On verification tools implemented in the System for Automated Deduction

Alexander Lyaletski[1], Konstantin Verchinine[2], and Andrey Paskevich[1,2]

[1] Faculty of Cybernetics, Kyiv National Taras Shevchenko University, Kyiv, Ukraine (e-mail: lav@unicyb.kiev.ua, andrey@raptor.kiev.ua)
[2] Math-Info Department, Paris 12 University, Creteil, France (e-mail: verko@logique.jussieu.fr)

Abstract. Among the tasks of the Evidence Algorithm programme, the verification of formalized mathematical texts is of great significance. Our investigations in this domain were brought to practice in the last version of the System for Automated Deduction (SAD). The system exploits a formal language to represent mathematical knowledge in a "natural" form and a sequential first-order formalism to prove statements in the frame of a self-contained mathematical text. In the paper, we give an overview of the architecture of SAD and its verification tools. In order to demonstrate the work of SAD, a sample verification session is examined[3].

1 Introduction

In this paper, the last investigations in the frame of a research programme called Evidence Algorithm, EA, are considered from the point of view of their use for verification of mathematical texts. Evidence Algorithm was advanced by Academician V. Glushkov as a programme on investigating for "doing" mathematics in wide sense [1]. Its main objective was to help to working mathematicians in constructing and proving/verifying automatically long, but in some sense "evident" proofs. For this purpose, V. Glushkov proposed to focus researchers' attention on problems from the following fields: formalized languages for presenting mathematical texts in the form most appropriate for a user; a formal notion of evolutionary developing computer-made proof step; EA information environment that has influence on the evidence of a proof step; and man-assisted search for a proof.

As a result, the System for Automated Deduction, SAD, appeared [5, 6]. Its current implementation is oriented to perform the following transformations.

Firstly, a mathematical text under consideration is formalized with the help of ForTheL language, which is formal on one hand, but is close to a natural language of mathematical publications, on the other hand. That initial ForTheL-text is then translated into a certain first-order output representation.

Secondly, the text is exposed to deductive processing. Three kinds of the deductive processing are distinguished depending on a type of a task to solve:

[3] These investigations are supported by the INTAS-project 2000-447.

– establishing deducibility of a first-order sequent;
– proving the last proposition in an initial ForTheL-text;
– verifying an initial ForTheL-text containing proofs of theorems.

Deduction in SAD is based on an original sequent calculus GD [5]. A sequent formalism was chosen as sequent methods reflect better the "natural" way of reasoning than resolutional ones. In particular, neither skolemization nor clausification are required. The efficiency of our proof search procedure is gained due to the goal-drivenness of the calculus and a special quantifier handling technique.

Because of the lack of space we do not describe here the grammar of ForTheL nor the inference rules of GD. The description of the former two kinds of deductive processing can be found in [5–7]. Below, we focus our attention on the problem of text verification, as it caused essential extensions of the architecture and the tools of SAD.

The on-line version of SAD is available at http://ea.unicyb.kiev.ua

2 Architecture of SAD

The current implementation of the system SAD can act as a verifier of formal texts. During any verification session, SAD solves three tasks: translating an input ForTheL-text to its internal representation; determining a sequence of affirmations to be verified (proved); searching for logical inference of every goal from its logical predecessors in the text.

In this connection, there exist four main modules in SAD: [ForTheL], [FOL], [Reason], and [Moses]. Below we give the brief description of each of them.

Modules [ForTheL] *and* [FOL]. The modules [ForTheL] and [FOL] perform parsing of ForTheL-texts and first-order texts, respectively. Each of these modules converts an input text to a corresponding internal representation. This conversion preserves the structure of an initial text and translates phrases into an ordered set of first-order formulas. The formulas obtained from ForTheL-phrases can contain annotations, which are used in proof search. For example, the parser [ForTheL] singles out the guards in theorems and axioms, and the "head" atom in predicate definitions. When an input text is written in the first-order language, no translation is needed, and [FOL] merely eliminates a "syntactical sugar".

Module [Reason]. This module runs a verification cycle and gives tasks to the prover of SAD. An algorithm which defines this cycle is given below.

A ForTheL-text translated by a parser module is taken as the input. It is represented with a list of *text units*: phrases (affirmations or assumptions) stored as first-order formulas, and sections (axioms, theorems, proofs, proof cases) stored as lists of inner units.

The state of verification process is determined by a *pointer* to a text unit being processed and by a *context*, the list of units which are logical predecessors of the unit under the pointer. To verify compound text units, subordinate verification cycles are started. These cycles are called *inner*, with respect to the *outer* cycles launching them.

1. Set the pointer to the first unit in the given text. Take the context from the outer verification cycle, if any; set it to the empty list, otherwise.
2. Let U be the text unit under the pointer. Depending on the kind of U (affirmation, assumption, theorem, axiom, proof, etc), decide whether U is a goal (i.e. whether U should be verified/proved). If so, go to the next step; otherwise, jump to the step 6.
3. If U is a section, start an inner verification cycle for the contents of U. When the subcycle is finished, jump to the step 6. Otherwise (when U is a phrase), go to the next step.
4. Try to simplify the goal under consideration. At this point, various heuristics and tactics can be applied. For example, if the goal is a conjunction of two variable-independent formulas, then split it into two subgoals and repeat this step for each of them.
5. Launch [Moses], the prover of SAD, for every subgoal that can't be simplified further. If [Moses] cannot find a proof, and there are no alternative subgoals, the whole verification process stops with the diagnostics "verification failed". Otherwise, consider the next subgoal. When all the subgoals of U are proved, U is considered to be verified.
6. If U is not the last unit in a section (or in the whole text), append it to the context, shift the pointer to the next unit, and jump to the step 2. If U is the last unit in a section, finish the current verification cycle and return to the outer one. If U is the last unit in the text, finish the verification with the diagnostics "verification successful".

Module [Moses]. This module is intended for proof search. Before starting the inference search routine, [Moses] prepares a deductive environment for its work: it finds disjunctively connected literals, determines variable dependencies, fixes the pairs of complementary literals, and so on.

The prover [Moses] is implemented on the base of the calculus GD [5]. The inference search procedure looks through the search space using bounded depth first search with iterative deepening and backtracking. In order to increase the efficiency of inference search, [Moses] uses special constraints and folding-up technique. Also, solutions of accumulated systems of equations are found only in reasonable cases. Recently, a special heuristics for definition application was adopted.

In order to provide SAD with equality handling capabilities, [Moses] implements a certain variation of Brand's modification method [8].

Note that since the calculus GD does not change initial premises, the prover can perform operations with a tree of literal goals instead of a tree of sequents. This drastically reduces the need in computational resources.

Due to the absence of preliminary skolemization, the prover of SAD is capable to use a relevant solver for solving equation systems. The submodule of [Moses] responsible for equation handling acts as a mediator between the prover and external solvers. It checks a substitution found by such a way for admissibility and builds additional equations if necessary. The procedure computing the most general unifier is used as a default equation solver.

3 Language of SAD

Any language intended for representation of formal texts in the EA-style must satisfy the following requirements. Such a language should have formal syntax and semantics. It should permit to write theory axioms, definitions, theorems, and proofs in order to obtain self-contained texts. Also the language should be close to natural one used in mathematical papers to provide a user with a suitable means to create and process mathematical texts in an interactive mode.

The last version of the language ForTheL [2] satisfy well above-mentioned requirements. Let us summarize its peculiarities.

A ForTheL-text is a sequence of sections, phrases, and special constructs like pattern introductors. Phrases are either assumptions (then they begin with "`let`" or "`assume`") or affirmations. Sections may be composed from sections of a lower level and phrases. Typical top-level sections are axioms, definitions, and propositions. Typical sections of lower level are proofs and proof cases.

The grammar of ForTheL-phrases imitates the grammar of English sentences. Phrases are built using nouns, which denote notions (classes) or functions, verbs and adjectives, which denote predicates, and prepositions and conjunctions, which define the logical meaning of a complex sentence. Here is a simple ForTheL-affirmation: "`Every closed subset of every compact set is compact.`"

A current thesaurus can be extended with the help of special top-level constructs called pattern introductors. Below we give a pattern introductor for a certain unary notion: "`[an element/elements of x]`". If a word used in a pattern has several possible forms, you can list them with slashes. One can introduce a new pattern as an alias of some expression of the same kind by using an introductor of the form "`[`*pattern* `@` *meaning*`]`". The only predefined ForTheL-pattern is the predicate pattern "`[x is equal to y]`".

Every affirmation in a text, e.g. the statement of a theorem, can be provided with a proof section. A ForTheL-proof is a sequence of assumptions, affirmations (which may have their own proofs), and proof cases.

When the verifier encounters a proof or a proof case, it starts a separate, "inner" verification cycle. If a section was successfully verified then its *formula image* is appended to the context of the outer verification process. The image of a proof section is used to verify the mere affirmation justified with this proof and is removed from the context then, whereas the image of a proof case section is used up to the end of the outer section.

The image of a section is a formula constructed from the contents of that section. In this formula, assumptions become the antecedents of implications, and affirmations and nested proof cases become conjuncts. Each variable declared inside a section is bound by a universal quantifier in the image. Thus, the image of a section "`case. let X be N. then P. assume H. then Q. end.`" is the formula $(\forall X(N(X) \supset (P \wedge (H \supset Q))))$.

Besides providing a proof, the verification process can be supported with *references*. After an affirmation, one can list the sections (by their labels) to give them a higher priority during the search for a proof of this affirmation.

4 Verification in SAD

Now we consider a simple self-contained ForTheL-text about some properties of natural number ordering defined in a certain subset of Peano arithmetic. This text was verified completely by SAD in less than fifteen seconds on a Pentium IV 1.8 GHz station.

The text will be considered section by section, intermixing fragments of ForTheL with comments and diagnostics of SAD.

The axioms of zero and the successor.

```
[a number] [the zero] [the successor of x]
[x is nonzero @ x is not equal to zero]

Axiom _NatZero.  Zero is a number.

Axiom _NatSucc.  The successor of every number is a number.

Axiom SuccNotZero.  The successor of no number is zero.

Axiom SuccEquSucc.  Let A be a number and B be a number.
    If the successor of A is the successor of B
        then A is equal to B.
```

Draw your attention to the notion "`number`" and to the axioms "`_NatZero`" and "`_NatSucc`". The language ForTheL (in its current implementation) requires every variable to be declared as a representative of some notion. That is why we should introduce an explicit notion for numbers and postulate that the values produced by the functions "`zero`" and "`successor of`" are numbers.

These two axioms are used in almost any proof, so we want the prover to never "lose sight" of them, even in presence of references to other axioms or theorems. In order to single it out, the labels of these axioms begin with underscores.

The above-mentioned fragment will be translated in the following first-order formulas:

_NatZero:	$Number(Zero)$
_NatSucc:	$\forall x_1(Number(x_1) \supset Number(SuccessorOf(x_1)))$
SuccNotZero:	$\forall x_1(Number(x_1) \supset SuccessorOf(x_1) \neq Zero)$
SuccEquSucc:	$\forall A, B((Number(A) \wedge Number(B)) \supset$ $(SuccessorOf(A) = SuccessorOf(B) \supset A = B))$

Introducing addition.

```
[the sum of x and y]

Axiom AddZero.  Let A be a number.
```

```
    The sum of A and zero is equal to A.

Axiom AddSucc. Let A be a number and B be a number.
    The sum of A and the successor of B is equal
        to the successor of the sum of A and B.
```

These two axioms are an example of a primitive recursive definition. Now we are working on further extensions both of the language and of deductive tools of SAD in order to handle such definitions in a direct and efficient way.

Supplementary axioms. So far, SAD cannot handle induction in proofs. That is why we include the following facts as axioms.

```
Axiom ZeroOrSucc.
    Every nonzero number is the successor of some number.

Axiom _NatAdd.
    The sum of every number and every number is a number.

Axiom AssoAdd.
    Let A be a number and B be a number and C be a number.
    The sum of A and the sum of B and C is equal
        to the sum of (the sum of A and B) and C.

Axiom InjAdd.
    Let A be a number and B be a number and C be a number.
    If the sum of A and B is equal to the sum of A and C
        then B is equal to C.

Axiom Diff.  Let A be a number and B be a number.
    There exists a number C such that
        B is the sum of A and C or A is the sum of B and C.
```

The axiom "ZeroOrSucc" is translated as follows:

$$\mathsf{ZeroOrSucc}: \quad \forall x_1((Number(x_1) \wedge x_1 \neq Zero) \supset \exists x_2(Number(x_2) \wedge x_1 = SuccessorOf(x_2)))$$

Introducing ordering. We define the ordering on the natural numbers by means of addition:

```
[x is less than y] [x is greater than y @ y is less than x]

Definition DefLess.  Let A be a number and B be a number.
    A is less than B  iff  B is equal to
        the sum of A and the successor of some number.
```

The formula image of this definition is:

$$\mathsf{DefLess}: \quad \forall A, B((Number(A) \wedge Number(B)) \supset (LessThan(A, B) \equiv \exists x_1(Number(x_1) \wedge B = SumOfAnd(A, SuccessorOf(x_1)))))$$

Irreflexivity proof. We start verification by proving the properties of order for the relation "`less than`". Since the ordering is strict, it must be irreflexive.

```
Theorem NReflLess.  Let A be a number.  A is not less than A.
Proof.
    Assume the contrary.
    Let C be a number such that A is equal to
        the sum of A and the successor of C.
    Then the successor of C is zero (by AddZero,InjAdd).
    We have a contradiction.
qed.
```

In ForTheL-sentences from proof sections, the special statements "`thesis`" and "`contrary`" denote an affirmation that is justified by a proof section under consideration and the negation of this affirmation, respectively. So, the first phrase in the proof under consideration is translated as "`assume not not LessThan(A,A).`".

The second assumption in the proof introduces a number variable `C`. At this moment, SAD does not try to prove the existence of such a number.

In order to verify the whole theorem "`NReflLess`", the verifier must prove three goals: the successor of `C` is zero; this fact contradicts the assumptions made above; the contradiction entails the affirmation of the theorem.

The first goal is not trivial: in order to prove it, we must recall that adding zero does not change the number (axiom "`AddZero`") and that no other number has this property (axiom "`InjAdd`"). We also need to know that the successor of `C` is a number. However, we do not mention the corresponding axiom with the help of an appropriate reference, since it will be taken into account by default. The system SAD gives the following diagnostics for this goal:

```
[Reason] line 85: encountered a goal
[Moses] launch (time limit: 10 sec, initial db: 1, final db: 100)
[Moses] proved in 0 msec -- proof tree nodes: 25 -- proof tree depth: 4
[Moses] inference steps: 450 -- born nodes: 1087 -- depth bound: 4
```

The second line reports the parameters of the proof search session. The prover disposes of ten seconds to find a proof, it starts the search with the depth bound (the maximal height of an inference tree) set to 1, and this bound can be increased up to 100.

The last two lines report the result of the search. The proof was found "almost immediately", the proof tree contains 25 literals and is of height 4. In order to find it, [Moses] made 450 inference steps (expansions and terminations) and produced altogether 1087 nodes. The last depth bound was set to 4.

We do not quote the proof tree here. An example of a proof generated by SAD will be given below.

Note that the proof was found quickly due to the references to the relevant axioms. Without these hints, the proof search would require much more time.

Now, the equality of some successor to zero immediately contradicts the axiom "`SuccNotZero`". The proof should be simple and we do not make hints to [Moses]. You can see, however, that the prover makes a lot of search to find the proof which is indeed trivial (6 literals in the proof tree):

```
[Reason] line 86: encountered a goal
[Moses] launch (time limit: 10 sec, initial db: 1, final db: 100)
[Moses] proved in 5280 msec -- proof tree nodes: 6 -- proof tree depth: 3
[Moses] inference steps: 649366 -- born nodes: 1161172 -- depth bound: 3
```

Such an "inefficiency" can be explained by two considerations. Firstly, we are proving not a concrete formula but an abstract contradiction. Therefore [Moses] tries to prove the negation of each formula in the current section as the initial goal in the proof before increasing the depth bound. Secondly, in problems containing equality, much more inferences are possible in comparison with non-equality problems.

Finally, we must prove that the obtained contradiction entails the affirmation of the theorem. In order to prove it, the prover must check that, besides the negation of the thesis, every assumption in the proof is satisfiable. At this moment, the existence of `C` has to be shown. Note that the goal under consideration is "`A is not less than A`" which occurs above the previous goals in the ForTheL-text.

```
[Reason] line 80: encountered a goal
[Moses] launch (time limit: 180 sec, initial db: 1, final db: 100)
[Moses] proved in 10 msec -- proof tree nodes: 26 -- proof tree depth: 3
[Moses] inference steps: 938 -- born nodes: 1917 -- depth bound: 3
```

Transitivity proof. The associativity of addition implies the transitivity property:

```
Theorem TransLess.
    Let A be a number and B be a number and C be a number.
    Assume A is less than B and B is less than C.
    Then A is less than C.
Proof.
    Let M be a number and N be the successor of M.
    Let P be a number and Q be the successor of P.
    Assume the sum of A and N is equal to B.
    Assume the sum of B and Q is equal to C.
    Let S be the sum of N and Q.
    S is the successor of the sum of N and P (by AddSucc).
    The sum of A and S is equal to C (by AssoAdd).
    Hence the thesis (by DefLess).
qed.
```

Verification of this theorem consists of proving of four goals. The diagnostics of SAD for this fragment of the text follows below.

```
[Reason] line 101: encountered a goal
[Moses] launch (time limit: 10 sec, initial db: 1, final db: 100)
[Moses] proved in 50 msec -- proof tree nodes: 26 -- proof tree depth: 5
[Moses] inference steps: 6222 -- born nodes: 11769 -- depth bound: 5
[Reason] line 102: encountered a goal
[Moses] launch (time limit: 10 sec, initial db: 1, final db: 100)
[Moses] proved in 10 msec -- proof tree nodes: 39 -- proof tree depth: 4
[Moses] inference steps: 1605 -- born nodes: 3206 -- depth bound: 4
[Reason] line 103: encountered a goal
[Moses] launch (time limit: 10 sec, initial db: 1, final db: 100)
[Moses] proved in 280 msec -- proof tree nodes: 35 -- proof tree depth: 5
[Moses] inference steps: 35685 -- born nodes: 85354 -- depth bound: 5
[Reason] line 94: encountered a goal
[Moses] launch (time limit: 10 sec, initial db: 1, final db: 100)
[Moses] proved in 1560 msec -- proof tree nodes: 50 -- proof tree depth: 4
[Moses] inference steps: 233910 -- born nodes: 529587 -- depth bound: 4
```

Though the last statement in the proof section is "`thesis`", the prover constructs a quite big proof tree (50 nodes) in order to accomplish the proof of the whole theorem. The reason is that the "`thesis`" in the line 103 was proved under assumption of existence of numbers `M`, `N`, `P`, `Q` with the needed properties. The satisfiability of this assumption is proved in the last proof session for the line 94.

Asymmetry proof. The asymmetry is an immediate corollary of the irreflexivity and transitivity properties. There is no need in an explicit proof: [Moses] proves the lemma in less than one millisecond.

```
Lemma ASymmLess.
    Let A be a number and B be a number less than A.
    Then A is not less than B.
```

Totality proof. To verify the theorem below, SAD launches seven proof search sessions. Altogether, the verification is completed in 3730 msec with 564302 inference steps made and 1022099 tree nodes produced.

```
Theorem TotalLess.
    Let A be a number and B be a number not equal to A.
    Then A is less than B or B is less than A.
Proof.
    Let C be a number such that
        A is the sum of B and C or B is the sum of A and C.
    If C is zero then B is equal to A.
    Hence C is the successor of some number.
    If B is the sum of A and C then A is less than B.
    Then A is the sum of B and C or A is less than B.
```

```
    If A is the sum of B and C then B is less than A.
    Hence the thesis.
qed.
```

Introducing multiplication. Now we define multiplication of natural numbers. The statements "_NatMul" and "Monot" are provable by induction in Peano arithmetic. We introduce them as axioms.

```
[the product of x and y]

Axiom MulZero.  Let A be a number.
    The product of A and zero is equal to zero.

Axiom MulSucc.  Let A be a number and B be a number.
    The product of A and the successor of B is equal
        to the sum of A and the product of A and B.

Axiom _NatMul.
    The product of every number and every number is a number.

Axiom Monot.
    Let A be a number and B be a number and C be a number.
    Assume A is greater than B and C is nonzero.
    Then the product of A and C is greater
        than the product of B and C.
```

Proof by case examination. The following theorem reformulates the axiom "Monot" for the non-strict ordering. We consider this theorem in order to demonstrate an example of a more complex ForTheL-proof.

```
Theorem MonotNSt.
    Let A be a number and B be a number and C be a number.
    Assume A is not less than B.
    Then the product of A and C is not less
        than the product of B and C.
Proof.
    Case 1.
        Assume A is equal to B or C is zero.
        Then the product of A and C is equal
            to the product of B and C (by MulZero).
        Hence the thesis (by NReflLess).
    end.
    Case 2.
        Assume A is not equal to B and C is nonzero.
        Hence the thesis (by Monot,TotalLess,ASymmLess).  # <--
    end.
qed.
```

Below we quote a proof tree generated by [Moses] for the goal marked above by the arrow. The tree is presented in a simplified form: the subproofs of membership of the class of numbers as well as the applications of the reflexivity axiom $\forall x(x = x)$ are omitted. That is why the number of nodes reported by [Moses] exceeds the number of actually quoted nodes.

In the tree given below, one node occupies one line; the height of a given node is determined by its indent from the left; the subsequent nodes of the same height denote a sequence of disjunctively connected literals. Thus, the root node in the tree is the literal "`-LessThan(ProductOfAnd(A,C), ProductOfAnd(B,C))`", the thesis of the theorem.

```
[Moses] launch (time limit: 10 sec, initial db: 1, final db: 100)
[Moses] proof tree (58 nodes, 5 levels):
[Moses]  1:   -LessThan(ProductOfAnd(A, C), ProductOfAnd(B, C))
[Moses]  2:      LessThan(ProductOfAnd(A, C), ProductOfAnd(B, C))
[Moses]  3:      LessThan(ProductOfAnd(B, C), ProductOfAnd(A, C))
[Moses]  4:         -LessThan(ProductOfAnd(B, C), ProductOfAnd(A, C))
[Moses]  5:         -EQU(C, Zero)
[Moses]  6:            EQU(C, Zero)
[Moses]  7:         LessThan(B, A)
[Moses]  8:            -LessThan(B, A)
[Moses]  9:            -LessThan(A, B)
[Moses] 10:               LessThan(A, B)
[Moses] 11:            -EQU(A, B)
[Moses] 12:               EQU(A, B)
[Moses] proved in 300 msec -- proof tree nodes: 58 -- proof tree depth: 5
[Moses] inference steps: 55456 -- born nodes: 141743 -- depth bound: 5
```

This proof can be read from bottom to top, i.e. from assumptions to goals, as follows (in parentheses, we justify a corresponding statement and indicate a line in the proof tree given above, where this statement occurs):

1. A is not equal to B (assumption, line 11)
2. A is not less than B (hypothesis of the theorem, line 9)
3. Hence, B is less than A ("`TotalLess`", line 7)
4. C is nonzero (assumption, line 5)
5. Hence, the product of B and C is less than the product of A and C ("`Monot`", line 3)
6. Hence, the product of A and C is not less than the product of B and C ("`ASymmLess`", line 1) and the proof is finished.

A module of SAD intended for translation of proof trees generated by [Moses] to ForTheL-proofs is now under development.

The theorem "`MonotNSt`" completes the ForTheL-text under consideration, and the verification session is successfully finished.

5 Conclusion

The current version of SAD is intended for solving the following tasks:

- establishing of the deducibility of sequents in the first-order classical logic;
- proving of ForTheL-theorems immersed into a self-contained ForTheL-text;
- verification of self-contained ForTheL-texts.

So, SAD can be helpful in attacking the following problems: distributed automated theorem proving, verification of mathematical papers for soundness, extraction of knowledge from mathematical papers, remote training in mathematical disciplines via Internet, construction of knowledge bases for mathematical theories, and integration of computer algebra with deduction. Also, SAD can be adapted to solve logical tasks of decision making theory, to verify the formal specifications of both software and hardware systems, and so on.

References

1. V. Glushkov. Some Problems of Automata Theory and Artificial Intelligence (in Russian). In: Kibernetika, No 2, 1970, 3–13.
2. K. Vershinin, A. Paskevich. ForTheL – the Language of Formal Theories. In: IJ Information Theories and Applications, v. 7-3, 2000, 121–127.
3. A. Degtyarev, A. Lyaletski, M. Morokhovets. Evidence Algorithm and Sequent Logical Inference Search. In: Lecture Notes in Artificial Intelligence, v. 1705, 1999, 44–61.
4. A. Degtyarev, A. Lyaletski, and M. Morokhovets. On the EA-Style Integrated Processing of Self-Contained Mathematical Texts. In: Symbolic Computation and Automated Reasoning (the book devoted to the CALCULEMUS-2000 Symposium: edited by M. Kerber and M. Kohlhase), A.K. Peters, Ltd, USA, 2001, 126–141.
5. K. Verchinine, A. Degtyarev, A. Lyaletski, and A. Paskevich. SAD, a System for Automated Deduction: a Current State. In: Proceedings of the Workshop on 35 Years of Automating Mathematics, Heriot-Watt University, Edinburgh, Scotland, 10-13 April, 2002, 12 pp.
6. Z. Aselderov, K. Verchinine, A. Degtyarev, A. Lyaletski, A. Paskevich, and A.Pavlov. Linguistic Tools and Deductive Technique of the System for Automated Deduction. In: Proceedings of of the 3rd International Workshop on the Implementation of Logics, Tbilisi, Georgia, October 14-18, 2002, 21-24.
7. Z. Aselderov, K. Verchinine, A. Degtyarev, A. Lyaletski, and A. Paskevich. Peculiarities of Mathematical Texts Processing in the System for Automated Deduction, SAD (in Russian). In: Artificial Intelligence (Proc. of the 3rd International Conference "Artificial Intelligence", Katsiveli, Ukraine), October 2002, No 4, 164-171.
8. D. Brand. Proving Theorems with the Modification Method. In: SIAM Journal of Computing, v. 4, 1975, 412–430.