Practical Proof Checking for Program Certification

Geoff Sutcliffe[1], Ewen Denney[2], Bernd Fischer[2]
[1]University of Miami
geoff@cs.miami.edu
[2] USRA/RIACS, NASA Ames Research Center
{edenney,fisch}@email.arc.nasa.gov

Abstract

Program certification aims to provide explicit evidence that a program meets a specified level of safety. This evidence must be independently reproducible and verifiable. We have developed a system, based on theorem proving, that generates proofs that auto-generated aerospace code adheres to a number of safety policies. For certification purposes, these proofs need to be verified by a proof checker. Here, we describe and evaluate a semantic derivation verification approach to proof checking. The evaluation is based on 109 safety obligations that are attempted by EP and SPASS. Our system is able to verify 129 out of the 131 proofs found by the two provers. The majority of the proofs are checked completely in less than 15 seconds wall clock time. This shows that the proof checking task arising from a substantial prover application is practically tractable.

1 Introduction

Program certification tries to show that a given program achieves a certain level of quality, safety, or security. Its result is a *certificate*, i.e., independently checkable evidence of the properties claimed. Certification approaches vary widely, ranging from code reviews to full formal verification. The highest degree of confidence is achieved with approaches that are based on formal methods, and use logic and theorem proving to construct the certificates.

Over the last few years we have developed, implemented, and evaluated a certification approach that uses Hoare-style techniques to formally demonstrate the safety of aerospace programs that are automatically generated from high-level specifications [WSF02a, WSF02b, DFS04a, DFS04b, DFS05]. In that work, we have extended a code generator so that it simultaneously generates code *and* the detailed annotations, e.g., loop invariants, that enable fully automated safety proofs. A verification condition generator (VCG) processes the annotated code and produces a set of *safety obligations* that are provable if and only if the code is safe. An automated theorem prover (ATP) discharges these obligations and its proofs serve as certificates; we focus on *automated*—as opposed to interactive or (the auto-modes of) tactic-based—provers, since we are aiming at a fully automated "push-button" tool.

For 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 yield incorrect results and, hence, that the certificates are valid. The assurance can take a variety of different forms, e.g., tool pedigree, code inspections, paper-and-pencil proofs, or result checking. In this paper, we focus on automatically checking the correctness of the proofs generated by the ATP, which are crucial elements in our certification chain.

Proof checking is of course not necessary if the applied ATP is known to be correct. However, program certification is a difficult task that requires substantial "deductive power": the longest proof found during experiments involved more than 8000 inference steps. Consequently, simple "correct-by-inspection" theorem provers like leanTAP [BP95], or tactic-based provers built on top of a trusted kernel like Isabelle [Pau89], are not powerful enough.[1] Instead, we need to employ high-performance ATPs, which use complicated calculi, elaborate data structures, and optimized implementations. This makes formal verification of their correctness infeasible [MSM00]. One could argue that these provers have been extensively validated by the theorem proving community (e.g., the soundness checks required for participation in the CADE ATP System Competition (CASC), [PSS02]), so that a formal verification is not necessary. However, this argument by tool pedigree is weak. Most ATPs are under continuous development and single versions are never subjected to enough validation to achieve sufficient "social validation."[2] Moreover, the validation is necessarily incomplete. There have been several published instances of (unintentional) unsoundness in ATPs participating in the CASC, which have been detected only afterwards [SS99, Sut00b, Sut05].

As an alternative to formally verifying or extensively validating the ATPs, they can be extended to generate sufficiently detailed proofs that can be independently verified by a *proof checker*. The checker's function is to verify that the ATP's output is really a proof in the logical system in use. There are several approaches to proof checking, including the syntactic validation of Otter proof steps by Ivy [MSM00], higher-order proof term reconstruction in Isabelle [BN00], higher-order proof step checking in HOL [Won99], reducing proof checking to type checking as in Coq [BC04], and semantic derivation verification [SB05]. Semantic derivation verification has been used in this work. In semantic derivation verification, the required semantic properties of each proof step are encoded in one or more *proof check obligations* (typically an implication from the premises of the applied inference rule to its conclusion), which are then discharged by trusted ATPs. This way, the trusted ATP verifies the proof output of the original ATP. This approach is tractable because the correctness proof for each individual step in the original proof is substantially easier than the original proof itself, and thus within reach of the trusted ATP. For certification purposes, all proofs found by the trusted ATP become part of the certificate that is delivered by the overall certification system.

This paper describes how a semantic derivation verifier has been used to check the proofs that are found by ATPs for the safety obligations generated in the program certification process. The success of ATPs in discharging the safety obligations has been described in [DFS04a]. The success of (trusted) ATPs in verifying the resultant proofs is demonstrated here. Section 2 provides the necessary background on the program certification process, and Section 3 describes the semantic verification technique. Sections 4 and 5 provide empirical data that illustrate the success of the approach. Section 6 concludes, and discusses directions for future work.

[1] See `http://www.cl.cam.ac.uk/users/jeh1004/software/metis/performance.html` for benchmark data.

[2] The notable exception is Otter [McC03b], which has been essentially unchanged since 1996. However, previous experiments have shown that its performance is not sufficient for discharging the safety obligations we generate [DFS05].

2 Formal Program Certification

Formal program certification is based on the idea that the mathematical proof of some program property can be regarded as an externally verifiable certificate of this property. It is a limited variant of full program verification because it proves only individual properties and not the complete behavior, but it uses the same underlying technology.

2.1 Safety Policies

Formal program certification ensures that a program complies with a given *safety policy*. This is a formal characterization that the program does not "go wrong", i.e., does not violate certain conditions. A safety policy is defined by a set of Hoare-style inference rules and auxiliary definitions. The formal basis of this approach is explored in [DF03].

Safety policies exist at two levels of granularity. *Language-specific* policies can be expressed in terms of the constructs of the underlying programming language itself. They are sensible for any given program written in the language, regardless of the application domain. Typical examples of language-specific policies are array-bounds safety (i.e., each access to an array element to be within the specified upper and lower bounds of the array) and variable initialization-before-use (i.e., each variable or individual array element has been assigned a defined value before it is used). Various coding standards (e.g., restrictions on the use of loop indices) also fall into this category. *Domain-specific* properties are, in contrast, specific to a particular application domain and not applicable to all programs. These typically relate to high-level concepts outside the language. In principle, they are independent of the target programming language although, in practice, they tend to be be expressed in terms of program fragments. A typical example is matrix symmetry which requires certain two-dimensional arrays to be symmetric.

2.2 Generating Safety Obligations

For certification purposes, code must be *annotated* with information relevant to the selected safety policy. The annotations contain local information in the form of logical pre- and postconditions and loop invariants, which is then propagated through the code. The fully annotated code is then processed by a verification condition generator (VCG), which applies the rules of the safety policy to the annotated code in order to generate the safety conditions. As usual, the VCG works backwards through the code, and safety conditions are generated at each line. Our VCG has been designed to be "correct-by-inspection", i.e., to be sufficiently simple that it is straightforward to see that it correctly implements the rules of the logic. Hence, the VCG does not implement any optimizations, such as structure sharing on verification conditions or even apply any simplifications. Consequently, the generated verification conditions tend to be large and must be simplified. The more manageable simplified verification conditions can then processed by an ATP.

2.3 Certifiable Program Synthesis

As usual in Hoare-based approaches, the annotation effort can quickly become overwhelming and constitute a barrier for the adoption of the technique. This can be overcome by a *certifiable program synthesis system* that automatically generates the code *and* the detailed annotations from a high-level specification of the problem. The basic idea is to make the annotations part of the code templates so that they can be instantiated and refined in parallel with the code fragments. We have implemented this approach in two synthesis systems, AUTOFILTER [WS04], which generates state estimation code based on the Kalman filter algorithm, and AUTOBAYES [FS03], which generates statistical data analysis code.

Figure 1 shows the overall architecture of a certifiable program synthesis system. At its core is the original synthesis system that generates code for a given specification. The core system is extended for certification purposes (i.e., by the annotation templates), and augmented with a VCG, a simplifier, an ATP, and a proof checker. These components are described in more detail in [DF03, DFS04b, DFS04a].

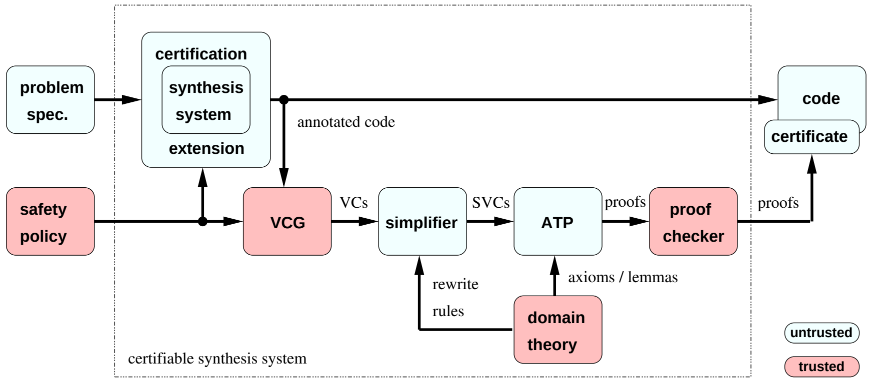

Figure 1: Certifiable program synthesis: System architecture

Similar to proof carrying code [NL98], the architecture distinguishes between trusted and untrusted components, shown in Figure 1 in red (dark grey) and blue (light grey), respectively. Components are called *trusted*—and must thus be correct—if any errors in them can compromise the assurance provided by the overall system. *Untrusted* components, on the other hand, are not crucial to the assurance because their results are double-checked by at least one trusted component. In particular, the correctness of the certifiable program synthesis system does not depend on the correctness of its two largest components: the original synthesis system (including the certification extensions), and the ATP; instead, we need only trust the safety policy, the VCG, and the proof checker.

3 Semantic Derivation Verification

The proofs produced by ATP systems can be considered more abstractly as derivations. For our purposes, 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 unique root node is the final derived formula. In semantic derivation verification, the required semantic properties of each inference step in a derivation are encoded in one or more *proof check obligations*. These are then *discharged* by trusted ATPs.

Derivation verification involves three notionally distinct phases. First, it is necessary to check the overall structure of the derivation. This ensures that the ATP output actually is a well-formed derivation DAG. Second, it is necessary to check that each leaf node is a formula that occurs in, or is derived from, the input problem. This ensures that the ATP actually solves the original problem. Third, it is necessary to check that each inferred formula has the required semantic relationship to its parents. This finally ensures that the proof is correct. 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 relation to its parents. A comprehensive list of inferred formula statuses is given in [SZS04]. Consequently, there are different forms of proof check obligations; currently GDV distinguishes between theorem obligations, satisfiability obligations, and leaf theorem obligations, which are explained in more detail in the following sections.

The main advantage of semantic derivation verification over other approaches is that it decouples proof checking from proof search—any ATP can serve as the trusted system that checks the output from the untrusted production system. Moreover, the approach is independent of the particular inference rules used in the production ATP, and is also robust with respect to any preprocessing of the input formulae that the production ATP might perform.

3.1 Logical Consequences and Relevance

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 [WB+02]. For each inference of a logical consequence in a derivation, a *theorem obligation* is formed; this formalizes that the inferred formula is a logical consequence of the parent formulae. 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 obligation is then handed to the trusted ATP system. If the trusted system solves the problem (i.e., finds a proof), the obligation has been discharged.

In practice (see Section 3.5), each attempt to discharge an obligation is constrained by a CPU time limit. Thus the failure to prove a theorem obligation may be because it is actually invalid (indicating a fault in the original derivation), or because the obligation is too hard for the trusted ATP system to prove within the CPU time limit. In order to try to differentiate between these two situations, if the trusted ATP system fails to prove a theorem obligation, GDV generates a *satisfiability obligation* to show that the set consisting of the parents and the negation of the inferred formula is satisfiable, which is then attempted by the trusted ATP. If this is successful then it is known that the theorem obligation cannot 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. If it is required that an inference step (that infers a formula other than a *false* formula) 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.

Due to the semi-decidability of first order logic, 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 [McC03a] or Paradox [CS03]. Second, a saturation of the axioms may be found using a saturating ATP system such as SPASS or EP [Sch02b]. 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.

3.2 Splitting

Many contemporary ATPs 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. There are several variants of splitting that have been implemented in specific ATPs, including *explicit splitting* as implemented in SPASS, and forms of *pseudo-splitting* as implemented in Vampire [RV01] and E. Verification of splitting inferences requires several theorem obligations to be discharged.

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 $S \cup \{R\}$. If both the subproblems have refutations (i.e., are unsatisfiable), then it is assured that the original problem is unsatisfiable. To verify an 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.

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 [RV01, Sch02a]. The underlying justification for pseudo-splitting is that it is equivalent to inferring logical consequences of the split clause and new definitional axioms: for (i) $t \Leftrightarrow \neg\forall L$, and for (ii) $t_1 \Leftrightarrow \neg\forall L$ and $t_2 \Leftrightarrow \neg\forall R$. Pseudo-splitting steps are verified by discharging theorem obligations that prove each of the replacement clauses from the split clause and the new definitional axiom(s).

3.3 Leaf Formulae

The leaf formulae of a derivation must occur in or be derived from the original problem—otherwise, the ATP solves a different problem. To verify this, *leaf theorem obligations* to prove each leaf formula from the input formulae must be discharged. An advantage of the semantic technique for verifying leaf formulae is that it is robust to some of the preprocessing inferences that are performed by ATP systems. For example, Gandalf [Tam98] 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 (i.e., first-order form including quantifiers, rather than CNF), and the derivation is a CNF 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. 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 step (i.e., unsound Skolemization steps can pass this check), it catches simple "typographical" errors and thus provides some additional assurance.

3.4 Structural Verification

All forms of proof checking also include, at least implicitly, some *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 derivations that claim to be CNF refutations, it is necessary to also check that the empty clause has been derived.

For refutations that use explicit splitting, two further structural checks are necessary. First, it is necessary to check that both subproblems have been refuted. Second, it is necessary to check that L (R) and its descendants are not used in 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.

3.5 Implementation

The semantic verification techniques described here have been implemented in the GDV system. GDV is implemented in C, using the JJParser library for input, output, and data structure support. The inputs to GDV are a derivation in TPTP format [SZS04], the original problem in TPTP format, a set of trusted ATPs to discharge the theorem obligations, and a CPU time limit for the trusted ATPs for each obligation. SystemOnTPTP [Sut00a] is used to run the trusted ATPs. Obligations that are successfully discharged are reported, and the output from the discharging is optionally retained for later inspection. If an obligation cannot be discharged, or a structural check fails, GDV reports the failure.

4 Experimental Setup

In [DFS05], we evaluate multiple ATPs on 366 safety obligations generated from the certification of programs generated by the AUTOBAYES and AUTOFILTER program synthesis systems. Of those 366 problems, 109 were selected for inclusion in the TPTP problem library [SS05], the standard library of test problem for testing and evaluating ATPs. The 109 problems were selected based on the results of evaluating several state-of-the-art ATPs against the problems, and were selected so as to be "difficult", i.e., with TPTP difficulty ratings strictly between 0.0 and 1.0 [SS01].

As a practical test and evaluation of the proof checking approach described in this paper, we scrutinized the proofs generated for these 109 problems by the ATPs EP (Version 0.82) [Sch02b][3] and SPASS (Version 2.1) [WB^{+}02]. Both EP and SPASS work by converting the axioms and the negated conjecture to CNF, and then using clausal reasoning to find a refutation.

[3] EP is a simple extension of E that produces explicit proofs.

The proofs output by EP include details of the FOF-to-CNF conversion, and the subsequent CNF refutation. The proofs are natively output in TPTP format. The proofs output by SPASS document the CNF refutation, but not the FOF-to-CNF conversion. The SPASS proofs are natively in DFG format, 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 systems have quite different control heuristics. As a result the proofs 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 ATPs used were Otter 3.3 [McC03b] for discharging theorem obligations, Paradox 1.1 [CS03] for finding finite models, and SPASS 2.1 for finding saturations.[4] The outputs from Otter, Paradox, and SPASS were retained, and are available as part of any certificate. The verifications were done on Intel P4 2.8GHz computers with 1GB RAM, and running the Linux 2.4 operating system. The CPU time limit for each discharge was 10s.

5 Experimental Results

Out of the 109 problems, EP can solve 48 and SPASS can solve 83, thus giving a total of 131 proofs to check. The 48 problems solved by EP are a subset of those solved by SPASS, but the proofs are obviously different. Table 1 summarizes the results. The first column gives the overall values for the verification of the EP proofs, including the verification of the steps converting from FOF to CNF 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's 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 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 numbers 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 steps in the FOF-to-CNF conversion. In particular, the obligations to verify the step that negates the conjecture,

[4]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.

	EP	EP-CNF	EP-Ref	SPASS
Problems solved	48			83
Proofs verified	46			83
Obligations discharged	590	309	281	19737
Average obligations / proof	12.8	6.7	6.1	273.8
Theorem 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	-

Table 1: Proof Checking Results

which entails proving the negation of the negation from the original, could not be discharged. All 83 of the SPASS proofs passed the verification checks chosen.

Most of the proofs require less than 10 obligations to be discharged, both for 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 slightly larger steps. Note that 19 of the EP proofs were completed in the FOF-to-CNF conversion, and 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). It is thus 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 theorems were not relevance checked because one of the parent clauses was derived from the negation of the conjecture.

6 Conclusions and Future Work

In this paper, we have described and evaluated a semantic derivation verification approach to proof checking. The evaluation, which is the main contribution of the paper, is based on 109 safety obligations arising in the certification of auto-generated aerospace code.

The results are encouraging. Our system is able to verify 129 out of the 131 proofs found by EP and SPASS, showing that the proof checking task is practically tractable. The vast majority of the proof check obligations are discharged in less than 0.1 seconds. The majority of the proofs are checked completely in less than 15 seconds wall clock time, although some of the longer proofs cannot be verified completely and even the partial checks can take more than five minutes. Moreover, as a consequence of the substantial overheads our current implementation incurs for intermediate format transformation steps, proof checking often requires *more* wall clock time than the actual proof search. However, this is not a fundamental limitation and could easily be changed by an optimized implementation.

There is still a lot of room for improvement. The verification of some trivial proof steps in the FOF-to-CNF conversion failed. The corresponding obligations were of the form $L \models \neg\neg L$, where L is very large. The trusted ATP (i.e., Otter) does not recognize this form and produces a very difficult CNF obligation. Using SPASS as the trusted ATP, however, solves this problem. Similarly, some forms of structural verification are very expensive, in particular for the large proofs found by SPASS. Moreover, the approach relies on the production ATP generating well documented proofs. Currently, only EP satisfies this criterion. SPASS proofs are missing the FOF-to-CNF conversion, and Vampire does not record the negation of the original goal, which makes its proofs uncheckable. Finally, we have evaluated our techniques only for ATPs based on the superposition calculus. Future work will thus be concerned with systems based on other calculi, such as non-clausal resolution or model elimination.

Derivation verification does not provide absolute assurance. The biggest gap is the verification of Skolemization steps, which are only satisfiability-preserving. While the full verification of such steps (and clausification in general) requires further research and experimentation, the partial verification provided here already gives some additional assurance. Other potential gaps are that the construction of the proof check obligations is wrong, and that the trusted ATP contains errors. Moreover, derivation verification as described here only addresses errors in the proofs found by the ATPs but not any errors in the construction and, in particular, simplification of the original verification conditions. However, similar techniques can also be applied to double-check the rewrite engine and rules used for simplification.

Ultimately, however, in order to convince users of the validity of the overall certification process, there needs to be some explicit linking or tracing between the logical entities and the program being certified. In [DF05], we describe a browser which enables a two-way linking between the verification conditions and the individual statements of the annotated program. We are also developing an extension to the VCG which adds "semantic markup" to formulas in the form of labels which explain their origin and meaning. The accumulated labels can then be converted into text and used to interpret the generated verification conditions. We would like to combine tracing, textual rendering, and proof checking into an integrated environment for certification.

References

[BC04] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. EATCS, 2004.

[BN00] S. Berghofer and T. Nipkow. “Proof Terms for Simply Typed Higher Order Logic”. In *Theorem Proving in Higher Order Logics*, pp. 38–52, 2000.

[BP95] B. Beckert and J. Posegga. “lean$T^A P$: Lean Tableau-based Deduction”. *J. Automated Reasoning*, **15**(3):339–358, 1995.

[CS03] K. Claessen and N. Sorensson. “New Techniques that Improve MACE-style Finite Model Finding”. In P. Baumgartner and C. Fermueller, (eds.), *Proc. CADE-19 Workshop: Model Computation - Principles, Algorithms, Applications*, 2003.

[DF03] E. Denney and B. Fischer. “Correctness of Source-Level Safety Policies”. In K. Araki, S. Gnesi, and D. Mandrioli, (eds.), *Proc. FM 2003: Formal Methods*, *Lect. Notes Comp. Sci.* 2805, pp. 894–913. Springer, 2003.

[DF05] E. Denney and B. Fischer. “A Program Certification Assistant Based on Fully Automated Theorem Provers”. In *Proc. Intl. Workshop on User Interfaces for Theorem Provers*, Electronic Notes in Theoretical Computer Science, 2005.

[DFS04a] E. Denney, B. Fischer, and J. Schumann. “Using Automated Theorem Provers to Certify Auto-Generated Aerospace Software”. In M. Rusinowitch and D. Basin, (eds.), *Proc. 2nd Intl. Joint Conf. Automated Reasoning*, *Lect. Notes Artificial Intelligence* 3097, pp. 198–212. Springer, 2004.

[DFS04b] E. Denney, B. Fischer, and J. Schumann. “An Empirical Evaluation of Automated Theorem Provers in Software Certification”. In *Proc. IJCAR 2004 Workshop on Empirically Successful First Order Reasoning*, 2004.

[DFS05] E. Denney, B. Fischer, and J. Schumann. “An Empirical Evaluation of Automated Theorem Provers in Software Certification”. *Intl. Journal of AI Tools*, 2005. To appear.

[FS03] B. Fischer and J. Schumann. “AutoBayes: A System for Generating Data Analysis Programs from Statistical Models”. *J. Functional Programming*, **13**(3):483–508, May 2003.

[McC03a] W. McCune. Mace4 Reference Manual and Guide. Technical Report ANL/MCS-TM-264, Argonne National Laboratory, Argonne, USA, 2003.

[McC03b] W. McCune. Otter 3.3 Reference Manual. Technical Report ANL/MSC-TM-263, Argonne National Laboratory, Argonne, USA, 2003.

[MSM00] 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, (eds.), *Computer-Aided Reasoning: ACL2 Case Studies*, Advances in Formal Methods 4, pp. 265–282. Kluwer Academic Publishers, 2000.

[NL98] G. C. Necula and P. Lee. “The Design and Implementation of a Certifying Compiler”. In K. D. Cooper, (ed.), *Proc. ACM Conf. Programming Language Design and Implementation 1998*, pp. 333–344. ACM Press, 1998. Published as SIGPLAN Notices 33(5).

[Pau89] L. C. Paulson. “The Foundation of a Generic Theorem Prover”. *Journal of Automated Reasoning*, **5**(3):363–397, 1989.

[PSS02] F. J. Pelletier, G. Sutcliffe, and C. B. Suttner. "The Development of CASC". *AI Communications*, **15**(2-3):79–90, 2002.

[RV01] A. Riazanov and A. Voronkov. "Splitting without Backtracking". In B. Nebel, (ed.), *Proc. 17th Intl. Joint Conf. Artificial Intelligence*, pp. 611–617. Morgan Kaufmann, 2001.

[SB05] G. Sutcliffe and D. Belfiore. "Semantic Derivation Verification". In I. Russell and Z. Markov, (eds.), *Proc. 18th Florida Artificial Intelligence Research Symposium*. AAAI Press, 2005.

[Sch02a] S. Schulz. "A Comparison of Different Techniques for Grounding Near-Propositional CNF Formulae". In S. Haller and G. Simmons, (eds.), *Proc. 15th Florida Artificial Intelligence Research Symposium*, pp. 72–76. AAAI Press, 2002.

[Sch02b] S. Schulz. "E: A Brainiac Theorem Prover". *AI Communications*, **15**(2-3):111–126, 2002.

[SS99] G. Sutcliffe and C. Suttner. "The CADE-15 ATP System Competition". *Journal of Automated Reasoning*, **23**(1):1–23, 1999.

[SS01] G. Sutcliffe and C. Suttner. "Evaluating General Purpose Automated Theorem Proving Systems". *Artificial Intelligence*, **131**(1-2):39–54, 2001.

[SS05] G. Sutcliffe and C. Suttner. The TPTP Problem Library. `http://www.TPTP.org`.

[Sut00a] G. Sutcliffe. "SystemOnTPTP". In D. McAllester, (ed.), *Proc. 17th Intl. Conf. Automated Deduction*, *Lect. Notes Artificial Intelligence* 1831, pp. 406–410. Springer, 2000.

[Sut00b] G. Sutcliffe. "The CADE-16 ATP System Competition". *Journal of Automated Reasoning*, **24**(3):371–396, 2000.

[Sut05] G. Sutcliffe. "The IJCAR-2004 Automated Theorem Proving Competition". *AI Communications*, **18**(1), 2005.

[SZS04] G. Sutcliffe, J. Zimmer, and S. Schulz. "TSTP Data-Exchange Formats for Automated Theorem Proving Tools". In W. Zhang and V. Sorge, (eds.), *Distributed Constraint Problem Solving and Reasoning in Multi-Agent Systems*, Frontiers in Artificial Intelligence and Applications 112, pp. 201–215. IOS Press, 2004.

[Tam98] T. Tammet. "Towards Efficient Subsumption". In C. Kirchner and H. Kirchner, (eds.), *Proc. 15th Intl. Conf. Automated Deduction*, *Lect. Notes Artificial Intelligence* 1421, pp. 427–440. Springer, 1998.

[WB$^+$02] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topic. "SPASS Version 2.0". In A. Voronkov, (ed.), *Proc. 18th Intl. Conf. Automated Deduction*, *Lect. Notes Artificial Intelligence* 2392, pp. 275–279. Springer, 2002.

[Won99] W. Wong. "Validation of HOL Proofs by Proof Checking". *Formal Methods in System Design: An International Journal*, **14**(2):193–212, March 1999.

[WS04] J. Whittle and J. Schumann. "Automating the Implementation of Kalman Filter Algorithms". *ACM Transactions on Mathematical Software*, **30**(4):434–453, December 2004.

[WSF02a] M. Whalen, J. Schumann, and B. Fischer. "AutoBayes/CC — Combining Program Synthesis with Automatic Code Certification (System Description)". In A. Voronkov, (ed.), *Proc. 18th Intl. Conf. Automated Deduction*, *Lect. Notes Artificial Intelligence* 2392, pp. 290–294. Springer, 2002.

[WSF02b] M. Whalen, J. Schumann, and B. Fischer. "Synthesizing Certified Code". In L.-H. Eriksson and P. A. Lindsay, (eds.), *Proc. Intl. Symp. Formal Methods Europe 2002: Formal Methods—Getting IT Right*, *Lect. Notes Comp. Sci.* 2391, pp. 431–450. Springer, 2002.