

Detecting Errors in Domain Theories

Julian Richardson¹ and Laurențiu Leuștean²

¹Research Insititute for Advanced Computer Science
NASA Ames Research Center, Moffett Field, CA 94035-1000, USA
Email: julianr@email.arc.nasa.gov

²Institute of Mathematics “Simion Stoilow” of the Romanian Academy
Calea Griviței 21, P.O. Box 1-764, Bucharest, Romania
Email: laurentiu.leustean@imar.ro

Abstract. Errors occur in mathematical domain theories. In this paper we investigate the problem of detecting these errors. Some errors result in inconsistent theories. Many, however, do not result inconsistent theories and so cannot be detected by inconsistency checking. We propose two solutions to this problem: a methodology of *theory testing* in which a prover is used to verify positive and negative test cases, and the addition of axioms to a suspect theory in order to encode its intended meaning.

1 Introduction

Errors occur in mathematical domain theories. In this paper we investigate the problem of detecting errors in mathematical domain theories. Erroneous theories can be divided into two categories: in *inconsistent theories*, we can derive *false* from a theory. By the rule of *ex falso quod libet* we can therefore deduce anything. Proofs in the theory are unsafe. Clearly this is undesirable. In *weak theories*, we cannot derive some intended consequences.

It is natural to draw an analogy between errors in programs and errors in mathematical domain theories. Indeed, in logic-based programming, and in program synthesis, transformation and verification, a mathematical domain theory is an important part of a software system. One important place where the analogy breaks down is in the detection of errors: there are well-developed methodologies and tools for detection of errors in programs, but neither methodology nor tools exist for mathematical domain theories. The early detection of errors in software has been shown to lower the cost of the software over its lifetime. This may also be the case for mathematical domain theories.

We consider three techniques for detecting erroneous theories:

- *ad hoc patching*, in which errors are detected and fixed as they are found when the theory is used. While this technique has no up front cost, we expect it to have a large cost in patching and maintaining the theory in use.
- Refutation proving, in which the theory is given to a refutation theorem prover to find inconsistencies. This can detect inconsistencies in the theory, but not weaknesses. In addition, proof search is hard so inconsistent theories may be missed.

- Theory testing, in which a theorem prover is used to check a number of positive (should be proved) and negative (should not be proved) test cases. The testing approach has the advantage that it is simple, efficient and can detect both inconsistency, weakness and unintended consequences.

The principal contributions of this paper are the proposal and evaluation of a *testing methodology* for domain theories. We show that testing can detect errors which cannot be detected by refutation finding techniques.

The paper is structured as follows: in §2 we discuss and provide examples of the different kinds of errors which can occur in a domain theory. We then (§3) discuss the four approaches mentioned above for detecting errors. In §§5,6,7 we apply these techniques to a number of examples.

In this paper we consider only first order untyped theories. The techniques discussed here are equally applicable to typed and to untyped and modal or higher-order theories, but provers for untyped logics are more readily available and more automated than for other logics.

2 Classification of Errors

A simple predicate calculus example will illustrate the two kinds of errors which we are considering in theories.

$$man(socrates) \tag{1}$$

$$\forall x man(x) \rightarrow mortal(x) \tag{2}$$

$$\forall x mortal(x) \rightarrow man(x) \tag{3}$$

$$mortal(k9) \tag{4}$$

$$dog(k9) \tag{5}$$

$$\forall x (\neg dog(x) \vee \neg man(x)) \tag{6}$$

The theory comprising (1,2,3,4,5,6) is clearly *inconsistent*. This is the most straightforward kind of error to detect, because it does not depend on the intended meaning of the theory. Inconsistent theories are very dangerous. From an inconsistent theory we can prove anything. It is not necessarily obvious from a proof that anything has gone wrong — indeed, if we use refutation theorem proving, then derivation of *false* is always the last step of the proof and faulty proofs look much like correct proofs.

If the theory is not inconsistent, correctness can be defined as having as a consequence everything we expect to be true, and not having as a consequence anything we expect to be false. In this case, if we expect this theory to have as consequences that Socrates is a mortal man, K9 is a mortal dog, and Socrates is not a dog, then the theory comprising (1,2,3,4,5) is consistent, but *inadequate* in that we cannot demonstrate that Socrates is not a dog. Note that our expression of the expected consequences of the theory here was informal. In §5.2 below we will consider two ways to encode our expectations in way which makes them verifiable: logical axiomatisation, and formulation of test cases.

3 Approaches to Detecting Errors

3.1 Introduction

Apart from detecting inconsistent theories, there is very little literature relating to the detection of errors in mathematical domain theories, so we hypothesise that errors are generally detected on an ad hoc basis, i.e. when the theory is being developed and subsequently used. If a theory does not permit the proof of a desired consequence, it is extended. If a theory is found to be inconsistent, it is corrected.

In the software world, it is well known that early detection of errors lowers software cost. We see no reason why this should not also be the case in the case of mathematical domain theories. Indeed, in program synthesis, verification and formal methods in general, theorem proving and associated domain theories may be an integral part of a software system, so if we are following good software development practice we should address correctness of the theory and prover early in development.

3.2 Refutation Finding

The clearest form of incorrectness in a theory is *inconsistency*. Refutation theorem provers, such as *Otter* [10] can conveniently be used to find refutations of a theory, but due to the size of proof search spaces and the incompleteness of proof search, refutation proving may fail to find an inconsistency even where one exists. Nevertheless, provers are readily available, and we used *Otter* in the experiments reported in this paper.

Two recent publications address explicitly detection of inconsistencies in domain theories. In [1], an abstraction is presented from first order logic with and without equality to second-order monadic logic without successor functions, which is decidable. The abstraction preserves provability, so that if the translation of a first order conjecture $T \vdash \phi$ is not provable in second-order monadic logic, then $T \vdash \phi$ is not provable in first order logic. If the translation is provable, then $T \vdash \phi$ may or may not be provable. This approach allows the refutation of a subset of first order theories. In [12], a system, CORAL, is described which is based on implicit induction. The technique implemented by the tool is guaranteed to refute non-theorems in finite time.

Whichever technique we use for detecting inconsistencies, not all incorrect theories are inconsistent.

We show in §5.2 below that incorrect, but consistent, theories can be detected if suitable axioms are added encoding the intent behind the theory, for example that rewriting generates an equivalence relation.

3.3 Testing

One of the main ways in which correctness of programs is addressed is *testing*. In contrast to formal verification, testing is lightweight — everybody uses it to

a greater or lesser degree. Test cases are derived from program requirements, yielding test inputs plus an expectation of result the program should produce when run on the test inputs.

To our knowledge, the testing methodology has not been systematically applied to the correctness of domain theories. We can write test cases to check that the theory agrees with the intent of a theory, if that intent is known. *Positive* test cases are conjectures which we expect to be consequences of the theory. *Negative* test cases are conjectures which we do not expect to be consequences of the theory. A (not necessarily refutation-based) theorem prover is applied to each of the test cases (together with the theory being tested) in turn. If any of the negative test cases succeed, we know there is a problem with the theory. If any of the positive test cases fail, we know there might be a problem with the theory. If the theory is inconsistent we expect all the positive and negative test cases to succeed.

4 Initial Experiments

4.1 Introduction

We performed a number of experiments in order to test the various techniques for detecting errors in domain theories.

One of the difficulties in carrying out the work described in this paper is finding large first order theories. The first choice for publicly available first order theories is the TPTP (Thousands of Problems for Theorem Provers) library [14]. TPTP provides both a collection of first order problems, and a common language for expressing them. We considered the problems in TPTP version 2.5.0, but found they were not suitable for our experiments because the theories are mostly small, and they have been extensively debugged as a consequence of their use in numerous CASC competitions.

In the following subsections, we describe our experimental set-up, and our initial experiments with two theories: an implementation of LPF from the AUTOBAYES system [5], and a deliberately faulty implementation of propositional calculus. Both testing and refutation finding both detected inconsistent theories. In addition, testing found errors in both theories which were not detected by refutation proving.

We then carried out a series of more comprehensive tests, which we describe in §7 below.

4.2 Experimental setup

For our initial experiments we used *Otter* version 3.3 and *Vampire* version 2.0 for proving, and *Mace* version 2.2 for model finding. In the experiments reported here, we settled on *Mace* for model finding and *Otter* in automatic proof (`set(auto)`) for proving for uniformity, and because it produces human-readable proofs. While a different prover could have had better performance on the domain theories we used in our experiments, we expect that larger domain theories

would produce similar results. In particular, testing would still have been faster and more effective than refutation finding.

5 Experiments with The LPF Theory

5.1 Introduction

After deciding not to use the TPTP domain theories in our experiments, we next considered the theories which are used in one of the program synthesis systems developed at NASA Ames, AUTOBAYES. Correctness of these theories is important if we are to avoid synthesising buggy programs. During the experiments reported in this section we developed our methodology of theory testing. The experiments highlighted issues of translation from proprietary to first order form, correctness versus consistency, and exploitation in the original system of tests on a translated theory.

5.2 The Theory

We first focused on an implementation in AUTOBAYES of the domain theory for LPF (Logic of Partial Functions) [2, 3], a three-valued logic for covering undefinedness in program proofs, whose propositional part is based on Kleene’s three-valued logic [8]. The truth-tables for the negation, disjunction, and conjunction of LPF are as follows:

<i>or</i>	true fail false	<i>and</i>	true fail false	<i>not</i>	
true	true true true	true	true fail false	true	false
fail	true fail fail	fail	fail fail false	fail	fail
false	true fail false	false	false false false	false	true

This domain theory is large enough — its implementation has 35 rewrite rules — and it is used in a real system. The use of *fail* is quite important in the AUTOBAYES — it is used to reason about exceptional behaviour, e.g. division by zero. Thus, *fail(.)* is used both as a third logical value (bottom), but also as a bottom value for other types (e.g. $1/0 = fail(.)$) Hence, it is important to get this domain theory right, otherwise synthesized programs may generate exceptions.

We wrote a Prolog program which automatically translated the rewrite rules into first order clauses. This posed a number of challenges:

- each AUTOBAYES rewrite rule can specify that a particular control strategy be employed when using it. Since this strategy component has no obvious meaning in first-order logic, we omitted it. As a consequence, some sequences of rewrite applications in our translated theory are never generated by AUTOBAYES.
- Rewrite rules can employ arbitrary Prolog predicates in their conditions, and in particular use *maplist* (which makes them at least second order). Use of *var/1* and *cut (!)* were omitted in the translation. Applications of *maplist*

were translated into new first order predicates. This worked because the LPF domain theory is mostly first order. Many other rewrite systems from AUTOBAYES are genuinely higher order¹ and translation would be much more difficult.

- Since the rewrite rules are Horn clauses, they have a model — the least Herbrand model — and are therefore consistent. We added (non-Horn) axioms for an *equivalent* predicate, stating that rewriting preserves logical equivalence in LPF.

Rewrite rules $LHS \Rightarrow RHS$ are expressed in AUTOBAYES (we simplify a little here) as Prolog clauses of the following form:

```
rewrite(Name, LHS, RHS) :-
    C1, C2, ..., Cn.
```

These are translated to TPTP clauses of the following form (where the C_i ' are the translations of the C_i):

```
input_clause(Name, axiom,
    [--C1', --C2', ..., --Cn',
    ++rewrite(LHS,RHS)]).
```

AUTOBAYES rewrite rules commonly call Prolog predicates in their conditions (the C_i above). In order to faithfully represent the rewrite rules we must provide TPTP axioms defining them. For example, the axioms defining *member* are:

```
input_clause(member_def_0, axiom,
    [--member(X, nil)]).
```

```
input_clause(member_def_1, axiom,
    [++member(X, cons(X, L))]).
```

```
input_clause(member_def_2, axiom,
    [--member(X, L), ++member(X, cons(Y, L))]).
```

We translated² $X = Y$ into *equal*(X', Y'), Prolog $\backslash+$ negation into logical not, Prolog lists $[]$ and $[A|B]$ into *nil* and *cons*(A', B'), and Prolog *cut* into constant *true* with the following clause defining it:

```
input_clause(true, axiom, [++true]).
```

Another issue is how to deal with higher-order predicates, like *maplist*/³ Prolog predicate. The higher-order predicate *maplist*($+Pred, ?List1, ?List2$) applies the predicate *Pred* on all successive pairs of elements from *List1* and *List2* and it fails if *Pred* cannot be applied to a pair. We replaced every application of *maplist* to a predicate *pred* with a new predicate *apply_pred*(*List1*, *List2*) and added corresponding clauses:

¹ This is to be expected — theories which reason about programs (synthesis, verification) are naturally at least second order.

² Primed terms are the translated form of their unprimed counterparts

```

input_clause(apply_pred0,axiom,
  [++apply_pred(nil, nil)]).

input_clause(apply_pred1,axiom,
  [--pred(Elem1, Elem2),
  --apply_pred(Tail1, Tail2),
  ++apply_pred(cons(Elem1,Tail1), cons(Elem2,Tail2))]).

```

This scheme breaks down if *maplist* is ever called with a variable in the (first) predicate position, but such a situation did not often occur in the rewrite rules we looked at.

The most important problem we face in our translation is the fact that the first order-part of every Prolog program is a general logic program, that is a finite set of general clauses, where a general clause has the form $A_0 \leftarrow L_1, \dots, L_n$ with A_0 positive literal, L_1, \dots, L_n positive or negative literals.

The Herbrand base of a general program always *is* a Herbrand model of the program. Of course, this model is rather uninteresting — every n -ary predicate of the program is interpreted as the full n -ary relation over the domain of ground terms — but still it is a model. As a consequence, the initial TPTP theory we get from the automatic translation of the Prolog theory is always consistent. Hence, even though the Prolog theory may be incorrect, we cannot detect it. To deal with this problem, we have to extend the translated theory with some non-Horn clauses. First we add the following non-Horn clause defining the constant *false*:

```
input_clause(false,axiom,[--false]).
```

Second, we replace the clauses which we added to define Prolog predicates used in the rewrite rules by their *Clark completions* [4]: for every predicate $p(X_1, \dots, X_m)$, replace all the clauses:

$$\begin{array}{l}
p(X_1, \dots, X_m) \leftarrow B_1 \\
\vdots \\
p(X_1, \dots, X_m) \leftarrow B_k
\end{array}$$

by the formula

$$\forall X_1, \dots, X_m (p(X_1, \dots, X_m) \leftrightarrow B_1 \vee \dots \vee B_k)$$

If there are no clauses having $p(X_1, \dots, X_m)$ as head, then add the formula:

$$\forall X_1, \dots, X_m (\neg p(X_1, \dots, X_m))$$

For example, the TPTP clauses defining the Clark completion of the *member* predicate are:

```
input_clause(member_def_1,axiom,
  [--equal(X,Y), ++member(X,cons(Y,L))]).
```

```
input_clause(member_def_2,axiom,
[member(X,L),member(X,cons(Y,L))]).
```

```
input_clause(member_def_3,axiom,
[member(X,cons(Y,L)),equal(X,Y),member(X,L)]).
```

We extended our translator to automatically generate clauses defining the completion of all Prolog predicates used in the rewrite system.

We used *Otter 3.3* to check the consistency of the LPF domain theory obtained from the translation explained above; it failed to find an inconsistency after 300 CPU seconds.

We therefore extended the theory by defining an *equivalent/2* predicate, which is a kind of congruence closure of the *rewrite/2* predicate, based on the semantics of the rewrite rules. It follows that for every domain theory we must define a specific *equivalent* predicate. In the case of LPF domain theory, we want to rewrite a formula *A* into a formula *B* only if *A* and *B* are logically equivalent. *equivalent* is defined by the following TPTP clauses:

```
% equivalent is an equivalence relation extending rewrite:
```

```
input_clause(rewrite_equivalent,axiom,
[rewrite(A,B),equivalent(A,B)]).
```

```
input_clause(equivalent_reflexivity,axiom,
[equivalent(A,A)]).
```

```
input_clause(equivalent_symmetry,axiom,
[equivalent(A,B),equivalent(B,A)]).
```

```
input_clause(equivalent_transitivity,axiom,
[equivalent(A,B),equivalent(B,C),equivalent(A,C)]).
```

```
% the extension of equivalent to lists of formulas:
```

```
input_clause(equivalent_list,axiom,
[equivalent(L1,L2),equivalent(A1,A2),
equivalent(cons(A1,L1),cons(A2,L2))]).
```

```
% equivalent is compatible with not, or, and:
```

```
input_clause(equivalent_not,axiom,
[equivalent(A,B),equivalent(not(A),not(B))]).
```

```
input_clause(equivalent_or,axiom,
[equivalent(L1,L2),equivalent(or(L1),or(L2))]).
```

```
input_clause(equivalent_or, axiom,
[--equivalent(L1,L2),++equivalent( and(L1), and(L2))]).
```

```
% true, false, fail are not equivalent:
```

```
input_clause(not_equiv_true_false, axiom,
[--equivalent(true,false)]).
```

```
input_clause(not_equiv_true_fail, axiom,
[--equivalent(true,fail)]).
```

```
input_clause(not_equiv_false_fail, axiom,
[--equivalent(false,fail)]).
```

5.3 The Experiments

We now wrote test cases, which are ground atomic formulas from the Herbrand universe. Test cases are positive and negative. Positive test cases are used to verify the strength of the theory. For example, we know that `fail or fail = fail` is true in LPF so we add its negation (the following clause) to the theory and the theorem prover should find a refutation. This should happen for all positive test cases.

```
input_clause(to_prove, conjecture,
[--rewrite(or(cons(fail, cons(fail,nil))), fail)]).
```

Negative test cases are used to verify the consistency of the theory. For example `(fail or fail) = true` is not true in LPF. So, we add its negation to the theory and if the theorem prover finds a proof, then the theory is inconsistent.

We formulated 60 negative test cases and 30 positive test cases, with the connectives *or*, *and*, *not* and *implies* and the truth-values *true*, *fail* and *false*. Formulating these test cases was significantly helped by the comments (especially the truth table) in the files defining the theory. Otter found proofs for more than 15 negative test cases; that is, more than 15 hidden inconsistencies in the domain theory. Amongst the test cases, we found proofs for both:

```
input_clause(to_prove, conjecture,
[--equivalent(or(cons(fail, cons(fail,nil))), fail)]).
```

```
input_clause(to_prove, conjecture,
[--equivalent(or(cons(fail, cons(fail,nil))), true)]).
```

This indicates that the theory is incorrect, but AUTOBAYES does not generate the second (erroneous) derivation because of restrictions imposed in the rewrites' control components. By inspecting the proofs, we altered the test case to find a real bug in AUTOBAYES:

```
?- simplify(or[not(fail),fail],A).  
A=true
```

Since $\text{not}(\text{fail}) = \text{fail}$, the answer should have been $A = \text{fail}$.

Finally, we applied *Otter* to the theory (extended with Clark completions and **equivalent**). *Otter* found that the theory is inconsistent. In the next examples we will see that test cases can reveal errors when a theory is not found to be inconsistent.

6 Experiments with The Propositional Calculus Theory

A second theory which we used in our experiments is an OBJ3 axiomatisation of propositional calculus [6], originally due to Hsiang [7]. We deliberately seeded an erroneous axiom into this theory, $\text{or}(X,Y) \leftrightarrow \text{and}(\text{not}(X),\text{not}(Y))$ to see whether our techniques could detect it.

First, we applied both *Otter 3.3* and *Vampire 6.0* to the theory to check for inconsistency. Both provers failed to find an inconsistency 300 CPU seconds (*Otter* was run locally, *Vampire* was run from the University of Miami TPTP Systems web page, accessible from <http://www.tptp.org>.)

We then used the theory to prove four positive test cases, but also four negative test cases. In all cases, the proofs were found quickly.

Finally, we tried to verify the consistency of the theory, using *Mace* to search for models. *Mace* found no models with domain size 2, but did find a model with domain size 3. Since every first order theory with a model is consistent, this theory is consistent.

This experiment illustrates that testing can quickly locate errors in a theory which cannot be found by refutation — in this case, the theory is consistent so cannot be refuted — and that both positive and negative test cases are needed.

7 Systematic Testing With LPF Domain Theory

7.1 The Test Sets

We then reused the test cases from the earlier LPF experiments, applying them in two systematic sets of tests. The first set was based on a consistent subset of the LPF domain theory. We created 30 variants of this theory by making small systematic changes. The changes were chosen manually but an effort was made to avoid biasing the experiments by making interesting choices. *Otter* was applied to each variant with a time limit of 300 CPU seconds in order to check for consistency. All of the test cases were then applied with a time limit of 30 CPU seconds to each of these variants.

The second set was based on a corrected version of the original LPF domain theory. As above, we created 15 variants, checked each for consistency with *Otter*, and applied the test cases to each of these variants.

7.2 Results

The following figures summarise the results of the tests on the two sets of theories. Each row corresponds to one version of the theory. The first column is the theory with no changes made. The first column of each row indicates the result of inconsistency testing using *Otter* (without test cases): ‘C’ denotes consistency, ‘I’ denotes inconsistency. Subsequent columns indicate the result of applying a test case to the theory: ‘P’ denotes that a proof was found, ‘.’ denotes that the proof attempt timed out without proof, ‘e’ denotes that *Otter* terminated with an empty SOS list. For space reasons we only show the results of the first 40 negative test cases (the others display the same pattern).

In the first set of tests, all negative test cases (Figure 1) succeeded when the theory was inconsistent. In one case (the seventh row), a negative test case revealed an error when the theory had not been found to be inconsistent. All positive test cases (Figure 2) also succeeded when the theory was inconsistent. Most positive test cases failed (i.e. timed out) on the consistent theories — this indicates that the theories were very weak.

In the second set of tests, the behaviour on inconsistent theories was unchanged from the first. The negative test cases (Figure 3) reveal two theories which had not been found to be inconsistent, but which did contain errors. The positive test cases (Figure 4) indicate that several of the theory variants were too weak.

7.3 Interpretation

The results, while not spectacular, do indicate that testing can be a useful way of detecting incorrectness. When the theory is inconsistent, all test cases (positive and negative) succeed. When the theory is not inconsistent, it may still be incorrect — in several of the experiments some of the negative test cases succeeded, and in many of the experiments, positive test cases failed. Success of negative test cases implies that the theory is incorrect but does not necessarily imply that the theory is inconsistent. Failure of the positive test cases, especially given the short time limit of 30 seconds, does not imply that those test cases are not consequences of the theory, but it can be seen as a warning that the theory may not have all desired consequences.

8 Conclusions and Further Work

In this paper we have examined the problem of detecting errors in mathematical domain theories from a practical point of view. We have described several techniques: the established technique of searching for refutations of the theory, *theory testing* in which a prover is used to verify or refute positive and negative test cases, and the addition of axioms to a suspect theory in order to encode its intended meaning. The formulation of useful test sets was made easier when it was clear what a theory was intended to mean — like good programs, good

theories should have comments, or even explicit (informal) statements of their requirements.

We have applied these techniques to a number of interesting domain theories, including one from a program synthesis system, and detected errors in these theories. In the case of the synthesis system, extensive comments in the theory source file were invaluable when writing test cases.

For further work, we would like to further evaluate these techniques, especially on large domain theories (if we can get hold of them). The complications of translation from Prolog using higher-order and impure features were a challenge. Theories which reason about programs (synthesis, verification) are naturally at least second order. Use of a second- or higher-order prover to refute theories and apply test cases would help to reduce the translation burden.

References

1. S. Autexier, C. Schürmann, Disproving False Conjectures, In M. Vardi, A. Voronkov (Eds.) Proceedings of LPAR'03, Almaty, Kazakhstan, September, 2003.
2. H. Barringer, J. H. Cheng, C. B. Jones, A Logic Covering Undefinedness in Program Proofs, *Acta Informatica*, Volume 21, 251-269, 1994.
3. J. H. Cheng, Logic for Partial Functions, PhD Thesis, Department of Computer Science, Manchester University, 1986.
4. K. L. Clark, Negation as failure. In H. Gallier, J. Minker, editors, *Logic and Data Bases*, Plenum Press, New York, 293-322, 1978.
5. B. Fischer, J. Schumann, AUTOBAYES: A System for Generating Data Analysis Programs from Statistical Models, to appear in *Journal of Functional Programming*.
6. J. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, J.-P. Jouannaud, Introducing OBJ, in J. Goguen (Ed.), *Applications of Algebraic Specification using OBJ*, Cambridge University Press, 1993.
7. Hsiang, Refutational Theorem Proving using Term Rewriting Systems. PhD thesis, University of Illinois at Champaign-Urbana, 1981.
8. S. C. Kleene, On a notation for ordinal numbers, *Journal of Symbolic Logic*, Volume 3, 1950-195, 1938.
9. M. Kohlhase and A. Franke, MBase: Representing Knowledge and Context for the Integration of Mathematical Software Systems, *Journal of Symbolic Computation*, Volume 32 number 4, 2001.
10. W. McCune, Otter 3.3 Reference Manual and Guide, Technical Memorandum no. 263, Argonne National Laboratory, 2003.
11. W. McCune, Mace 2.0 Reference Manual and Guide, Technical Memorandum no. 249, Argonne National Laboratory, 2001.
12. G. Steel, A. Bundy, E. Denney, Finding Counterexamples to Inductive Conjectures and Discovering Security Protocol Attacks, in *Proceedings of the 2002 Workshop on Foundations of Computer Security*.
13. A. Riazanov, A. Voronkov, The Design and Implementation of Vampire, *AI Communications*, Volume 15. Numbers 2-3, 2002.
14. C. B. Suttner, G. Sutcliffe, The TPTP Problem Library. TPTP v.2.5.0, Technical report, University of Miami, 2002.