International Journal on Artificial Intelligence Tools

Semantic Derivation Verification: Techniques and Implementation

GEOFF SUTCLIFFE

Department of Computer Science, University of Miami
P.O. Box 248154, Coral Gables, FL 33124, USA
geoff@cs.miami.edu

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Automated Theorem Proving (ATP) systems are complex pieces of software, and thus may have bugs that make them unsound. In order to guard against unsoundness, the derivations output by an ATP system may be *semantically* verified by trusted ATP systems that check the required semantic properties of each inference step. Such verification needs 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 and testing in the GDV verifier.

Keywords: Derivation verification; Automated theorem proving.

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, typically mission critical, that it be sound, i.e., that it never reports 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 [11]. 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 (or, at least, expected) answer, confidence in the system's soundness may grow to a sufficient level. For example, it is commonly accepted that Otter [13] is sound, thanks to its extensive accepted usage by many researchers over many years. This is essentially the philosophical standpoint of [16]. 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 its parent formulae by the inference rule stated. This is the

approach taken in the IVY system [11], in the Omega proof checker after reduction to ND form [24], and that is planned for the in-house verifier for Vampire [18]. A serious disadvantage of syntactic verification is that the trusted system must implement all of the inference rules of all of the ATP systems whose derivations are 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 [9]. This is the approach taken by Bliksem's in-house verifier [2], whose proofs are then checked by Coq [1]. A weakness of this approach is the introduction of translation software, which may introduce or 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 trusted ATP systems. For example, deduction steps are verified by checking that the inferred formula is a logical consequence of its parent formulae. This is the approach taken in the low level checker of the Mizar system [21], that has been adopted by several in-house verifiers for contemporary ATP systems (e.g., SPASS [32]), that has been implemented using the "hints" strategy in Otter [31], and that forms the core of the verification process described in this paper.

Advantages of semantic verification include: independence of the trusted ATP systems 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 and Isabelle [14]); independence from the particular inference rules used in the derivation - see Section 2; and robustness to the preprocessing of the input formulae that some ATP systems perform - see Section 2.3. Semantic verification is able to verify any form of derivation, not only proofs[a], in which the required semantic properties of each inference step can be checked by a trusted system. Checking inference steps in which the inferred formula is a logical consequence of its parents is quite simple, while checking inference steps that have other semantic relationships, e.g., Skolemization and splitting, can be done with inference rule specific techniques.

All forms of verification that examine ATP systems' derivations include, at least implicitly, some *structural* verification that checks that inferences have been used correctly in the context of the overall derivation. A basic structural check is, e.g., that the specified parents of each inference step do exist in the derivation. Structural checking is the basis for the high level checker of the Mizar system. The necessary structural verification techniques that complement semantic verification are described in this paper.

This paper describes techniques for semantic verification of derivations, and reports on their implementation in the GDV verifier. The techniques were developed to verify derivations in classical 1st order logic, including the common approach of

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

proof-by-refutation in clause normal form (CNF) – see [19] for background material on reasoning in 1st order logic. However, the principles are more widely applicable. Section 2 describes how various parts of a derivation are semantically verified, and Section 3 describes the necessary structural verifications. Section 4 provides some details of the implementation of the techniques in the GDV verifier, and the results of testing GDV in a real-world application are given in Section 5. Section 6 examines the extent to which (these) verification techniques can be trusted. Section 7 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 interior nodes are formulae inferred from parent formulae, and whose root node is the final derived formula. 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 $false$ clause. Figure 1 shows the structure. The formulae 1 to 6 form the input problem, and are used to form the leaves L_i of the derivation. The internal formulae D_i are inferred from their parents, ending at the root node F.

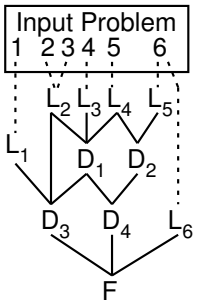

Fig. 1. Derivation DAG

Semantic verification of a derivation has two notionally distinct aspects. First, it is necessary to check that each leaf node is a formula that occurs in, or can be appropriately derived (as explained in Section 2.3) from the input problem. This ensures that the derivation is a solution to the input problem. Second, it is necessary to check the semantic properties of each inference step in the derivation. This is done by encoding the required semantic relationship between each inferred formula and its parent formulae into logical *obligations*, in the form of ATP problems. The obligations are then *discharged* by having trusted ATP systems solve the ATP problems. The required semantic relationship between an inferred formula and its parent formulae depends on the intent of the inference rule used, and correspond-

ingly different types obligations are produced. Most commonly an inferred formula is intended to be a logical consequence of its parent formulae, but in other cases, e.g., Skolemization and splitting, the inferred formula has a weaker relationship with its parents. A comprehensive list of inferred formula statuses is given in the SZS ontology [29]. Two types of obligations arise in this work: *theorem obligations*, which produce an ATP problem to prove a conjecture from some axioms, and *satisfiability obligations*, which produce an ATP problem to show a set of formulae is satisfiable.

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

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. For each inference of a logical consequence in a derivation, a theorem obligation to prove the inferred formula from its 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. The theorem obligation is handed to a trusted ATP system. If the trusted system proves the theorem, the obligation has been discharged. This process is shown in Figure 2, where the inferred formula D_3 must be proved from its parents $\{\mathsf{L}_1, \mathsf{L}_2, \mathsf{D}_1\}$. Theorem obligations are very typically simple ATP problems that are easily discharged.

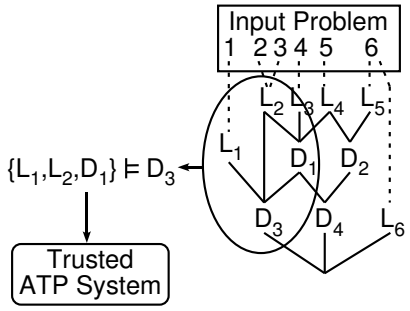

Fig. 2. Verifying a Logical Consequence

A special case of logical consequence is making a copy of a formula. Copying inferences can be verified by discharging a theorem obligation as above, but may also be checked directly by comparing the two formulae.

If the input problem is in FOF (First-Order Form - including quantifiers, etc, as opposed to CNF), and is converted to CNF for a proof by refutation, then one of the inferences is to negate the conjecture, i.e., the inferred clause is a counter theorem (in the SZS ontology) of its parent. In this case the theorem obligation

is to prove the negation of the inferred formula from its parent. (As is indicated in Section 5, although this kind of theorem obligation is simple to discharge in principle, modern CNF refutation based trusted ATP systems can find such theorem obligations difficult.)

There are inference steps that use a process of *proof by contradiction*, in which the assumption of the negation of a given formula seeds the derivation of the $false$ formula, from which the given formula can be inferred. Such inference steps must be structurally verified as described in Section 3, and then a theorem obligation to prove the inferred formula from the negation of the assumed negated formula, must be discharged.

The verification of logical consequences ensures the soundness of the inference steps, but does not check for *relevance*. As a contradiction in first order logic entails everything, an inference step with contradictory parents can soundly infer anything. An inference step with contradictory parents can thus always be the last in a derivation. A derivation containing an irrelevant inference is shown in Figure 3, in which $false$ should have been inferred directly from p and ∼p. If it is required that an inference step (that infers a formula other than $false$) is not irrelevant, a satisfiability obligation consisting of the parents of the inference must be discharged. This verification step should not be implemented during conversion from FOF to CNF when there is a single parent formulae that is (derived from) the negation of the conjecture - such parent formulae are correctly unsatisfiable when the conjecture is a tautology. 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 2.2.

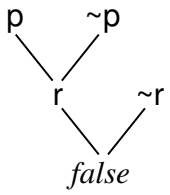

Fig. 3. Irrelevant Inferences

In practice (see Section 4), each attempt to discharge an obligation is constrained by a CPU time limit. Thus the failure to discharge a theorem obligation may be because it is not logically possible (indicating a fault in the derivation being verified), or because the obligation is too hard for the trusted ATP system within the CPU time limit. In order to try to differentiate between these two situations, if a trusted ATP system fails to prove a theorem obligation, a satisfiability obligation to show that the set consisting of the axioms and the negation of the conjecture of the theorem obligation is satisfiable, is attempted. If this is successful then it is

known that the theorem obligation cannot be discharged, and that there is a fault in the derivation being verified. Due to the semi-decidability of first order logic, satisfiability obligations also 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 Mace4 [12] or Paradox [3]. Second, a saturation of the axioms may be found using a saturating ATP system such as SPASS or EP [23]. Third, an attempt to show the axioms to be contradictory can be made using a refutation system. If that succeeds then the satisfiability obligation cannot be discharged. If it fails, it provides an incomplete assurance that the formulae are satisfiable. The incompleteness of this final test for satisfiability makes it unsuitable for some checks, e.g., the check for non-provability of theorem obligations. Its use must be appropriately controlled.

2.2. *Splitting*

ATP systems that build refutations for CNF problems may 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 subsubproblems, 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 [17]), and forms of *pseudo splitting* as implemented in Vampire and EP (also called *splitting without backtracking* in [17]). The verification of splitting steps has been omitted in existing ATP systems' in-house verifiers.

2.2.1. *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 ensured 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 by contradiction, with L (R) and the $false$ root of the subproblem's refutation 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 (L is commonly used by SPASS in the refutation of $S \cup \{R\}$). Semantic verification of explicit splitting and the anti-kid inferences are described here. The structural verification of explicit splitting is described in Section 3.

To verify a explicit splitting step's role in establishing the overall unsatisfiability of the original problem clauses, a theorem obligation to prove $\neg(L \vee R)$ from $\{\neg L, \neg R\}$ is discharged. The soundness of the split is then ensured as follows: The ATP system's (verified) refutations of the L and R subproblems show that every model of S is a model of neither L nor R, and thus every model of S is a model of

both $\neg L$ and $\neg R$. The discharge of the theorem 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 theorem obligation problem by CNF refutation 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 theorem obligation is to prove $\neg\forall X(p(X) \vee q(X))$ from $\{\neg\forall X p(X), \neg\forall X q(X)\}$. When the problem is converted to CNF, two Skolem constants are generated, producing the CNF problem $\{p(X) \vee q(X), \neg p(sk1), \neg q(sk2)\}$. This clause set is satisfiable, and the theorem obligation cannot be discharged.

While discharging the theorem obligation ensures 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 theorem 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 theorem 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 a theorem obligation to prove A from $\neg L$ ($\neg R$). The refutation of the L (R) subproblem shows that $\neg L$ ($\neg R$), and thus by modus ponens A is a logical consequence of S.

2.2.2. *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 EP 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 set, and any model of the modified clause set satisfies the original one [17,22]. The underlying justification for pseudo splitting is that it is equivalent to inferring logical consequences of the split clause with new definitional axiom(s): for (i) $t \Leftrightarrow \neg\forall L$, and for (ii) $t_1 \Leftrightarrow \neg\forall L$ 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 [17], can be treated with generalizations of the verification steps described here.

Pseudo splitting steps are verified by discharging theorem obligations that prove each of the replacement clauses from the split clause with the new definitional axiom(s). The definitional axioms should be freshly generated by the verifier, for otherwise the ATP system could produce incorrect split clauses and definitional axioms, but such that the replacement clauses can be derived from them. Discharging the theorem obligations confirms that the replacement clauses are logical consequences of the parent formulae, which is sufficient for the soundness of a CNF refutation.

It is also the case that the split clause is a logical consequence of the replacement clauses, which demonstrates the completeness of the splitting step.

As is the case with explicit splitting, the theorem obligations cannot all be discharged if L and R share variables. For example, in the (ii)nd form of pseudo splitting, a theorem obligation to prove $\neg t_1 \vee \neg t_2$ from the split clause with 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. One of the theorem obligations 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 theorem obligation cannot be discharged.

2.3. *Leaf Formulae*

The leaf formulae of a derivation should, optimally, be copies (modulo variable renaming) of formulae from the input problem. This can be checked directly. Some ATP systems perform preprocessing inferences on the input formulae, and do not retain the original formulae of the input problem, e.g., Gandalf [30] may factor and simplify input clauses before storing them in its clause data structure. In this case the leaf formulae are not copies of input formulae, but are logical consequences of input formulae. Such leaf formulae are verified by discharging a theorem obligation to prove the formula from input formulae. This technique can also be used to verify leaf formulae that are copies of input formulae. An advantage of always discharging a theorem obligation to verify a leaf formulae is that is provides consistent robustness to undocumented preprocessing inferences.

Discharging a theorem obligation to prove a leaf from the input formulae can normally be done directly, as shown in the example on the left hand side of Figure 4. In the example, the leaf formula $\mathsf{L_2}$ is proved directly from the input formulae {1, 2, 3, 4, 5, 6} (as the figure indicates, only input formulae 2 and 3 are needed to discharge the obligation, but that would not be known in advance). Directly proving the leaves from the input formulae is ineffective when the derivation being verified is a refutation, and the theorem obligations are discharged using a trusted refutation system (that would refute the set consisting of the input formulae and the negated leaf). Verifying a CNF refutation using a trusted CNF refutation system is a common example of this scenario. In this scenario, a refutation of the input clauses and the negated leaf formula does not necessarily mean that the leaf formula is a relevant logical consequence of the input formulae, because the refutation may be of the input formulae alone. A sound approach is to extract satisfiable maximal subsets of the input formulae (an input formula may be a member of more than one subset), and to then form alternative theorem obligations to prove the leaf from any one of the subsets. This is shown in the right hand side of Figure 4, where the leaf formula $\mathsf{L_2}$ is proved by refutation of $\sim\mathsf{L_2}$ with a satisfiable subset $\mathsf{Sat_i}$ of the input

formulae. If the trusted refutation system discharges any one of these obligations, then the leaf is a relevant logical consequence of the input formulae. If it fails, that may be because the leaf formula is derived from parents that are not all in one of the subsets. Alternative subsets may then be tried. In the implementation described in Section 4, subsets that are satisfied by the positive and negative interpretations are formed. If any formulae are not satisfied by either of these interpretations, those formulae are tested by a satisfiability obligation. If successful then all formulae are in at least one satisfiable subset, and leaf verification is typically successful. Otherwise some input formulae are unavailable for leaf verification, which may lead to verification failure for some leaves.

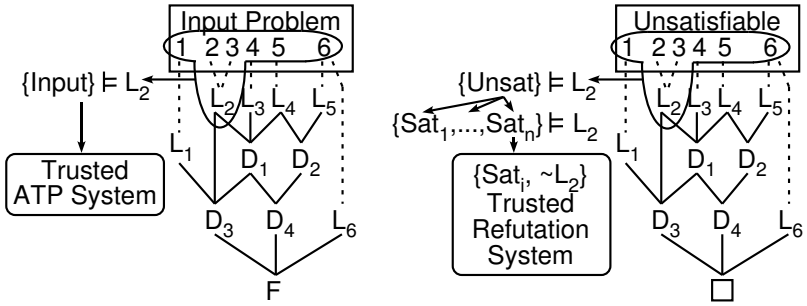

Fig. 4. Verification of Leaf Formulae

If the input problem is in FOF, and the derivation is a CNF refutation, some leaf clauses may have been formed with the use of Skolemization. Such leaf clauses are not logical consequences of the FOF input formulae. Skolemization steps can be incompletely verified by discharging a theorem obligation to prove the parent formula from the Skolemized formula. Although this is an incomplete verification, i.e., unsound Skolemization steps can pass this check, it does catch simple "typographical" errors.

3. Structural Verification

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.

For derivations that claim to be refutations, it is necessary to check that the $false$ clause has been derived. If explicit splitting is used, multiple such checks are

necessary, as described below. Inferences that use proof by contradiction to infer a formula, e.g., the inference of anti-kids in explicit splitting, must be checked to ensure that there are two parents, one of which is $false$ and the other of which is an ancestor of the $false$ parent (the assumed parent, as described in Section 2.1).

For refutations that use explicit splitting, special structural checks may be required. There are two reasons for an ATP system to do explicit splitting. 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 $false$ root descendant in the derivation. 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 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" really are new, and not used elsewhere in the refutation.

4. Implementation

The semantic verification techniques described in Sections 2 and 3 have been implemented in the GDV verification system. GDV is implemented in C, using the JJParser library's input, output, and data structure support. SystemOnTPTP [25] is used to run the trusted ATP systems. The overall architecture and use of GDV are shown in Figure 5.

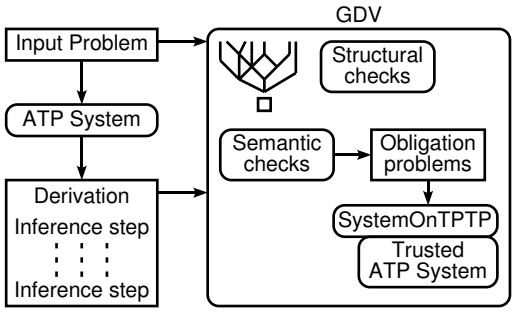

Fig. 5. The GDV Architecture

The inputs to GDV are: a derivation in TPTP format [29], the input problem in TPTP format, names of trusted ATP systems to discharge obligations, and a CPU time limit for the trusted ATP systems. Additionally, there are flags that control the level of debug output, force the verification process to continue even when a verification step fails (useful for finding multiple faults in a derivation), enable and disable the verification of leaves, enable and disable relevance checks on theorem obligations, indicate that the derivation should be a refutation (which enables the check for a $false$ root, and permits relevance failures when parent clauses are derived from the negation of the conjecture), and specify whether or not the outputs from the trusted systems should be retained (so that they are available, e.g., to form part of a certificate for the calling application process).

The implementation relies centrally on a function that takes as parameters an inferred formula, a list of it's parents, and the required semantic relationship between them. Depending on the required semantic relationship, the function prepares appropriate ATP problems and invokes one or more of the trusted ATP systems on the problems through SystemOnTPTP. SystemOnTPTP provides the necessary pre- and postprocessing to prepare a problem for a (trusted) ATP system, and to interpret the output from the ATP system. This makes it easily possible to use different trusted ATP systems, as specified on GDV's command line.

The verification proceeds in five phases. First, the derivation is read in and prepared for verification. The main part of this is preparation for verification of splitting steps, which is necessary because the derivations output by ATP systems do not always contain all the necessary information. In particular, it is necessary to generate the definitional axioms for the pseudo splitting done by EP[b], and to tag the left and right children, and the anti-kid produced, of explicit splits done by SPASS.

Second, structural verification is performed. This does some simple initial checks that the formulae in the derivation are uniquely named, and that all parents of each inference exist in the derivation and are annotated with information necessary for verification. The structural checks described in Section 3 are then performed. After structural verification all CNF formulae are converted to FOF for the remaining checks.

Third, inference rule specific verifications are performed. This includes checking explicit and pseudo splitting steps, and proofs by contradiction.

Fourth, leaf verification is performed. The first part of this is dividing the input formulae into maximal satisfiable subsets, as described in Section 2.1. First a satisfiability obligation to show that the entire input problem is satisfiable is attempted. If this succeeds then the entire input problem is used as the only satisfiable subset, otherwise the subsets are built. Then for each leaf of the derivation, ignoring definitional axioms inserted for pseudo splitting, a search for a copy in the input problem

[b]The author of EP has promised to output the definitional axioms natively, but at the time of writing (June 2005, EP version 0.82) this is not available.

is made. If the leaf is not found, then theorem obligations to prove the leaf from one of the satisfiable subsets are attempted.

Fifth, the inference rule non-specific inference steps are verified. This includes checking copied formulae, inferences of logical consequences, negation of conjectures (see Section 2.1 for these three), and the checking of Skolemization steps (see Section 2.3).

Obligations that are successfully discharged are reported. If an obligation cannot be discharged, or a structural check fails, GDV reports the failure. As is explained in Section 2.1, a failure to discharge an obligation does not always imply a fault in the derivation, which may motivate the user to use the command line flag to force the verification process to continue even when a verification step fails.

5. Testing

In the program certification process described in [5], a code generator has been extended to generate annotations in the code, e.g., loop invariants, that facilitate automatic checking of *safety conditions* on the code. A verification condition generator processes the annotated code, and produces a set of logical *safety obligations* that are provable if and only if the code is safe. An ATP system proves these obligations, and its proofs serve as part of the certificate for the safety of the program. For software certification purposes, users and certification authorities like the FAA must be assured – or better yet, given explicit evidence – that none of the individual tool components used in the certification process yield incorrect results. It is therefore useful to verify the proofs of the safety obligations produced by the ATP system. In [6], multiple ATP systems were evaluated on 366 safety obligations generated from the certification of programs generated by the AutoBayes [7] and AutoFilter [33] program synthesis systems. Of those 366 problems, 109 were selected for inclusion in the TPTP problem library [27], the standard library of test problems for testing and evaluating ATP systems. The 109 problems were selected based on the results of evaluating several state-of-the-art ATP systems against the problems, and were selected so as to be "difficult", i.e., with TPTP difficulty ratings strictly between 0.0 and 1.0 [28]. As a practical test and evaluation of the derivation verification techniques and implementation described in this paper, the proofs generated for the 109 problems by the ATP systems EP 0.82 and SPASS 2.1 have been verified.

Both EP and SPASS work by converting the axioms and negated conjecture to CNF, and then using clausal reasoning to find a refutation. The derivations output by EP include details of the FOF to CNF conversion, and the subsequent CNF refutation. The derivations are natively output in TPTP format. The derivations output by SPASS document the CNF refutation, but not the FOF to CNF conversion. The SPASS derivations are natively in DFG format [8], which is translated to TPTP format prior to verification. Both systems are based on the superposition calculus, but differ in the specific inference rules used. A notable difference is EP's use of pseudo-splitting and SPASS's use of explicit splitting. Additionally, the sys-

tems have quite different control heuristics. As a result, the derivations produced by the two systems have quite different characteristics.

For the verification of the EP proofs, GDV was configured to verify all aspects of each proof: the derivation was structurally verified, leaves were verified as being (possibly derived) from the input problem, all inferred formulae were semantically verified with relevance checking, and all splitting steps were verified. For the verification of the SPASS proofs, GDV was configured to verify selected aspects of each proof: leaves were not verified because SPASS does not document the FOF to CNF conversion, all inferred formulae were semantically verified but without relevance checking, all splitting steps were verified but the independence of the subproblems was not verified in the larger proofs because of the computational complexity, and the derivation was structurally verified (with the exception of the splitting aspect just mentioned). The trusted ATP systems were Otter 3.3 [13] for discharging theorem obligations, Paradox 1.1 [3] for finding finite models, and SPASS 2.1 for finding saturations.[c] The outputs from Otter, Paradox, and SPASS were retained to be available as part of a certificate. The verifications were done on Intel P4 2.8GHz computers with 1GB RAM, and running the Linux operating system (kernel version 2.4). The CPU time limit for the trusted ATP systems was 10s.

5.1. *Results*

Out of the 109 problems, EP can solve 48 and SPASS can solve 83, thus giving a total of 131 derivations to verify. The 48 problems solved by EP are a subset of those solved by SPASS, but the derivations are obviously different. Table 1 summarizes the results. The first column gives the overall values for the verification of the EP proofs, including both the verification of both the steps of FOF to CNF conversion and the inferences in the refutation. The next two columns split these values into the two parts. The final column gives the values for the verification of the inferences in SPASS' refutations. The last two columns are thus directly comparable. The first row shows the number of problems solved out of the 109, and the second row shows how many of those were completely verified by GDV with the checks described above. The next row gives the numbers of theorem obligations that were generated for the verifications and discharged by Otter. The next row gives the average number of theorem obligations per proof, and then the next five rows give their distribution, thus giving an indication of the distribution of the proof sizes. The next block of four rows gives the distribution of the CPU times taken by Otter to discharge the theorem obligations. The final row gives the number of finite models found in the relevance checking done for EP proofs.

The table shows that 46 of the 48 problems solved by EP were fully verified. Both failure cases were caused by Otter's inability to discharge obligations arising from

[c]Satisfiability tests, which employ saturation finding, are used only in the verification of leaves and relevance checking. As these checks were not done for the SPASS proofs, this is not a case of SPASS checking itself.

Table 1. Verification Results

	EP	EP-CNF	EP-Ref	SPASS
Problems solved	48			83
Proofs verified	46			83
Theorem obligations discharged	590	309	281	19737
Average obligations / proof	12.8	6.7	6.1	273.8
Number of obligations / proof				
0	0	0	19	0
1-10	35	38	22	52
10-100	10	8	4	13
100-1000	1	0	1	12
> 1000	0	0	0	6
Discharge time / obligation				
0.0-0.1s	208	123	85	19737
0.1-0.2s	362	172	190	0
0.2-0.3s	17	7	10	0
> 0.3s	3	3	0	0
Models found	361	140	221	-

steps in the FOF to CNF conversion. In particular, the obligations to verify the step that negates the conjecture, which entails proving the negation of the negation from the original, could not be discharged. The obligations are of the form $L \models \neg\neg L$, which Otter does not recognize as trivial, and if L is large the resultant CNF problem is too difficult for Otter. (SPASS is able to discharge these obligations.) All 83 of the SPASS proofs passed the verification checks chosen.

Most of the proofs require less than 10 obligations to be discharged, for both EP and SPASS. However, SPASS produces some very large proofs that consequently require a very large number of obligations to be discharged; the largest proof resulted in 3493 theorem obligations. This difference in distribution leads to a significant difference between the average numbers of obligations that had to be discharged per problem. At the same time, all of the SPASS obligations were discharged in almost no time. These figures indicate that SPASS proofs contain very many small, easily verified steps, while EP proofs have some larger steps. Note that 19 of the EP proofs were completed in the FOF to CNF conversion. EP's largest proof steps, requiring the longest times for verification (over 0.3s), are within the FOF to CNF conversion. There is some overhead starting Otter for each theorem obligation, and this dominates the wall clock time taken (i.e., the time the user has to wait for a proof to be verified). In this reality, it is preferable to have fewer but harder theorem obligations to discharge, as offered by EP.

Of the 590 theorem obligations discharged for EP, 361 had the parents verified as satisfiable, confirming the relevance of the parents to the inferred clause. The remaining 229 theorem obligations were not relevance checked because either one of the parent clauses was derived from the negation of the conjecture or the inferred clause was $false$.

Testing GDV on the program certification examples has provided a high level of confidence in the soundness of GDV (the soundness and completeness of derivation

verifiers, as opposed to the soundness of an ATP system whose derivations are verified, is discussed in Section 6). However, it was expected that all those derivations would be correct, and therefore failure to find fault with them was expected. In order to test the completeness of GDV, faults have been inserted by hand into derivations taken from the TSTP solution library [26], and GDV has successfully found these. In order to accumulate empirical evidence of the completeness of GDV it would be necessary to have a test library of faulty derivations - something that seems difficult to find or generate.

6. 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 [10], 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 to check the type correctness of the translated proof. The combination of the translator and Coq thus forms the trusted system. For semantic verification, trust is placed in the ATP systems that are used to discharge the obligations. 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 means that some obligations may not be discharged due to that incompleteness. In such a situation the verifier can possibly make an unsound claim to have found a fault in the derivation. Although undesirable, this is not catastrophic, as explained 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 - empirical evidence, provides an exit from this circle of doubt. The trust may be enhanced by making the trusted system as simple as possible. The simpler the trusted 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 simple. This may be a desirable feature of derivations, depending on the application. Thus, as the trusted system

gets weaker, the level of confidence in the trusted system rises and the inference steps in the derivation being verified must get simpler. The weakest form of trusted system seems to be one that implements something like "obvious inferences" [20,4]. In GDV 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.

In addition to the verification-level issues described above, there are questions regarding the underlying programming system, operating system, hardware, etc, upon which the verification process relies. These issues are addressed in [16], in which the bottom line is to build trust in the overall system through empirical testing in all environments. The use of GDV by the ATP community, e.g., in the CADE ATP system competitions [15], may provide this.

7. Conclusion

This paper describes techniques for semantic verification of derivations, focusing particularly on derivations in 1st order logic. The techniques have been implemented in the GDV system, resulting in a verifier that can verify any TPTP 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.

The most salient future work is to deal more completely with Skolemization steps in FOF to CNF conversion. Verification of Skolemization can be done directly in second order logic, and an approach that converts the resultant second order obligation to a first order one[d] is being investigated. Although the resultant first order obligation may not always be solvable, i.e., the approach is not complete, it is expected to be an improvement on the technique described in Section 2.3. As well as improving the verification aspects of this work, it is planned to integrate some user level derivation analysis into the process, e.g., a 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 in their derivation output.

References

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

[d]Conceived by Koen Claessen of Chalmers University of Technology.

Science. Springer-Verlag, 2004.
2. D. Bezem and H. de Nivelle. Automated Proof Construction in Type Theory using Resolution. *Journal of Automated Reasoning*, 29(3-4):253–275, 2002.
3. K. Claessen and N. Sorensson. New Techniques that Improve MACE-style Finite Model Finding. In P. Baumgartner and C. Fermueller, editors, *Proceedings of the CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, 2003.
4. M. Davis. Obvious Logical Inferences. In Hayes P., editor, *Proceedings of the 7th International Joint Conference on Artificial Intelligence* , pages 530–531, 1981.
5. E. Denney and B. Fischer. Correctness of Source-level Safety Policies. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *Proceedings of FM 2003: Formal Methods*, number 2805 in Lecture Notes in Computer Science, pages 894–913. Springer-Verlag, 2003.
6. E. Denney, B. Fischer, and J. Schumann. Using Automated Theorem Provers to Certify Auto-generated Aerospace Software. In M. Rusinowitch and D. Basin, editors, *Proceedings of the 2nd International Joint Conference on Automated Reasoning*, number 3097 in Lecture Notes in Artificial Intelligence, pages 198–212, 2004.
7. B. Fischer and J. Schumann. A Method for Generating Data Analysis Programs from Statistical Models. *Journal of Functional Programming*, 13(3):483–508, 2003.
8. R. Hähnle, M. Kerber, and C. Weidenbach. Common Syntax of the DFG-Schwerpunktprogramm Deduction. Technical Report TR 10/96, Fakultät für Informatik, Universät Karlsruhe, Karlsruhe, Germany, 1996.
9. R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the ACM*, 40(1):143–184, 1993.
10. M. Kaufmann, P. Manolios, and J. Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.
11. W. McCune and O. Shumsky-Matlin. Ivy: A Preprocessor and Proof Checker for First-Order Logic. In M. Kaufmann, P. Manolios, and J. Strother Moore, editors, *Computer-Aided Reasoning: ACL2 Case Studies*, number 4 in Advances in Formal Methods, pages 265–282. Kluwer Academic Publishers, 2000.
12. W.W. McCune. Mace4 Reference Manual and Guide. Technical Report ANL/MCS-TM-264, Argonne National Laboratory, Argonne, USA, 2003.
13. W.W. McCune. Otter 3.3 Reference Manual. Technical Report ANL/MSC-TM-263, Argonne National Laboratory, Argonne, USA, 2003.
14. L.C. Paulson and T. Nipkow. *Isabelle: A Generic Theorem Prover*. Number 828 in Lecture Notes in Computer Science. Springer-Verlag, 1994.
15. F.J. Pelletier, G. Sutcliffe, and C.B. Suttner. The Development of CASC. *AI Communications*, 15(2-3):79–90, 2002.
16. R Pollack. How to Believe a Machine-Checked Proof. Technical Report RS-97-18, BRICS, 1997.
17. A. Riazanov and A. Voronkov. Splitting without Backtracking. In B. Nebel, editor, *Proceedings of the 17th International Joint Conference on Artificial Intelligence* , pages 611–617. Morgan Kaufmann, 2001.
18. A. Riazanov and A. Voronkov. The Design and Implementation of Vampire. *AI Communications*, 15(2-3):91–110, 2002.
19. A. Robinson and A. Voronkov. *Handbook of Automated Reasoning*. Elsevier Science, 2001.
20. P. Rudnicki. Obvious Inferences. *Journal of Automated Reasoning*, 3(4):383–393, 1987.
21. P. Rudnicki. An Overview of the Mizar Project. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, pages 311–332, 1992.
22. S. Schulz. A Comparison of Different Techniques for Grounding Near-Propositional CNF Formulae. In S. Haller and G. Simmons, editors, *Proceedings of the 15th Florida*

Artificial Intelligence Research Symposium, pages 72–76. AAAI Press, 2002.
23. S. Schulz. E: A Brainiac Theorem Prover. *AI Communications*, 15(2-3):111–126, 2002.
24. J.H. Siekmann, C. Benzmüller, V. Brezhnev, L. Cheikhrouhou, A. Fiedler, A. Franke, H. Horacek, M. Kohlhase, A. Meier, E. Melis, M. Moschner, I. Normann, M. Pollet, V. Sorge, C. Ullrich, C.P. Wirth, and J. Zimmer. Proof Development with OMEGA. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction*, number 2392 in Lecture Notes in Artificial Intelligence, pages 143–148. Springer-Verlag, 2002.
25. G. Sutcliffe. SystemOnTPTP. In D. McAllester, editor, *Proceedings of the 17th International Conference on Automated Deduction*, number 1831 in Lecture Notes in Artificial Intelligence, pages 406–410. Springer-Verlag, 2000.
26. G. Sutcliffe. The TSTP Solution Library. http://www.TPTP.org/TSTP, URL.
27. G. Sutcliffe and C. Suttner. The TPTP Problem Library. http://www.TPTP.org, URL.
28. G. Sutcliffe and C.B. Suttner. Evaluating General Purpose Automated Theorem Proving Systems. *Artificial Intelligence*, 131(1-2):39–54, 2001.
29. G. Sutcliffe, J. Zimmer, and S. Schulz. TSTP Data-Exchange Formats for Automated Theorem Proving Tools. In W. Zhang and V. Sorge, editors, *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, number 112 in Frontiers in Artificial Intelligence and Applications, pages 201–215. IOS Press, 2004.
30. T. Tammet. Towards Efficient Subsumption. In C. Kirchner and H. Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction*, number 1421 in Lecture Notes in Artificial Intelligence, pages 427–440. Springer-Verlag, 1998.
31. R. Veroff. Using Hints to Increase the Effectiveness of an Automated Reasoning Program: Case Studies. *Journal of Automated Reasoning*, 16(3):223–239, 1996.
32. C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topic. SPASS Version 2.0. In A. Voronkov, editor, *Proceedings of the 18th International Conference on Automated Deduction*, number 2392 in Lecture Notes in Artificial Intelligence, pages 275–279. Springer-Verlag, 2002.
33. J. Whittle and J. Schumann. Automating the Implementation of Kalman Filter Algorithms. *ACM Transactions on Mathematical Software*, 30(4):434–453, 2004.