Semantic Derivation Verification

Geoff Sutcliffe, Diego Belfiore
Department of Computer Science, University of Miami
P.O. Box 248154, Coral Gables, FL 33124, USA
Email: `geoff@cs.miami.edu`, `diego@mail.cs.miami.edu`

Abstract

Automated Theorem Proving (ATP) systems are complex pieces of software, and thus may have bugs that make them unsound. In order to guard against such unsoundness, the derivations output by an ATP system may be *semantically* verified by a trusted system that checks the required semantic properties of each inference step. Such verification may need to be augmented by structural verification that checks that inferences have been used correctly in the context of the overall derivation. This paper describes techniques for semantic verification of derivations, and reports on their implementation in the DVDV verifier.

1. Introduction

Automated Theorem Proving (ATP) systems are complex pieces of software, and thus may have bugs that make them unsound or incomplete. While incompleteness is common (sometimes by design) and tolerable, when an ATP system is used in an application it is important, and in some cases mission critical, that it be sound, i.e., that it never report that a solution has been found when this is not the case. Directly verifying the soundness of an implemented state-of-the-art ATP system seems impractical, due to the complexity of the low level coding that is typically used (McCune & Shumsky-Matlin 2000). Thus other techniques are necessary, and several possibilities are evident.

First, an ATP system may be *empirically* verified, by testing it over a large number of problems. If the system consistently returns the correct (at least, expected) answer, confidence in the system's soundness may grow to a sufficient level. For example, it is commonly accepted that Otter (McCune 2003b) is sound, thanks to its extensive accepted usage by many researchers over many years. Second, the derivations output by a system may be *syntactically* verified. In syntactic verification each of the inference steps in a derivation are repeated by a trusted system, to ensure that the inferred formula can be inferred from the parent formulae by the inference rule stated. This is the approach taken in the IVY system (McCune & Shumsky-Matlin 2000), in the Omega proof checker after reduction to ND form (Siekmann 2002), and is planned for the in-house verifier for Vampire (Riazanov & Voronkov 2002). A serious disadvantage of syntactic verification is that the verification system must implement all of the inference rules of all of the ATP systems to be verified (which is impossible to do in the present for inference rules of the future). Third, an appeal may be made to *higher order techniques*, in which a 1st order proof is translated into type theory and checked by a higher order reasoning system (Harper, Honsell, & Plotkin 1993). This is the approach taken by Bliksem's in-house verifier (Bezem & de Nivelle 2002). A weakness of this approach is the introduction of translation software, which may introduce or even hide flaws in the original 1st order proof. Fourth, the derivations output by an ATP system may be *semantically* verified. In semantic verification, the required semantic properties of each inference step are checked by a trusted ATP system. For example, deduction steps are verified by checking that the inferred formula is a logical consequence of the parent formulae. This is the approach taken in the low level checker of the Mizar system (Rudnicki 1992), has been adopted by several in-house verifiers for contemporary ATP systems, has been implemented using the "hints" strategy in Otter (Veroff 1996), and forms the core of the verification process described in this paper.

The advantages of semantic verification include: independence of the trusted ATP system from the ATP system that produced the derivation (this advantage also applies to syntactic checking and higher order techniques, and contrasts with the internal proof checking in systems such as Coq (Bertot & Casteran 2004) and Isabelle (Paulson & Nipkow 1994)); independence from the particular inference rules used in the ATP system - see Section 2; and robustness to the preprocessing of the input formulae that some ATP systems perform - see Section 2.2. Semantic verification is able to verify any form of derivation, not only proofs, in which the required semantic properties of each inference step are known.[1] However, semantic verification of inference steps where the inferred formula is not a simple logical consequence of the premises, e.g., Skolemization and splitting, is not straight forward. At this stage no general purpose semantic verification technique has been developed for such inference steps, and techniques that are specific to each of

[1]That is the reason for using the term "derivation verification" rather than the more common "proof checking".

the inference rules are used.

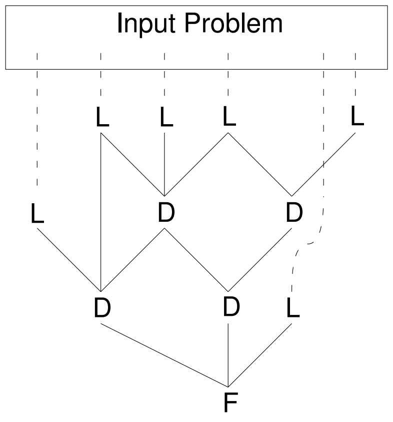
All the forms of verification that examine ATP systems' derivations implicitly include some *structural* verification, i.e., verification of properties of the derivation structure, rather than of the individual inferences within the derivation. A basic structural check is that the specified parents of each inference step do exist in the derivation, and more complex checks are required in certain situations. Structural checking is the basis for the high level checker of the Mizar system.

This paper describes techniques for semantic verification of derivations, and reports on their implementation in the DVDV verifier.[2] The techniques were developed to verify derivations in classical 1st order logic, and have been particularly applied to derivations in clause normal form (CNF). However, the principles are more widely applicable. Section 2 describes how various parts of a derivation are semantically verified, and Section 3 describes structural verifications. Section 4 gives some details of the implementation of DVDV. Section 5 examines the extent to which these verification techniques can be trusted. Section 6 concludes the paper.

2. Semantic Verification

A derivation is a directed acyclic graph (DAG), whose leaf nodes are formulae (possibly derived) from the input problem, whose non-leaf nodes are formulae inferred from parent formulae, and whose root node is the final derived formula. Figure 1 shows this structure. For example, a CNF refutation is a derivation whose leaf nodes are the clauses formed from the axioms and the negated conjecture, and whose root node is the empty clause.

Figure 1: Derivation DAG



Semantic verification of a derivation has two notionally distinct parts. First, it is necessary to check that each leaf node is a formula (possibly derived) from the input problem. Second, it is necessary to check that each inferred formula has the required semantic relationship to its parents. The required semantic relationship of an inferred formula to its parents depends on the intent of the inference rule used. Most commonly an inferred formula is intended to be a logical consequence of its parents, but in other cases, e.g., Skolemization and splitting, the inferred formula has a weaker link to its parents. A comprehensive list of inferred formula statuses is given in (Sutcliffe, Zimmer, & Schulz 2004).

[2]The acronym has nothing to do with "derivation verifier" - it stands for "Diego's Version of the Dashti Verifier", named after the programmers.

The techniques described here verify:

- that leaf formulae are, or are derivable from, the input problem formulae;
- that inferred formulae are logical consequences of their parents;
- three forms of splitting inferences;
- some structural requirements of derivations, particularly of splitting steps in CNF refutations.

The verification of logical consequences is described first because the technique is also used in checking leaf formulae and splitting steps.

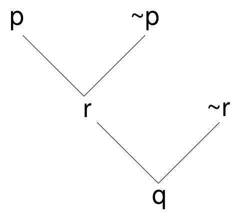
2.1 Logical Consequences

The basic technique for verifying logical consequences is well known and quite simple. The earliest use appears to have been in the in-house verifier for SPASS (Weidenbach *et al.* 2002). For each inference of a logical consequence in a derivation, an *obligation* problem, to prove the inferred formula from the parent formulae, is formed. If the inference rule implements any theory, e.g., paramodulation implements most of equality theory, then the corresponding axioms of the theory are added as axioms of the obligation problem. The obligation problem is then handed to a trusted ATP system. If the trusted system solves the problem, i.e., finds a proof, the obligation has been *discharged*.

In the case of CNF derivations, the obligation problem is a first order form (FOF) problem, formed from the universally quantified forms of the parents and the inferred formula. If an obligation is discharged by an ATP system that converts the problem to CNF and finds a refutation, then the universally quantified inferred formula in the obligation problem is negated before conversion to CNF.

This verification of logical consequences ensures the soundness of the inference steps, but does not check for *relevance* (Anderson & Belnap 1975). As a contradiction in first order logic entails everything, an inference step with contradictory parents can soundly infer anything. A derivation containing two examples of such inferences is shown in Figure 2. If such inferences should be rejected, an obligation problem consisting of only axioms is formed from the parents of the inference, and must be discharged by being shown to be satisfiable. (This verification step is definitely not applicable to the final inference of a refutation.) Due to the semi-decidability of first order logic, such satisfiability obligations cannot be guaranteed to be discharged. Three alternative techniques, described here in order of preference, may be used to show satisfiability. First, a finite model of the axioms may be found using a model generation system such as MACE (McCune 2003a) or Paradox (Claessen & Sorensson 2003). Second, a saturation of the axioms may be found using a saturating ATP system such as SPASS or E (Schulz 2002b). Third, an attempt to show the axioms to be contradictory can be made using a refutation system. If that succeeds then the obligation cannot be discharged. If it fails it provides an incomplete assurance that the formulae are satisfiable. In addition to being useful for rejecting inferences from contradictory parents, relevance checking is also useful in the verification of splitting steps, as described in Section 2.3.

Figure 2: Irrelevant Inferences



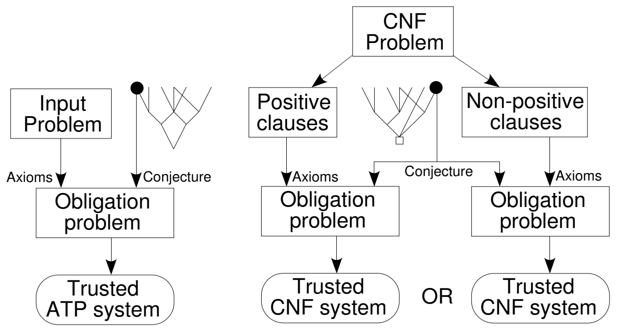
2.2 Leaf Formulae

To verify that a leaf formula is a formula (possibly derived) from the input problem, an obligation to prove the leaf formula from the input formulae must be discharged. This technique can normally be used directly, as shown in the left hand side of Figure 3. In the case that the derivation being verified is a CNF refutation, and obligations are discharged by finding a CNF refutation, more careful control is required. A refutation of the input clauses and the negated leaf clause does not necessarily mean that the leaf clause is a clause (possibly derived) from the input clauses, because the refutation may be of the input clauses alone. The solution is to partition the input clauses into two satisfiable parts, e.g., one part containing all the positive clauses and the other part containing all the non-positive (mixed and negative) clauses, as shown in the right hand side of Figure 3. Two alternative obligation problems are then formed, one consisting of the first part and the negated leaf clause, and the other consisting of the second part and the negated leaf clause. If the trusted CNF system discharges either of these, then the leaf clause is a clause (possibly derived) from the input clauses. If it fails, that may be because the leaf formula is derived from parents that are not all in one of the two partitions of the input clauses. Alternative partitioning may then be tried, e.g., a predicate partition.

An advantage of the semantic technique for verifying leaf formulae is that it is somewhat robust to preprocessing inferences that are performed by some ATP systems. For example, Gandalf (Tammet 1998) may factor and simplify input clauses before storing them in its clause data structure. The leaves of refutations output by Gandalf may thus be derived from input clauses, rather than directly being input clauses. These leaves are logical consequences of the original input clauses, and can be verified using this technique.

If the input problem is in FOF, and the derivation is a CNF

Figure 3: Verification of Leaf Formulae



refutation, the leaf clauses may have been formed with the use of Skolemization. Such leaf clauses are not logical consequences of the FOF input formulae. The verification of Skolemization steps has not yet been addressed, and remains future work (see Section 6). This is a form of verification unsoundness, which can be tolerated, as discussed further in Section 5.

2.3 Splitting

Many contemporary ATP systems that build refutations for CNF problems use *splitting*. Splitting reduces a CNF problem to one or more potentially easier problems by dividing a clause into two subclauses. Splitting may be done recursively; a clause in a subproblem may be split to form sub-subproblems, etc. There are several variants of splitting that have been implemented in specific ATP systems, including *explicit splitting* as implemented in SPASS (also called *explicit case analysis* in (Riazanov & Voronkov 2001)), and forms of *pseudo splittng* as implemented in Vampire and E (also called *splitting without backtracking* in (Riazanov & Voronkov 2001)). The verification of splitting steps has been omitted in existing ATP systems' in-house verification programs.

2.3.1 Explicit Splitting Explicit splitting takes a CNF problem $S \cup \{L \vee R\}$, in which L and R do not share any variables, and replaces it by two subproblems $S \cup \{L\}$ and $S \cup \{R\}$. These are referred to as the L and R subproblems, respectively. If both the subproblems have refutations i.e., are unsatisfiable, then it is assured that the original problem is unsatisfiable. In SPASS' implementation of explicit splitting, when a refutation of the L (R) subproblem has been found, $\neg L$ ($\neg R$) is inferred, with the root of the subproblem's refutation, L (R), and $L \vee R$ as parents. This inferred clause is called the *anti-kid* of the split. It is a logical consequence of S, and can be used in any problem that includes S. Semantic verification of explicit splitting steps and the anti-kid inferences are described here. Constraints required for the soundness of explicit splitting are structurally verified, as described in Section 3.

To verify a explicit splitting step's role in establishing the overall unsatisfiability of the original problem clauses, an obligation to prove $\neg(L \vee R)$ from $\{\neg L, \neg R\}$ must be discharged. The soundness of the split is then assured as follows: The ATP system's (verified) refutations of the L and R subproblems show that every model of S is not a model of L or of R, and thus every model of S is a model of $\neg L$ and of $\neg R$. The discharge of the obligation problem shows that every model of $\neg L$ and $\neg R$ is a model of $\neg(L \vee R)$, and therefore not a model of $L \vee R$. Thus every model of S is not a model of $L \vee R$, and $S \cup \{L \vee R\}$ is unsatisfiable.

Discharging the obligation problem by CNF refutation also ensures that L and R are variable disjoint - a simple example illustrates this: Let the split clause be $p(X) \vee q(X)$. L is $p(X)$ and R is $q(X)$, i.e., they are not variable disjoint. The obligation problem is to prove $\neg\forall X(p(X) \vee q(X))$ from $\{\neg\forall X p(X), \neg\forall X q(X)\}$. When converted to CNF, the variables in the two unit formulae are converted to Skolem constants, producing the CNF problem $\{p(X) \vee q(X), \neg p(sk1), \neg q(sk2)\}$. This clause set is satisfiable, and the obligation is not discharged.

While discharging the splitting obligation problem assures the soundness of the overall refutation, it does not ensure that the splitting step was performed correctly. For example, it would be incorrect to split the clause $p \vee q$ into p and $\neg p$, but the obligation to prove $\neg(p \vee q)$ from $\{\neg p, p\}$ is easily discharged because of the contradictory axioms of the obligation problem. In such cases the refutations of the two subproblems, $S \cup \{p\}$ and $S \cup \{\neg p\}$, show that S is unsatisfiable alone. If such incorrect splits should be rejected, the discharge of the obligation must also check for relevance, as described in Section 2.1.

An anti-kid A of the L (R) subproblem of a split is verified by discharging an obligation to prove A from $\neg L$ ($\neg R$). The refutation of the L (R) subproblem shows that $S \Rightarrow \neg L$ ($S \Rightarrow \neg R$), and thus by modus ponens A is a logical consequence of any subproblem that includes S.

2.3.2 Pseudo Splitting Pseudo splitting takes a CNF problem $S \cup \{L \vee R\}$, in which L and R do not share any variables, and replaces $\{L \vee R\}$ by either (i) $\{L \vee t, \neg t \vee R\}$, or (ii) $\{L \vee t_1, R \vee t_2, \neg t_1 \vee \neg t_2\}$, where t and t_i are new propositional symbols. Vampire implements pseudo splitting by (i) and E implements it by (ii). The replacement does not change the satisfiability of the clause set – any model of the original clause set can be extended to a model of the modified clause set, and any model of the modified clause set satisfies the original one (Riazanov & Voronkov 2001; Schulz 2002a). The underlying effect of (i) is to introduce a new definitional axiom, $t \Leftrightarrow \neg\forall L$, and of (ii) is to introduce two new definitional axioms, $t_1 \Leftrightarrow \neg\forall L$ and $t_2 \Leftrightarrow \neg\forall R$. Variants of these forms of splitting, that allow L and R to have common variables, and that split a clause into more than two parts (Riazanov & Voronkov 2001), are treated with generalizations of the verification steps described here.

Pseudo splitting steps are verified by discharging obligations that prove the equivalence of the split clause and the replacement clauses. These are done in two parts: First, an obligation to prove the split clause from the replacement clauses must be discharged. To check that this obligation is not discharged because the axioms of the obligation problem (the replacement clauses) are unsatisfiable, the discharge of the obligation problem must also check for relevance, as described in Section 2.1. Second, obligations to prove each of the replacement clauses from the split clause and the new definitional axiom(s) must be discharged.

As is the case with explicit splitting, the obligations cannot all be discharged if L and R share variables. For example, in the second form of pseudo splitting ((ii) above), an obligation to prove $\neg t_1 \vee \neg t_2$ from the split clause and the definitional axioms must be discharged. Let the split clause be $p(X) \vee q(X)$. L is $p(X)$ and R is $q(X)$, i.e., they are not variable disjoint. The obligation problem is to prove $\neg t_1 \vee \neg t_2$ from $\{\forall X(p(X) \vee q(X)), t_1 \Leftrightarrow \neg\forall p(X), t_2 \Leftrightarrow \neg\forall q(X)\}$. When the problem is converted to CNF, two Skolem constants are generated, producing the CNF problem $\{p(X) \vee q(X), \neg t_1 \vee \neg p(sk1), \neg t_2 \vee \neg q(sk2), t_1 \vee p(X), t_2 \vee q(X), t_1, t_2\}$. This clause set is satisfiable, and the obligation is not discharged.

3. Structural Verification

Structural verification checks that inferences have been used correctly in the context of the overall derivation.

For all derivations, two structural checks are necessary: First, the specified parents of each inference step must exist in the derivation. When semantic verification is used to verify each inference step then the formation of the obligation problems relies on the existence of the parents, and thus performs this check. The check can also be done explicitly. Second, there must not be any loops in the derivation. For this check it is sufficient to check that there are no cycles in the derivation, using a standard cycle detection algorithm, e.g., Kruskal's or Prim's algorithm.

For derivations that claim to be CNF refutations, it is necessary to check that the empty clause has been derived. If explicit splitting is used, multiple such checks are necessary, as described below.

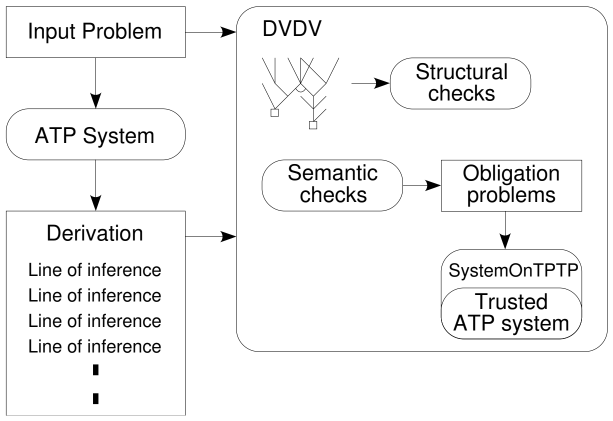
For refutations that use explicit splitting, structural checks of certain splitting steps are required. In such a refutation there are two possible reasons for splitting a clause. The first reason is to produce two easier subproblems, both of which are refuted. The second reason is to refute just one of the subproblems in order to form an anti-kid that is then used in another part of the overall refutation. Structural checking is required in the first case. Clauses that were split for the first reason can be found by tracing the derivation upwards from the root nodes (a derivation with explicit splits has multiple root nodes), but not passing through anti-kid nodes, noting the split clauses that are found. The splitting steps performed on these split clauses then require two structural checks: First, it is necessary to check that both subproblems have been refuted. This is done by ensuring that both the L and R clauses have a false descendant in the refutation DAG. Second it is necessary to check that L (R) and it's descendants are not used in the refutation of the R (L) subproblem. This is done by examination of the ancestors of the false root clause of the refutation of the R (L) subproblem.

For refutations that use pseudo splitting, a structural check is required to ensure that the "new propositional symbols" genuinely are new, and not used elsewhere in the refutation.

4. Implementation and Testing

The semantic verification techniques described in Sections 2 and 3 have been implemented in the DVDV system. DVDV is implemented in C, using the JJParser library's input, output, and data structure support. The inputs to DVDV are a derivation in TSTP format (Sutcliffe, Zimmer, & Schulz 2004), the input problem in TPTP (Sutcliffe & Suttner 1998) or TSTP format, trusted ATP systems to discharge obligation problems, and a CPU time limit for the trusted ATP systems for each obligation problem. SystemOnTPTP (Sutcliffe 2000) is used to run the trusted ATP systems. The overall architecture and use of DVDV is shown in Figure 4.

Figure 4: The DVDV Architecture



Obligations that are successfully discharged are reported. If an obligation cannot be discharged, or a structural check fails, DVDV reports the failure. As is explained in Section 5, a failure to discharge an obligation does not necessarily imply a fault in the derivation, and thus a command line flag provides the option to continue after a failure.

DVDV has been tested on solutions from SPASS and EP (EP is a combination of E and a proof extraction tool) for FOF theorems and CNF unsatisfiable sets in TPTP version 2.7.0, as contained in the TSTP solution library (Sutcliffe URL). Testing DVDV on the large number of solutions in the TSTP has provided a high level of confidence in the soundness of DVDV (the soundness and completeness of derivation verifiers, as opposed to the soundness of an ATP system whose derivations are verified, is discussed in Section 5). However, it is expected that all the derivations in the TSTP are correct, and therefore failure to find fault with them is expected. In order to test the completeness of DVDV, faults have been inserted by hand into existing derivations, and DVDV has successfully found these. In order to build a higher level of confidence in the completeness of DVDV it would be necessary to have a test library of faulty derivations - something that seems difficult to find or generate.

5. Trusting the Verifier

A derivation verifier is *complete* if it will find every fault in a derivation. A derivation verifier is *sound* if it claims to have found a fault only when a fault exists. Conversely to the case of ATP systems, completeness is more important than soundness - if a verifier mistakenly claims a fault the derivation can be further checked by other means, but if a fault is bypassed the flawed derivation may be accepted and used.

All verifiers that rely on a trusted system must contend with the possibility that the trusted system is unsound or incomplete. This is the case for all except the first of the verification techniques described in Section 1. In the case of IVY, the trusted system is implemented in ACL2 (Kaufmann, Manolios, & Strother Moore 2000), and has been verified in ACL2 as being sound in the context of finite models (it is believed that this may be extended to infinite models). In this case the trust has ultimately been transferred to ACL2's verification mechanisms. In Bliksem's case the 1st order proof is translated into type theory with in-house software, and the higher order reasoning system Coq is used for testing the type correctness of the translated proof. The combination of the translator and Coq thus form the trusted system. For semantic verification trust is placed in the ATP systems that are used to discharge the obligation problems. In the first order case, even if the trusted systems are theoretically complete and correctly implemented, the finite amount of resources allocated to any particular run means that the trusted systems are practically incomplete.

For semantic verification, incompleteness of the trusted system for theoretical, implementation, or resource limit reasons, means that some proof obligations may not be discharged. In such a situation the verifier can make a possibly unsound claim to have found a fault. Although undesirable, this is not catastrophic, as described above. In contrast, unsoundness of the trusted system leads to incompleteness of the semantic verifier, and must be avoided. The question then naturally arises, "How can the trusted system be verified (as sound)?" The first approach to verifying a system - extensive use, provides one exit from this circle of doubt. The trust may be enhanced by making the trusted system as simple as possible. The simpler the system, the less likely that there are bugs. At the same time, simpler systems are less powerful. If a derivation has to be semantically verified using a weak trusted system, that requires that the inference steps be reasonably fine grained. This may be a desirable feature of derivations, depending on the application. Thus, as the trusted system's *limbo bar* is lowered, the level of trust in the trusted system rises and the granularity of the inference steps must get finer. The lowest level for the limbo bar seems to be something like the "obvious inferences" described in (Davis 1981) and (Rudnicki 1987). In DVDV the trusted systems are configured to use small but (refutation) complete sets of inference rules, e.g., Otter is configured to use binary resolution, factoring, and paramodulation.

A further technique to enhance the level of confidence in semantic verification is *cross verification*. For a given input problem and a set of ATP systems, cross-verification requires that every system be used as the trusted system for verifying the inference steps of every other system's solution to the problem. In this manner it is ensured that either all or none of the systems are faulty.

6. Conclusion

This paper describes techniques for semantic verification of derivations, focussing particularly on derivations in 1st order logic. The techniques have been implemented in the DVDV system, resulting in a verifier that can verify any TSTP format derivation output by any ATP system. It has been successfully tested on proofs from SPASS and EP. This is the first systematic development of a general purpose derivation verifier.

In the future it is hoped to deal with Skolemization in FOF to CNF conversion. Some aspects of user level verification will also be integrated - the first check will be that the problem's axioms are satisfiable, to avoid proofs due to contradictory axioms. The second check will be to check that all formulae output as part of a derivation are necessary for the derivation - already in this work it has been noticed that some ATP systems output superfluous formulae.

References

Anderson, A., and Belnap, N. 1975. *Entailment: The Logic of Relevance and Necessity, Vol. 1*. Princton University Press.

Bertot, Y., and Casteran, P. 2004. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag.

Bezem, D., and de Nivelle, H. 2002. Automated Proof Construction in Type Theory using Resolution. *Journal of Automated Reasoning* 29(3-4):253–275.

Claessen, K., and Sorensson, N. 2003. New Techniques that Improve MACE-style Finite Model Finding. In Baumgartner, P., and Fermueller, C., eds., *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*.

Davis, M. 1981. Obvious Logical Inferences. In P., H., ed., *Proceedings of the 7th International Joint Conference on Artificial Intelligence* , 530–531.

Harper, R.; Honsell, F.; and Plotkin, G. 1993. A Framework for Defining Logics. *Journal of the ACM* 40(1):143–184.

Kaufmann, M.; Manolios, P.; and Strother Moore, J. 2000. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers.

McCune, W., and Shumsky-Matlin, O. 2000. Ivy: A Preprocessor and Proof Checker for First-Order Logic. In Kaufmann, M.; Manolios, P.; and Strother Moore, J., eds., *Computer-Aided Reasoning: ACL2 Case Studies*, number 4 in Advances in Formal Methods. Kluwer Academic Publishers. 265–282.

McCune, W. 2003a. Mace4 Reference Manual and Guide. Technical Report ANL/MCS-TM-264, Argonne National Laboratory, Argonne, USA.

McCune, W. 2003b. Otter 3.3 Reference Manual. Technical Report ANL/MSC-TM-263, Argonne National Laboratory, Argonne, USA.

Paulson, L., and Nipkow, T. 1994. *Isabelle: A Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer-Verlag.

Riazanov, A., and Voronkov, A. 2001. Splitting without Backtracking. In Nebel, B., ed., *Proceedings of the 17th International Joint Conference on Artificial Intelligence* , 611–617. Morgan Kaufmann.

Riazanov, A., and Voronkov, A. 2002. The Design and Implementation of Vampire. *AI Communications* 15(2-3):91–110.

Rudnicki, P. 1987. Obvious Inferences. *Journal of Automated Reasoning* 3(4):383–393.

Rudnicki, P. 1992. An Overview of the Mizar Project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, 311–332.

Schulz, S. 2002a. A Comparison of Different Techniques for Grounding Near-Propositional CNF Formulae. In Haller, S., and Simmons, G., eds., *Proceedings of the 15th Florida Artificial Intelligence Research Symposium*, 72–76. AAAI Press.

Schulz, S. 2002b. E: A Brainiac Theorem Prover. *AI Communications* 15(2-3):111–126.

Siekmann, J. 2002. Proof Development with OMEGA. In Voronkov, A., ed., *Proceedings of the 18th International Conference on Automated Deduction*, number 2392 in Lecture Notes in Artificial Intelligence, 143–148. Springer-Verlag.

Sutcliffe, G., and Suttner, C. 1998. The TPTP Problem Library: CNF Release v1.2.1. *Journal of Automated Reasoning* 21(2):177–203.

Sutcliffe, G.; Zimmer, J.; and Schulz, S. 2004. TSTP Data-Exchange Formats for Automated Theorem Proving Tools. In Zhang, W., and Sorge, V., eds., *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, number 112 in Frontiers in Artificial Intelligence and Applications. IOS Press.

Sutcliffe, G. 2000. SystemOnTPTP. In McAllester, D., ed., *Proceedings of the 17th International Conference on Automated Deduction*, number 1831 in Lecture Notes in Artificial Intelligence, 406–410. Springer-Verlag.

Sutcliffe, G. URL. The TSTP Solution Library. http://www.TPTP.org/TSTP.

Tammet, T. 1998. Towards Efficient Subsumption. In Kirchner, C., and Kirchner, H., eds., *Proceedings of the 15th International Conference on Automated Deduction*, number 1421 in Lecture Notes in Artificial Intelligence, 427–440. Springer-Verlag.

Veroff, R. 1996. Using Hints to Increase the Effectiveness of an Automated Reasoning Program: Case Studies. *Journal of Automated Reasoning* 16(3):223–239.

Weidenbach, C.; Brahm, U.; Hillenbrand, T.; Keen, E.; Theobald, C.; and Topic, D. 2002. SPASS Version 2.0. In Voronkov, A., ed., *Proceedings of the 18th International Conference on Automated Deduction*, number 2392 in Lecture Notes in Artificial Intelligence, 275–279. Springer-Verlag.