The SAD System: Deductive Assistance in an Intelligent Linguistic Environment

Alexander Lyaletski, Andrei Paskevich, Konstantin Verchinine

Abstract— **Formal methods are widely used in the computer science community. Formal verification and certification is an important component of any formal approach. Such a work can not be done by hand, hence the software that can do a part of it is rather required. The verification methods are often based on a deductive system and "verify" means "prove". Corresponding software is called proof assistant.**

We describe in this paper the System for Automated Deduction (SAD): its architecture, input language, and reasoning facilities. We show how to use SAD as a proof assistant. We outline specific features of SAD — a handy input language, powerful reasoning strategy, opportunity to use various low level inference engines. Examples and results of some experiments are also given.

I. Introduction

The idea to use a formal language along with formal symbolic manipulations to solve complex "common" problems, already appeared in G. W. Leibniz's writings (1685). The idea seemed to obtain more realistic status only in the early sixties when first theorem proving programs were created [1]. It is worth noting how ambitious was the title of Wang's article! Numerous attempts to "mechanize" mathematics led to less ambitious and yet more realistic idea of "computer aided" mathematics as well as to the notion of "proof assistant" — a piece of software that is able to do some more or less complex deductions for you. Usually one has in mind either long but routine inferences or a kind of case analysis with enormously large number of possible cases. Both situations are embarrassing and "fault intolerant" for humans.

A great number of projects proclaimed to be proof assistants have been started since the beginning of seventies. Most of them are already dead, some of them were stopped and only a small number of them have survived. Typically, a proof assistant has to put at user's disposal at least two things: a formal language to describe the domain the user is interested in and a theorem proving program to automatize deduction. Historically, the first sustained project to do this was N. G. de Bruijn's Automath Project [2].

Actually, one can clearly observe three "pure" branches in the domain of proof assistance.

The first one, the mainstream, is aimed at formal verification and analysis of *imperative* or *executable* texts — programs, software packages, hardware specifications, various protocol families. Domain description language is the language of typed λ-terms, deduction system (or logical framework) is based on a type theory, theorem proving procedure is often built on top of term rewriting facilities. Well known representatives are Coq [3], Isabelle (in particular Isabelle/HOL [4]), MetaPRL (successor to NuPRL) [5], PVS [6], ACL2 (successor to Nqthm) [7].

A. Lyaletski is with the Faculty of Cybernetics, Kyiv National Taras Shevchenko University, Ukraine (e-mail: lav@unicyb.kiev.ua)

A. Paskevich and K. Verchinine are with the Math-Info Department, Paris 12 "Val de Marne" University, France (e-mail: verko@logique.jussieu.fr)

The second branch consists of computer algebra systems. Widely known REDUCE [8], Mathematica [9], Maple [10], AXIOM [11] et al. are among them.

The third one deals with *declarative*, usually mathematical, texts and is aimed at "classical" proof verification. Domain description language is mostly built on the top of usual one- or multisorted first-order language augmented with some navigation means (one needs to formalize *texts*, not only isolated sentences). Logical framework is based on some complete and sound calculus, theorem proving procedure implements a proof search method for above mentioned calculus (rather often resulution-like). Mizar [12] (firstly mentioned in [13]) is probably the oldest active inhabitant here. Other examples are LP, the prover for the LARCH system [14], and IMPS [15]. (Note that the underlying logic of the latter project is an extension of the first-order classical logic that admits partial functions). The SAD system falls into this category, too.

There are projects that do not fit any of "pure" classes above. They try to integrate a broad range of automated proof methods and even more: proof planning facilities, specific proof methods (e.g. reasoning by analogy), lemma generation, etc. Those are for instance Omega [16], Theorema [17], Isabelle/Isar [18].

An interesting comparison of some of existent mathematical assistants was done in [19].

Let us also note that almost any of the above listed systems provides a kind of a knowledge base in which some basic preliminaries as well as results of past experiences are accumulated. The elements of the knowledge base may be freely reused that makes subsequent proofs more concise and easier to build/verify automatically. The SAD system contains no such base, so the input text to SAD must be "self-contained".

The SAD project is the continuation of a project initiated by academician V. Glushkov at the Institute for Cybernetics in Kiev more than 30 years ago [20]. The title of the original project was "Evidence Algorithm" and its goal was to formalize what is "evident" in mathematics. Axioms are evident, everything that can be deduced from axioms in one step is evident, everything that can be deduced from what is already evident in one step is evident and so on. The question was: provided a computer program which makes deductions, is it possible to obtain something that is evident to the program but is not evident anymore to a human mathematician? From the practical viewpoint such a system could help a working mathematician to verify long and tiresome but routine reasonings. To implement that idea, three main components had to be developed: an inference engine (we call it *prover* below) that implements the basic level of evidence, an extensible collection of tools (we call it *reasoner*) to reinforce the basic engine, and a formal input language which must be close to natural mathematical language and easy to use. The work on the "Evidence Algorithm" was successfully started in the beginning of the seventies, then interrupted and afterwards restored about six years ago. Actually, a working version of the SAD system exists and is available online [21], [22], [23], [24]. In a general setting, SAD may be positioned as a declaration style proof verifier that accepts input texts written in the special formal language ForTheL [25], [23], uses an automated first-order prover as the basic inference engine and possesses an original reasoner (which includes, in particular, a powerful method of definition expansion). As a result, rather "coarse grained" proofs can be successfully processed by SAD.

II. SAD: What does it do?

Generally speaking, SAD verifies the correctness of a given input ForTheL text which, like any usual mathematical text, consists of definitions, assumptions, affirmations, theorems, proofs, etc. Figure 1 gives an idea of what ForTheL text looks like.

What does "correctness" of such an object mean? We distinguish three types of correctness of a ForTheL text: syntactical, ontological and logical.

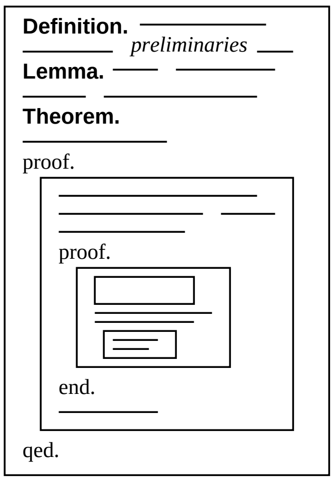

Fig. 1. ForTheL text's structure

Syntactical correctness (well-formedness) is checked by the parser and is a necessary condition for any further actions.

Ontological correctness means that the text in question contains no occurrence of a symbol (constant, function, notion or relation) that comes from nowhere. Every such symbol must be either a signature symbol or introduced by a correct instance of a definition.

Logical correctness is imposed on particular affirmations in the text: theorems, lemmas, intermediate statements in proofs. Any such affirmation must be deducible from its logical predecessors.

Ontological and logical correctness are, to some extent, independent. It is quite obvious that an ontologically correct text may contain false affirmations. Also, an ontologically incorrect text may appear to be logically correct: e.g., we may not know anything about the relation P and the constant c, yet prove $(P(c) \supset P(c))$. Nevertheless, we prefer to require an input text to be ontologically correct for the following reasons. First, ontological verification helps to indicate flaws in your formalization (similarly to type checking in programming languages). Second, during ontological verification the system obtains some important knowledge about the text, which will be used later in logical verification.

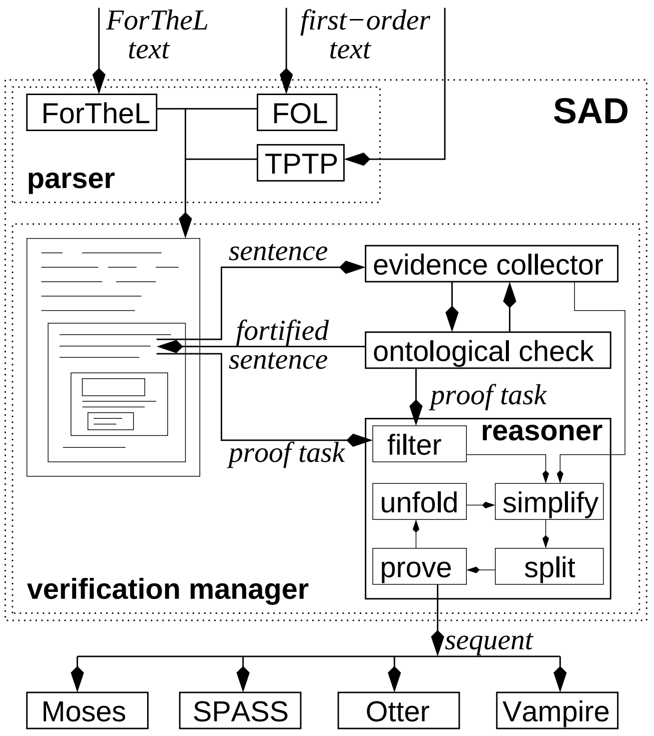

Fig. 2. Architecture of SAD

III. SAD: How does it do that?

Look at Figure 2. All the principal components of the SAD system are shown there. Let us consider some of them in more details.

Parser accepts a ForTheL text, checks its syntactical correctness and converts the text into a normalized form that will be convenient for further processing (e.g. all synonyms are replaced with their canonical representatives).

Verification manager makes her round through the normalized text sentence by sentence. Every sentence is first sent to the "evidence collector" which accumulates so-called *term properties* for the term occurrences in the sentence.

Term properties are literals that tell us something important about a given term occurrence. A literal (i.e. an atomic formula or its negation) L is considered to be a property of a term t in a context Γ (usually, a set of logical predecessors of a given occurrence of t in the text), whenever t is a subterm of L and L is deducible in Γ. The most important purpose of term properties is to hold information about term "types", which is usually expressed by an atomic statement of the form "t `is a` $\langle$*notion*$\rangle$". Some simple properties, like nonemptiness, are highly useful, too.

Note that the evidence collector does not apply the reasoning facility of SAD to check the deducibility of a candidate literal. Instead, there is a simple syntactical procedure that scans the context of a given occurrence and checks what can be "easily" inferred (by series of instantiations and *modus ponens*) from the properties already known. Consider a simple example: let S be declared as a set of integer numbers and x be declared as an element of S. Then, anywhere in view of these declarations, the term $-x$ will be known to be an integer.

Fortified with the found properties, the sentence is passed through the ontological checker. Then, if the sentence is an affirmation to be proved, the manager forms a kind of sequent (we call it *proof task*) and sends it to the *reasoner*. Note that the ontological checker may also resort to the reasoner in order to find whether the guards of a given definition or signature extension are satisfied.

Reasoner can be viewed as a kind of automated heuristic based prover, supplied with a collection of proof task transformation rules. This collection is not intended to form a complete logic calculus. The purpose of the reasoner is not to find the entire proof on its own, but rather to simplify inference search for the *background prover*. The latter is a combinatorial automated prover in classical first-order logic, whose duty is to complete the proofs started by the reasoner. If the background prover fails to find the inference at some instant, the reasoner may continue the proof task transformation or try an alternative way, or just reject the text.

IV. ForTheL language

Like in any other declarative language, the main entity in ForTheL is *sentence*. The syntax of a ForTheL sentence follows the rules of English grammar. Sentences are built of units: statements, predicates, notions (that denote classes of objects) and terms (that denote individual entities). Units are composed of syntactical primitives: nouns which form notions (e.g. "`subset of`") or terms ("`closure of`"), verbs and adjectives which form predicates ("`belongs to`", "`compact`"), symbolic primitives that use a concise symbolic notation for predicates and functions and allow to consider usual quantifier-free first-order formulas as ForTheL statements. Of course, just a little fragment of English is formalized in the syntax of ForTheL.

There are three kinds of sentences in ForTheL: assumptions, selections, and affirmations. Assumptions are statements preceded with the words "`let`" or "`assume that`". They serve to declare variables or to provide some hypotheses for the following text. For example, the following sentences are typical assumptions: "`Let S be a nonempty set.`", "`Suppose that m is greater than n.`". Selections state the existence of representatives of notions and can be used to declare variables, too. Here follows an example of a selection: "`Take an even prime number X.`". Finally, affirmations are just statements: "`If p divides n - p then p divides n.`". The semantics of a sentence is determined by a series of transformations that convert a ForTheL statement to a first-order formula — the *formula image*. For example, the formula image of a simple statement "`all closed subsets of any compact set are compact`" is:

$$\begin{aligned}&\forall\, \mathtt{A}\,((\mathtt{A\ is\ a\ set} \wedge \mathtt{A\ is\ compact}) \supset\\ &\quad \forall\, \mathtt{B}\,((\mathtt{B\ is\ a\ subset\ of\ A} \wedge \mathtt{B\ is\ closed}) \supset\\ &\qquad\qquad\qquad\qquad \mathtt{B\ is\ compact}))\end{aligned}$$

ForTheL sentences form sections: top-level ones (axioms, definitions, theorems) and low-level ones (proofs, proof cases, raw subproof blocks). Sections are used to structure the text, they limit the scope of assumptions and variable declarations. The language supports various proof schemes like proof by contradiction, by case analysis, and by general induction.

The last scheme merits special consideration. Whenever a proof by induction is encountered, the parser automatically creates an appropriate induction hypothesis and reformulates the actual statement to be verified. This induction hypothesis mentions some binary relation which we *declare* a well-founded ordering (hence, suitable for induction proofs). Note that we cannot express the very property of well-foundness in ForTheL (since it is essentially a first-order language), and so the correctness of this declaration is unverifiable. We must take it for granted. After that purely syntactical transformation, the proof section and the transformed statement itself can be verified in a first-order setting, and the reasoner of SAD has no need in specific means to build induction proofs.

Is ForTheL practical as a formalization language? Our numerous experiments (not only recent but also the ancient ones) show that rather often the ForTheL text is sufficiently close to the original hand-made one. Look for instance at the following human-written text:

> A partially ordered set U is a *complete lattice* if any set in U has an infimum and a supremum in U.
>
> A function on U is called *isotone* if for all $x, y \in U$, $x \leqslant y$ implies $f(x) \leqslant f(y)$.
>
> Theorem (Tarski). *Let U be a complete lattice and f, an isotone function on U. The set of all fixed points of f is a complete lattice.*
>
> *Proof.* Let S be the set of fixed points of f. Consider an arbitrary set $T \subseteq S$, possibly empty. Let us show that T possesses a supremum in S (the proof for infimum is quite similar).
>
> Consider the set Q of all the upper bounds x of T such that $f(x) \leqslant x$. Since U is a complete lattice, there exists an infimum q of Q.
>
> First, $f(q)$ is a lower bound of Q. Indeed, for any $x \in Q$, we have $q \leqslant x$, hence $f(q) \leqslant f(x) \leqslant x$.
>
> Second, $f(q)$ is an upper bound of T. Indeed, any $x \in T$ is a lower bound of Q, hence $x \leqslant q$, hence $x = f(x) \leqslant f(q)$.
>
> Therefore, $f(q) = q$. Indeed, $f(q) \leqslant q$, since q is

the infimum of Q. But then $f(f(q)) \leqslant f(q)$, hence $f(q) \in Q$, hence $q \leqslant f(q)$.

So q is a fixed point of f, hence, belongs to S. Consider any upper bound r of T in S. Obviously, r belongs to Q, so $q \leqslant r$. Hence q is the supremum of T in S. □

This text can be translated to ForTheL as follows:

```
Definition DefCLat. A complete lattice
    is a set S such that every subset of S
    has an infimum in S and a supremum in S.

Definition DefIso.
  f is isotone iff for all x,y << Dom f
    x <= y  =>  f(x) <= f(y).

Theorem Tarski.
  Let U be a complete lattice.
  Let f be an isotone function on U.
  Let S be the set of fixed points of f.
  S is a complete lattice.
Proof.
  Let T be a subset of S.

  Let us show that T has a supremum in S.
    Take Q = { x << U | f(x) <= x and
      x is an upper bound of T in U }.

    Take an infimum q of Q in U.
    f(q) is a lower bound of Q in U.
    f(q) is an upper bound of T in U.
    q is a fixed point of f.
    Thus q is a supremum of T in S.
  end.

  Let us show that T has a supremum in S.
    ### SAD does not support proofs by analogy,
    ### so we have to repeat here our reasoning.
    ...
  end.
qed.
```

So in that example the translation is almost straightforward. Certainly, it is not always the case but there is necessarily a fruitful side-effect of the hard formalization work — one understands much better the subject after doing it.

V. Reasoner

It was already noted that the reasoner applies a collection of transformation rules to the given proof task. The richer the reasoner's collection of transformation rules is, the more complex proof tasks it can fulfil, the more coarse-grained and terse mathematical texts can be verified by the system. The simplest reasoner would just send the received proof task to the background prover — that was the point where the development of SAD has started.

At present, the capabilities of the reasoner are as follows: propositional goal splitting, formula simplification with respect to accumulated term properties, simple filtering of premises according to explicit references in the text, incremental definition expansion.

The reasoner of SAD uses term properties to simplify goal formulas and formulas which arise from definition expansion: any literal that appears to hold as a property of some of its subterms can be replaced by logical constant "truth" (indeed, it can be deduced from the current context and, hence, is redundant). Similarly, a literal can be replaced by "false", if its complement occurs among the properties of its subterms.

Definitions, taken as straight logical formulas, are quite difficult to handle in an automated prover: they tend to drastically extend the search space and produce hardly catchable redundancies. Besides, it costs additional inference steps to eliminate definition guards. Therefore, we prefer to hide definitions from the prover and apply them in the reasoner (the preliminary check of ontological correctness comes in handy here). Surely, the reasoner cannot know precisely which occurrences must be unfold, and, since exhaustive unfolding up to the signature symbols is obviously impractical, some strategy of expansion must be provided.

In SAD, we have considered several expansion strategies. The one used in the current version of the system is as follows. At first, we try to prove the goal without any expansion at all. If the prover fails to find the proof, we choose some "promising" occurrences in the neighborhood of the goal or in the goal itself. What is promising and what is not, is decided by several heuristics taking into account the level of a symbol in the hierarchy of definitions and the properties of the terms occurring in the goal formula. After the first wave of expansion is made, we split and simplify the obtained proof task and call the prover again. Where it fails, we apply our strategy again to choose and unfold some new occurrences, and the whole process is repeated.

This strategy does not use backtracking, that is, we unfold definitions incrementally, wave after wave, and never return to previous states of the proof task. In order to avoid drowning of important information (which can occur if the reasoner unfolds some occurrences that should rather have rest and participate in the proof "as is"), we apply definitions in a non-destructive way, preserving the original symbol at the side of the inserted formula.

VI. Background prover

The native background prover of SAD, called Moses, is based on a special goal-driven sequent calculus [22], [23]. The prover explores the search space using bounded depth-first search with iterative deepening and backtracking, it uses constraints and folding-up [26] to increase the efficiency of search. In order to provide SAD with equality handling, Moses implements a variation of Brand's modification method [27]. The original notion of admissible substitution used in the calculus allows to preserve the initial signature of the task so that accumulated unification problems (sets of equations) can be sent to a specialized solver, e.g. an external computer algebra system (this functionality is not yet implemented).

We call Moses native, because the background prover is supposed to be independent from SAD by design, so that an external theorem proving system like Otter [28], SPASS [29], or Vampire [30] could be used. Note that this capability of SAD provides us with a (yet another) scale to compare automated theorem provers: trying them on relatively simple problems in complex and heavily redundant contexts rather than on hard problems with a preadjusted set of relevant premises (mostly the case for problems in the famous TPTP library [31]).

In our experiments with the three aforementioned provers, the best results were obtained with SPASS. This is due, in particular, to its original technique of handling sort-like information, which abounds in mathematical texts.

VII. Experiments

In the course of development of SAD, we have conducted a number of essays on formalization and verification of non-trivial mathematical results:

- Ramsey's Finite and Infinite theorems (as presented in [32]).
- Stability of a refinement relation over a number of operations on program specifications [33].
- Some properties of finite groups (as presented in [34]).
- Cauchy-Bouniakowsky-Schwarz inequality.
- The square root of a prime number is irrational: 30 statements in preliminaries (integer numbers), 5 definitions, 7 lemmas, about 50 sentences in the proof of the main lemma (any prime dividing a product divides one of the factors), 10 sentences in the proof of the theorem (see [23] for detailed explanation of this experiment).
- Chinese remainder theorem and Bezout's identity in terms of abstract rings: 25 statements in preliminaries (ring axioms, operations on sets), 7 definitions (ideal, principal ideal, greatest common divisor, etc), 3 lemmas, 8 sentences in the proof of CRT, about 30 sentences in the proof of Bezout's identity.
- Tarski's fixed point theorem (cited above): 11 statements in preliminaries (ordered sets), 7 definitions (upper and lower bounds, supremum, infimum, complete lattice, isotone function, fixed point), 2 lemmas, 18 sentences in the proof of the theorem.

The texts listed above were written in ForTheL and automatically verified in SAD (using SPASS as the background prover). This work have taught us many important lessons. To mention some:

- Formalization style is critical: the choice of symbols to introduce in definitions, the choice of preliminary facts, and even the way a proof is structured may decide whether the text will be verified or not.
- It is highly desirable to comprehend the proofs before writing them in ForTheL. The SAD system may succeed to fulfil the gaps in a well thought-out reasoning, but it will not invent one for you.
- In most cases, the background prover finds the proof in three seconds — or does not find it at all.

VIII. Conclusion

Certainly, we could not give here a detailed description of all nice features of SAD. SAD is a powerful system and its power lies in its reasoning facility. Experiments show that, for example, the specific strategy of definition processing contributes a lot to the success of the whole verification process. If we use definitions straightforwardly — convert them into formula images and add the corresponding premises to the sequent that goes into a prover — we have no chance to verify the proof of Tarski fixed-point theorem as it is formulated above, even when the winner of CASC competitions is chosen as the background prover.

SAD is not a perfect system (if any!). One can easily see how it may be improved and developed. Our research and implementation plans are: extend ForTheL and SAD with some means to talk and reason about second-order objects (functions, vectors, sequences) and operations on them; provide users with a facility to implement custom strategies for the reasoner; develop and implement a mathematical library of SAD to accumulate verified portions of mathematical knowledge and to support further (deeper) advances in formalization.

Where can the SAD system be useful? In any domain where precise mathematical style formalism is appreciated as the means of problem description. Note that the problem formalization is always the hardest part of the whole work. Right formalization is a 80% guarantee of a successful verification.

Sometimes we wonder: why are we doing all that? Yes, it is interesting, what else? It seems that at least two questions are extremely attractive for those who live in the domain of automated deduction. First, is the mathematical creative work formalizable? Are there any limits to formalization process? And second, how far can the power of computers go? Ad infinitum?

References

[1] Hao Wang, "Towards mechanical mathematics," *IBM J. of Research and Development*, vol. 4, pp. 2–22, 1960.

[2] N. G. de Bruijn, "The mathematical language AUTOMATH, its usage and some of its extensions," in *Symposium on Automatic Demonstration*, M. Laudet, D. Lacombe, L. Nolin, and M. Schützenberger, Eds. 1970, vol. 125 of *Lecture Notes in Computer Science*, pp. 29–61, Springer-Verlag.

[3] B. Barras, S. Boutin, C. Cornes, J. Courant, J.C. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner, "The Coq proof assistant reference manual — version v6.1," Tech. Rep. 0203, INRIA, 1997.

[4] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, vol. 2283 of *Lecture Notes in Computer Science*, Springer-Verlag, 2002.

[5] Jason Hickey, Aleksey Nogin, Robert L. Constable, Brian E. Aydemir, Eli Barzilay, Yegor Bryukhov, Richard Eaton, Adam Granicz, Alexei Kopylov, Christoph Kreitz, Vladimir N. Krupski, Lori Lorigo, Stephan Schmitt, Carl Witty, and Xin Yu, "MetaPRL — modular logical environment," in *Theorem Proving in Higher Order Logics: 16th International Conference, TPHOLs 2003*, David Basin and Burkhart Wolff, Eds. 2003, vol. 2758 of *Lecture Notes in Computer Science*, pp. 287–303, Springer-Verlag.

[6] Sam Owre, John M. Rushby, and Natarajan Shankar, "PVS: a prototype verification system," in *Automated Deduction: 11th International Conference, CADE-11*, Deepak Kapur, Ed. 1992,

vol. 607 of *Lecture Notes in Computer Science*, pp. 748–752, Springer-Verlag.

[7] Matt Kaufmann and J Strother Moore, "An industrial strength theorem prover for a logic based on Common Lisp," *IEEE Transactions on Software Engineering*, vol. 23, no. 4, pp. 203–213, Apr. 1997.

[8] Anthony C. Hearn, "REDUCE: A user-oriented interactive system for algebraic simplification," in *Interactive Systems for Experimental Applied Mathematics*, M. Klerer and J. Reinfelds, Eds., pp. 79–90. Academic Press, New York, 1968.

[9] Stephen Wolfram, *The Mathematica Book, Fifth Edition*, Wolfram Media, Inc., 2003.

[10] Frank Garvan, *The Maple Book*, CRC Press, 2001.

[11] R. D. Jenks and R. S. Sutor, *Axiom: The Scientific Computation System*, Springer-Verlag, 1992.

[12] Andrzej Trybulec and Howard Blair, "Computer assisted reasoning with Mizar," in *Proc. 9th International Joint Conference on Artificial Intelligence*, Aug. 1985, pp. 26–28.

[13] A. Trybulec, "Informationslogische Sprache Mizar," Dokumentation-Information 33, TU Ilmenau, 1977.

[14] Stephen J. Garland and John V. Guttag, "A guide to LP, the Larch Prover," Tech. Rep., MIT Laboratory for Computer Science, Dec. 1991.

[15] W. M. Farmer, J. D. Guttman, and F. J. Thayer, "IMPS: an interactive mathematical proof system (system description)," in *Automated Deduction: 10th International Conference, CADE-10*, M. E. Stickel, Ed. 1990, vol. 449 of *Lecture Notes in Computer Science*, pp. 653–654, Springer-Verlag.

[16] Christoph Benzmüller, Lassaad Cheikhrouhou, Detlef Fehrer, Armin Fiedler, Xiaorong Huang, Manfred Kerber, Michael Kohlhase, Karsten Konrad, Andreas Meier, Erica Melis, Wolf Schaarschmidt, Jörg H. Siekmann, and Volker Sorge, "ΩMEGA: Towards a mathematical assistant," in *Automated Deduction: 14th International Conference, CADE-14*, William McCune, Ed. 1997, vol. 1249 of *Lecture Notes in Computer Science*, pp. 252–255, Springer-Verlag.

[17] Bruno Buchberger, Tudor Jebelean, Franz Kriftner, Mircea Marin, Elena Tomuta, and Daniela Vasaru, "A survey of the Theorema project," in *ISSAC'97 — Proc. International Symposium on Symbolic and Algebraic Computation*, Wolfgang Küchlin, Ed., Maui, Hawaii, USA, 1997, pp. 384–391, ACM Press.

[18] Markus Wenzel, "Isar — a generic interpretative approach to readable formal proof documents," in *Theorem Proving in Higher Order Logics: 12th International Conference, TPHOLs'99*. 1999, vol. 1690 of *Lecture Notes in Computer Science*, pp. 167–184, Springer-Verlag.

[19] Freek Wiedijk, Ed., *The Seventeen Provers of the World*, vol. 3600 of *Lecture Notes in Computer Science*, Springer-Verlag, 2006.

[20] V. M. Glushkov, "Some problems of automata theory and artificial intelligence (in Russian)," *Kibernetika*, vol. 2, pp. 3–13, 1970.

[21] A. Lyaletski, K. Verchinine, and A. Paskevich, "On verification tools implemented in the System for Automated Deduction," in *Proc. 2nd CoLogNet Workshop on Implementation Technology for Computational Logic Systems (ITCLS'2003)*, Pisa, Italy, Sept. 2003, pp. 3–14.

[22] Alexander Lyaletski, Andrey Paskevich, and Konstantin Verchinine, "Theorem proving and proof verification in the system SAD," in *Mathematical Knowledge Management: Third International Conference, MKM 2004*, Andrea Asperti, Grzegorz Bancerek, and Andrzej Trybulec, Eds. 2004, vol. 3119 of *Lecture Notes in Computer Science*, pp. 236–250, Springer-Verlag.

[23] A. Lyaletski, A. Paskevich, and K. Verchinine, "SAD as a mathematical assistant — how should we go from here to there?," *J. of Applied Logic*, (to appear).

[24] "The Evidence Algorithm project," http://ea.unicyb.kiev.ua.

[25] K. Vershinin and A. Paskevich, "ForTheL — the language of formal theories," *International Journal of Information Theories and Applications*, vol. 7, no. 3, pp. 120–126, 2000.

[26] R. Letz and G. Stenz, "Model elimination and connection tableau procedures," in *Handbook for Automated Reasoning*, A. Robinson and A. Voronkov, Eds., vol. II, pp. 2017–2116. Elsevier Science, 2001.

[27] D. Brand, "Proving theorems with the modification method," *SIAM Journal of Computing*, vol. 4, pp. 412–430, 1975.

[28] William McCune, "Otter 3.0 reference manual and guide," Tech. Report ANL-94/6, Argonne National Laboratory, Argonne, USA, 1994.

[29] Christoph Weidenbach, Uwe Brahm, Thomas Hillenbrand, Enno Keen, Christian Theobald, and Dalibor Topic, "SPASS version 2.0," in *Automated Deduction: 18th International Conference, CADE-18*, Andrei Voronkov, Ed. 2002, vol. 2392 of *Lecture Notes in Computer Science*, pp. 275–279, Springer-Verlag.

[30] Alexandre Riazanov and Andrei Voronkov, "The design and implementation of VAMPIRE," *AI Communications*, vol. 15, no. 2–3, pp. 91–110, 2002.

[31] Geoff Sutcliffe, Christian B. Suttner, and Theodor Yemenis, "The TPTP problem library," in *Automated Deduction: 12th International Conference, CADE-12*, Alan Bundy, Ed. 1994, vol. 814 of *Lecture Notes in Computer Science*, pp. 252–266, Springer-Verlag, see also http://tptp.org.

[32] R. L. Graham, *Rudiments of Ramsey Theory*, AMS, 1981.

[33] Amel Mammar, *Un environnement formel pour le développement d'application bases de données*, Ph.D. thesis, Conservatoire National des Arts et Métiers, France, 2002.

[34] J.-P. Serre, *Cours d'Arithmetique*, Presses Universitaires de France, 1970.