth will be allocated to that tanker, else the tanker must join the back of a queue, waiting to be berthed. Information to be kept include the berths maintained by the port authority, all tankers known to the authority, a queue of all tankers waiting to be berthed and a record of which tanker occupies which berth.

The following additional requirements are to be met:

(1) A tanker cannot simultaneously be in the queue and in a berth.

(2) The tankers queueing will all be different.

(3) A tanker will queue only if all the berths are full.

(4) The tankers occupying berths will all be different.

(5) Two tankers cannot occupy the same berth.

An abstract state space of the system in schema notation is (numbers in brackets label the formal counterparts of the informal requirements (1) to (5) above):

$$
\begin{array}{|ll}
\multicolumn{2}{l}{\textit{Oil\_tanker\_control}} \\
\hline
\textit{berths} : \mathbb{P}\,\textit{Berth} & \\
\textit{known} : \mathbb{P}\,\textit{Tanker} & \\
\textit{waiting} : \mathrm{seq}\,\textit{Tanker} & \\
\textit{docked} : \textit{Tanker} \rightarrowtail\!\!\!\!\!\!\,\twoheadrightarrow \textit{Berth} & (4), (5) \\
\hline
\mathrm{ran}\,\textit{waiting} \cap \mathrm{dom}\,\textit{docked} = \varnothing & (1) \\
\#\textit{waiting} = \#(\mathrm{ran}\,\textit{waiting}) & (2) \\
\#\textit{waiting} > 0 \Rightarrow (\textit{berths} = \mathrm{ran}\,\textit{docked}) & (3) \\
\mathrm{ran}\,\textit{docked} \subseteq \textit{berths} & \\
\mathrm{ran}\,\textit{waiting} \cup \mathrm{dom}\,\textit{docked} \subseteq \textit{known} & \\
\hline
\end{array}
$$

I shall define one partial operation on the state to illustrate some ideas. The reader is referred to [25] for a comprehensive operational development of the above requirements definition. An operation to allow for the successful arrival of a tanker at the port is:

$$
\begin{array}{|l}
\textit{Arrive} \\
\hline
\Delta\,\textit{Oil\_tanker\_control} \\
\textit{tanker}? : \textit{Tanker} \\
\textit{report}! : \textit{Report} \\
\hline
(\mathrm{ran}\,\textit{docked} \neq \textit{berths} \wedge \\
\quad (\exists\, b)(b \in \textit{berths} \wedge \\
\quad\quad b \notin \mathrm{ran}\,\textit{docked} \wedge \\
\quad\quad \textit{docked}' = \textit{docked} \oplus \{\textit{tanker}? \mapsto b\}) \wedge \\
\quad \textit{waiting}' = \textit{waiting} \wedge \\
\quad \textit{known}' = \textit{known} \wedge \\
\quad \textit{report}! = \textit{OK}) \\
\quad\quad \vee \\
(\mathrm{ran}\,\textit{docked} = \textit{berths} \wedge \\
\quad \textit{waiting}' = \textit{waiting} \frown \langle \textit{tanker}? \rangle \wedge \\
\quad \textit{docked}' = \textit{docked} \wedge \\
\quad \textit{known}' = \textit{known} \wedge \\
\quad \textit{report}! = \textit{wait}) \\
\hline
\end{array}
$$

The operation is called *Arrive* and the tanker that arrives is denoted by *tanker*?. $\Delta\,\textit{Oil\_tanker\_control}$ in the declaration part indicates that, since a new tanker arrives there is a possible change in the state. In Z a prime ($'$) denotes the value of a variable after an operation. The symbol $\oplus$ is Z's overriding operator — it replaces tuples in a relation or a function where the first coordinate matches the coordinate of the overriding tuple. Concatenation of sequences is indicated by $\frown$. The first *disjunct* inside the predicate part specifies what happens when a tanker arrives and there is at least one free berth. The berth is allocated to the tanker. The second disjunct specifies what happens if there is no free berth for the newly arrived tanker — the tanker is added to the back of a waiting queue. Note that *Arrive* does not specify any error conditions. Details on these appear in [25].

Proving properties of a formal specification may be a rewarding exercise. Amongst other things reasoning about the specification may reveal possible omissions. In the next section we see how an attempt at discharging a proof obligation reveals that a precondition to an operation has to be strengthened.

3.2 The utility of proof

Some proof obligations arise from the definition of *Arrive* and I shall give one such proof attempt to illustrate the idea. Recall that the state of this system is given by *Oil_Tanker_Control* in Section 3.1. One of the invariants of the *after state*, i.e. the state resulting from *Arrive* is:

$$\mathrm{ran}\,\textit{waiting}' \;\cup\; \mathrm{dom}\,\textit{docked}' \;\subseteq\; \textit{known}' \qquad (3)$$

A proof of (3) would show that *Arrive* preserves part of the invariant of the system.

Suppose the first disjunct of *Arrive* is applicable, that is, ran $docked \neq berths$, i.e. there is a free berth available when a new tanker arrives. A proof of (3) boils down to proving each of the following two cases:

(1) ran $waiting' \subseteq known'$

(2) dom $docked' \subseteq known'$

Case (1) above presents no problem, since:

if ran $waiting \subseteq known$ [before state invariant]
then ran $waiting' \subseteq known$ [$waiting' = waiting$]
i.e. ran $waiting' \subseteq known'$ [$known' = known$]

Case (2) however reveals a problem:

dom $docked'$
$=$ dom $(docked \cup \{tanker? \mapsto b\})$
[$\exists\, b \in (berths - \text{ran}\ docked)$]
$=$ dom $docked \cup$ dom$\{tanker? \mapsto b\}$ [law of $\cup$]

Now we have:

dom $docked \subseteq known$
[invariant part of before state]
i.e. dom $docked \subseteq known'$ [known' = known]

The last part to prove is:

dom$\{tanker? \mapsto b\} \subseteq known'$
i.e. $\{tanker?\} \subseteq known'$ [definition of dom]
i.e. $tanker? \in known'$

We notice that in order for *tanker*? to be an element of $known'$, we need to add $tanker? \in known$ to the precondition of *Arrive* above, i.e. we need to strengthen the precondition. This is, however, not the end of the story. Our precondition needs to be strengthened further, since it has to provide for the case that a newly arrived tanker must not already be queued or docked. Details on how this is discovered through a further proof attempt appear in [25].

Proofs by hand are tedious for humans and it is, therefore, no coincidence that much research went into the development of automated or at least semi-automated reasoners over the past couple of decades. In the following section we take a brief look at some of these developments and challenges. Owing to my research interests, the discussion centers around progress made with resolution-based reasoners.

4 Advances Made by Automated Reasoners

The paper by Woodrow (Woody) Bledsoe in the collection compiled by Jörg Siekmann and Graham Wrightson [5] gives an account of an early theorem-proving program called PROVER. In essence the prover divides a problem into subproblems through two routines called SPLIT (for general mathematical problems) and REDUCE (for problems in set theory). The subproblems were then passed on to a resolution procedure to perform the necessary proofs. This divide-and-conquer approach was very necessary, as Bledsoe wrote: "Resolution, when used, is relegated to the job it does best, proving relatively easy assertions". The capabilities of these early provers were limited, as pointed out again by Bledsoe himself: "But this ability [dividing a problem into subproblems], which is really an overall *planning* capacity, is still severely limited". An example of the operation of the split and reduce routines are given below.

Example 2

Suppose a specifier wishes to prove $(t \in (A \cup B) \rightarrow D)$ where A, B and D have been defined before. The reduce routine would rewrite the proof obligation as $((t \in A \vee t \in B) \rightarrow D)$ whereafter SPLIT would divide it into two subproblems $(t \in A \rightarrow D)$ and $(t \in B \rightarrow D)$. The resolution procedure would then attempt to refute each of the two subproblems.

Resolution-based reasoners made steady progress through the 1980s and nineties. The OTTER (Organised Techniques for Theorem-proving and Effective Research) theorem prover [17] is a success story from this era. I have used this reasoner extensively in my own work (OTTER easily discharges the proof obligation in Example 2 above), and so has one of the fathers of Automated Reasoning, Larry Wos [31]. In fact, a variant of OTTER called EQP was used by its author (McCune) to discharge a long standing open conjecture, namely, that Robbins Algebras are boolean algebras [16]. The automated proof of this famous conjecture made the front pages of national newspapers worldwide [8].

Reasoning with set theory continues to pose difficulty to resolution-based reasoners, despite the theoretical advances made in this area, e.g. hyperresolution [24], set-of-support strategy [32], paramodulation [23], resonance [30] and the hot-list strategy [33] to name just a few. A number of researchers, e.g. [7], [22] and [26] demonstrated the challenges that set-theoretic proofs pose to automated reasoning programs. Set-theoretic constructs are strongly hierarchical and may

lead to constructs that are deeply nested. A reasoner should avoid 'opening up' every set-theoretic definition so that inferences can be made at the appropriate level. Some definitions must, however, be expanded. A key technique would be to layer the deductions and to identify suitable occasions for crossing from one layer to another.

It is generally recognised that *heuristics* would play an important role in launching an attack on the complexities of set theory, thereby increasing the success rate of an automated reasoner [8]. An example of one of these heuristics from my own work [26] and [25] in reasoning about mathematical set theory is given below. All the proof attempts reported on in the following section were done on a Pentium III with a clock speed of 600MHz and 32MB RAM, running Red Hat Linux Release 9.

Example 3

Suppose we want to prove:

$$\mathbb{P}\{0,1\} = \{\varnothing, \{0\}, \{1\}, \{0,1\}\} \tag{4}$$

When writing the contents of sets in list notation, e.g. the contents of the above set on the right-hand side, one naturally tends to define these sets using one or more levels of indirection by moving from the various elements to a symbol representing the collection of those elements. Therefore, we rewrite the above set-theoretic equality to make the relevant constructions explicit, i.e.:

$$A = \{0\} \wedge B = \{1\} \wedge C = \{0,1\} \wedge D = \mathbb{P}(C) \wedge E = \{\varnothing, A, B, C\} \rightarrow D = E \tag{5}$$

OTTER fails to find any proof for (5) in 30 minutes. Suppose we cut down on the complexity of information by eliminating the symbol E in the above definition. Hence we effectively remove one extra level of indirection, i.e.

$$A = \{0\} \wedge B = \{1\} \wedge C = \{0,1\} \wedge D = \mathbb{P}(C) \rightarrow D = \{\varnothing, A, B, C\} \tag{6}$$

With the above formulation OTTER finds a proof in 4 minutes 5 seconds. This brought about an important heuristic proposed in [25], namely, to avoid unnecessary levels of elementhood in set-theoretic formulae by using the elements of sets directly. Through the use of the divide-and-conquer technique, this last proof attempt may be streamlined even further. The details appear in [25].

The Vampire theorem prover [27] considered to be the benchmark for resolution-based reasoners, owing to its consistent success at the annual CADE ATP System Competitions (CASC [19], see also http://www.cs.miami.edu/~tptp/CASC/), also fails to find a proof of (5) in 30 minutes. If we, however, resort to (6) then Vampire easily finds a proof in just 0.8 seconds. The OTTER reasoner has since been decommissioned by McCune and replaced by a more advanced reasoner, Prover9 [18] (see also http://www.cs.unm.edu/~mccune/prover9/). I have not tested Prover9 on the proofs I did with OTTER but I recently secured a MSc student in this area and it would be interesting to see the performance of Prover9 on the proofs in [25].

5 Summary

In this paper I gave a brief overview of some aspects of formal program specification and verification, the importance of stating proof obligations arising from a specification and the role that automated reasoners can play in discharging these proof obligations to increase confidence in the correctness of the specification.

The Floyd-Hoare contributions made to the verification scene during the early years were acknowledged. The construction of a formal specification and the subsequent reasoning thereof for humans turn out to be a tedious task. Mathematical set theory on which many formal specification languages are based, pose demanding challenges to resolution-based reasoners. Nevertheless, much progress with the automation of proofs have made during the last couple of decades, owing to theoretical advances in this area as well as the sophistication of the reasoners employing the theory.

Some basic challenges in set-theoretic reasoning remain. In his book Automated Reasoning: 33 Basic Research Problems [29], Larry Wos poses a research question largely still unsolved: *What inference rule, if any, effectively performs for set theory as paramodulation does for equality?* The object of an application of paramodulation is to cause an equality substitution to take place from one clause into another. In their paper, Bailin and Barker-Plummer [3] claimed to have found such a rule which they call the 'Z-match', but they also state that they have not been able to prove the challenge problem that accompanies the research problem proposed by Wos. Their approach is furthermore not resolution-based.

So, yes Derrick, there are definite advances that have been made in the formal methods arena. Since some disagree on the usefulness of mathematical formalism [9], an obvious next step as proposed by Dr. Stefan Gruner and others would be to determine the utility of formal methods in industry. Hopefully this is a project we can all enjoy doing in the years to come!

References

[1] J.-R. Abrial. *The **B** Book: Assigning Programs to Meanings*. Cambridge University Press, August 1996.

[2] R. Backhouse. *Program Construction: Calculating Implementations from Specifications*. John Wiley and Sons, 2003.

[3] S. C. Bailin and D. Barker-Plummer. Z-match: An Inference Rule for Incrementally Elaborating Set Instantiations. *Journal of Automated Reasoning*, 11(3):391 – 428, December 1993.

[4] R. L. Barber. *The Spine of Software*. John Wiley and Sons, 1992.

[5] W. W. Bledsoe. Splitting and Reduction Heuristics in Automatic Theorem Proving. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning: Classical Papers on Computational Logic 1967 to 1970, Volume 2*, pages 508 – 530. Springer-Verlag, Germany, 1983.

[6] J. P. Bowen. Z: A formal specification notation. In M. Frappier and H. Habrias, editors, *Software Specification Methods: An Overview Using a Case Study*, FACIT, chapter 1, pages 3 – 19. Springer-Verlag, 2001.

[7] R. Boyer, E. Lusk, W. McCune, R. Overbeek, M. Stickel, and L. Wos. Set Theory in First-Order Logic: Clauses for Gödel's Axioms. *Journal of Automated Reasoning*, 2(3):287 – 327, September 1986.

[8] A. Bundy. A Survey of Automated Deduction. Technical Report EDI-INF-RR-0001, Division of Informatics, University of Edinburgh, April 1999.

[9] B. L. Charlier and P. Flener. Specifications Are Necessarily Informal or: Some More Myths of Formal Methods. *The Journal of Systems and Software*, 40(3):275–296, March 1998.

[10] H. B. Enderton. *Elements of Set Theory*. Academic Press, Inc., 1977.

[11] R. W. Floyd. Assigning Meanings to Programs. *Proceedings of the Symposium of Applied Mathematics*, 19:19 – 32, 1967. American Mathematical Society, Providence, R. I.

[12] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576 – 580, 583, October 1969.

[13] C. A. R. Hoare. Preface. In D. Bjørner, C. Hoare, and H. Langmaack, editors, *VDM'90: VDM and **Z** - Formal Methods in Software Development*, number 428 in LNCS, 1990.

[14] C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall International (UK), 1986.

[15] D. Lightfoot. *Formal Specification Using Z*. Palgrave Macmillan, 2nd edition, 2001.

[16] W. W. McCune. Solution of the robbins problem. *Journal of Automated Reasoning*, 19(3):263 – 276, 1997.

[17] W. W. McCune. *OTTER 3.3 Reference Manual*. Argonne National Laboratory, Argonne, Illinois, August 2003. ANL/MCS-TM-263.

[18] W. W. McCune. *Prover9 is Better Than Otter*. Argonne Workshop on Automated Reasoning and Deduction (AWARD), August 2005.

[19] F. J. Pelletier, G. Sutcliffe, and C. Suttner. The Development of CASC. *AI Communications*, 15(2):79 – 90, 2002.

[20] K. Pohl. The Three Dimensions of Requirements Engineering. In C. Rolland, F. Bodart, and C. Cauvet, editors, *Fifth International Conference on Advanced Information Systems Engineering (CAiSE'93)*, pages 275 – 292, Paris, 1993. Springer-Verlag.

[21] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and **Z***. Prentice-Hall, 2nd edition, 1996.

[22] A. Quaife. *Automated Development of Fundamental Mathematical Theories*. Automated Reasoning Series. Kluwer Academic Publishers, 1992.

[23] G. Robinson and L. Wos. Paramodulation and Theorem Proving in First-Order Theories with Equality. In B. Meltzer and D. Michie, editors, *Machine Intelligence, Volume 4*, pages 135 – 150. Edinburgh University Press, Edinburgh, 1969.

[24] J. A. Robinson. Automatic Deduction with Hyper-Resolution. *International Journal of Computer Mathematics*, 1:227 – 234, 1965.

[25] J. A. van der Poll. *Automated Support for Set-Theoretic Specifications*. PhD thesis, University of South Africa, June 2000.

[26] J. A. van der Poll and W. A. Labuschagne. Heuristics for Resolution-Based Set-Theoretic Proofs. *South African Computer Journal*, Issue 23:3 – 17, July 1999.

[27] A. Voronkov. The Anatomy of Vampire: Implementing Bottom-Up Procedures with Code Trees. *Journal of Automated Reasoning*, 15(2):237 – 265, 1995.

[28] J. B. Wordsworth. *Software Development with **Z***. International Computer Science Series. Addison-Wesley, 1992. A Practical Approach to Formal Methods in Software Engineering.

[29] L. Wos. *Automated Reasoning: 33 Basic Research Problems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.

[30] L. Wos. The Resonance Strategy. *Computers and Mathematics with Applications*, 29(2):133 – 178, February 1995. (Special issue on Automated Reasoning).

[31] L. Wos. Programs that Offer Fast, Flawless, Logical Reasoning. *Communications of the ACM*, 41(6):87 – 95, June 1998.

[32] L. Wos, D. Carson, and G. Robinson. Efficiency and Completeness of the Set of Support Strategy in Theorem Proving. *Journal of the Association for Computing Machinery*, 12(4):536 – 541, October 1965.

[33] L. Wos and G. W. Pieper. The Hot List Strategy. *Journal of Automated Reasoning*, 22(1):1 – 44, 1999.