Trusted Extensions of Interactive Theorem Provers: Workshop Summary

Konrad Slind

Advanced Technology Center, Rockwell Collins

Abstract. A fundamental strength of interactive theorem provers (ITPs) is the high degree of trust one can place in formalizations carried out in them. ITPs are usually also extensible, both at the logic level and at the implementation level. There is consequently a substantial body of existing and ongoing research into the extension of ITPs while preserving trust.

In order to survey existing and new work in this area, we organized the *Trusted Extension of Interactive Theorem Provers* (TEITP) workshop.[1] As a result of the workshop we have been able to get an overview of the approaches taken by most of the major ITPs. In this document we summarize the meeting and provide some background information for readers unfamiliar with the area.

Introduction

The focus of the TEITP workshop arises from a conflict. On one hand, users often want to extend the power of an ITP, *e.g.*, by tapping into the power provided by third-party automated theorem proving software, or by writing their own proof procedures. On the other hand, such activity can obscure the trust story of the ITP, since the correctness of the combined or extended system then needs to be justified. The goal is to allow extensions of an ITP while preserving its trust proposition.

In Section 1 we provide some detail about the workshop. Section 2 introduces the major ideas and approaches to trusted extension that have been taken. Section 3 presents developer-provided descriptions of the trust architecture of several ITPs (Isabelle, ACL2, ProofPower, HOL-4, Coq, and PVS). Section 4 provides a supplemental bibliography for the area.

This document was written by Konrad Slind following discussions at the TEITP workshop and comments by Matt Kaufmann and Mike Gordon. System trust descriptions are included with the permission of the developers (whose names are included with the descriptions in Section 3).

[1] Held August 11-12, 2010, in Cambridge, England. Organizers:Matt Kaufmann (UT Austin), Konrad Slind (Rockwell Collins), and Mike Gordon (Cambridge University).

1 Workshop details

The workshop webpage

`http://www.cs.utexas.edu/users/kaufmann/itp-trusted-extensions-aug-2010/`

provides a great deal of information beyond this summary. In particular, it has slides for all the talks, the participant list, and a transcript of an extended discussion section held at the end of the workshop, along with other discussions that occurred later.

The workshop presentations were of two kinds: (1) representatives of ACL2, Coq, Isabelle, HOL-4, and PVS presented the trust proposition for their systems; and (2) talks on trust extensions and applications of trust in certification processes.

Discussion session The discussion session ranged beyond the workshop topics to encompass more general aspects of trust and trustworthiness of ITPs. The discussion covered topics such as how to leverage the high level of trustworthiness of ITPs, how to characterize or quantify trustworthiness, *etc.* We were fortunate to have (in Kaufmann) a typist who could keep up with the discussion and the resulting transcript provides a valuable supplement to the formal presentations. Participants were given the opportunity after the workshop to review the notes for accuracy and resulting corrections were incorporated. See the webpage for the full text.

2 Approaches to Trusted Extension

Some fundamental approaches to ITP extension have emerged. ITP developers and users typically use a selection of the following techniques:

1. Careful extension by experts
2. LCF style
3. Proof objects
4. Metafunctions
5. Implementation verification
6. Logic interpretation

We now give a little more detail on each of these. Of course, a given system may employ more than one of these approaches.

2.1 Careful extension by experts

In this approach, the representations and procedures by which a theorem may be proved in an ITP are strictly controlled and may only be revised or extended by an expert, usually one of the system developers. Therefore, the correctness of an extension to the system depends on the care taken by the developers who implement the extension. ACL2 is a prime example of a system of this kind. (Of course, there is always ongoing activity in any implementation to improve the code and underlying algorithms. Moreover, all ITPs provide facilities for extending the logic by definitions, and for improving the power of automated procedures by adding lemmas. However, an ITP based on the *Careful Extension* model will typically not support the implementation of a new proof procedure, except by one of the developers.)

Some examples of such extensions:

- In [10], Boyer and Moore discuss the issues involved with integrating a linear arithmetic decision procedure into the Boyer-Moore theorem prover, paying particular attention to the interplay between conditional rewriting and the decision procedure.
- In [23] an integration of model-checking into PVS is described.
- In [2], Armand, Gregoire, Spiwack, and Thery extend Coq by adding machine integers and arrays as primitive logical objects.

2.2 LCF style

This approach relies on the data abstraction facilities of the host programming language. An abstract datatype of theorems is the central concept, with the constructors of the abstract type being just the axioms and primitive rules of inference of the logic. Typically, the rules of inference in an LCF-style system are extremely simple—at the level of *modus ponens*, for example. The host programming language (ML is popular) is required to have a strong type system, so that arbitrary user-written proof procedures are guaranteed to ultimately reduce to primitive inferences; thus user code cannot compromise the soundness of the kernel implementation of the logic.

Isabelle, Proof Power, HOL Light, and HOL-4 are the main current examples of LCF-style systems. The reference manuals for these systems provide background and overviews of how the LCF style is implemented.

The requirement that all proof steps be justified by passing through a simple logical kernel can lead to relatively inefficient proof procedures in LCF style systems. The thesis of Boulton [7] proposes techniques to ameliorate the problem, and there is another good discussion of the issues in [16]. As mentioned, ML has to date been the programming language of choice when implementing an LCF style system. However, Pollack [21] offers reasons why a dependently-typed programming language may be preferable.

2.3 Proof objects

In this approach, a theorem prover that successfully proves a formula also produces a *proof object*, which is a data structure. This data structure can then be analyzed to see if it is in fact a formal proof of the formula. Proof objects decouple the proof of a theorem from the system it is generated in. Thus, they provide an extremely high level of assurance: a proof object can be, for example, checked by multiple independently implemented proof checkers. Some provers only emit a *witness*, or *certificate*, from which a full formal proof may be synthesized.

A long-standing issue with proof objects is their size: naive datastructures representing proofs are too large. On the other hand, sophisticated representations make it harder to write proof checkers.

Several different 'use-cases' have been developed for proof objects:

– Constructive logics, *e.g.*, Coq [5] and Agda [9], which are justified by the Proofs-as-Programs principle, explicitly incorporate proofs in the logic of the ITP. A formula in such a logic is a type, and a proof that the type is inhabited amounts to a program that obeys the formula. Checking proof objects is the essential component in the correctness argument for such systems. Note that some systems, *e.g.*, Coq, avoid some of the issues with the size of proof objects by treating evaluation as a single step.
– Isabelle also provides proof objects [4], although they are not crucial to its trust story, as Isabelle is an LCF design.
– A variety of SAT, FOL, and SMT provers generate proof objects, which are used to guide automatic proof generation in a variety of ITPs. This allows an ITP to extend its automation with the power of current automated provers without compromising the trust story [6, 24].
– Proof objects have also been used for translating theories between different implementations of HOL. Recent work in this area is the *OpenTheory* implementation of Hurd [19]. Earlier work in this area has been done by Skalberg and Obua [20] in order to translate HOL-4 theories to Isabelle/HOL.

2.4 Metafunctions

This technique is originally due to Boyer and Moore[11] , but has been rediscovered several times since. It is sometimes called *reflection* or *partial reflection*.[2] The approach is based on an internalization of the syntax and semantics of a subset[3] S of the terms[4] of the ITP's logic. Suppose the subset S can be formalized as a object-logic datatype T, and that a given object-logic algorithm

$$A : T \to T$$

is formalized and proved—inside the theorem prover—to transform elements of T to other elements of T having the same semantics (think: normalization). This allows the following deduction: if a term t lying in S is encountered, it can be mapped to T, and then A is applied; the result can be translated from T back to a formula t' in the object logic and the theorem

$$\vdash t = t'$$

[2] Note that this terminology overlaps with other usage of the word *reflection*, which is an overloaded term in this area. See Section 2.5.
[3] S must be a *proper* subset of the formulas of the logic, by Tarski's Undefinability of Truth theorem.
[4] The metafunctions approach also works for formulas.

derived. In an implementation, there are significant choices to be made about how to map terms between S and T and also in how to run A. For example, the execution of A can be done purely by deductive steps, but it can be far faster to compile and run the code for A in the metalanguage. However, in logics such as higher order logic, taking this speedy route can mean that the equality of t and t' needs to be asserted as a new axiom since running the compiled code steps out of the inference kernel.

Howe [18] developed an approach similar to metafunctions in the Nuprl type theory, going as far as formalizing and verifying a term rewriter. Later work in the Nurpl group [1] explored more recondite aspects of reflection, where the logic featured an explicit rule of reflection. An approach to metafunctions in Coq was presented in [8] and it has become a heavily exploited technique, for example in Gonthier's proof of the Four Color Theorem [14]. In Isabelle/HOL the metafunctions approach has been applied to incorporate a linear arithmetic decision procedure [12].

2.5 Implementation verification

One way to trust an ITP extension is to prove it correct, as we saw with the metafunctions approach. However, that method essentially depends on the formalization of a subclass of terms. The lecture at TEITP by Kaufmann introduced a new technique, known as `defattach`, which can be understood as a methodology for refining the kernel code of an ITP. Using `defattach` allows users to safely override existing prover functionality, provided that the invariants specified for that functionality by the kernel developers are formally proved to be preserved. In this way, code to be 'attached' need not identify a set of terms comprising its domain, instead, it has to—provably—at least maintain the invariants of the old code.

Harrison verified a formal model of the HOL Light kernel [17] against an abstract set-theoretic specification. On the basis of that work, one could prove proof procedures to be correct, *e.g.*, by showing that a proof procedure proves no more theorems than the kernel inference rules. Such approaches are also known as *procedural reflection*. Harrison's paper on reflection [16] gives a comprehensive overview of the topic, with an extensive bibliography.

Another reflective approach is developed in Jared Davis' *Milawa* prover[13], which is obtained by bootstrapping from a simple prover (so simple that it can be seen to be correct by inspection) to a system that provides a significant subset of the functionality of the ACL2 prover. Each of the eleven bootstrapping steps is formally shown to preserve the set of provable formulas and thus the final, complex, prover is just as trustworthy as the simple initial implementation.

The work of Harrison and Davis can be seen to be complementary: Harrison verifies a kernel, while Davis starts from an assumed correct kernel.

In his TEITP presentation, Shankar proposed an ITP design that uses a variety of implementation verification ideas, but focuses on the use of proof certificates. Instead of the proof translation idea described earlier, he advocates a methodology based on the verification of proof certificate checkers.

2.6 Logic Interpretation

Instead of interpreting proofs, one can move to the level of logic interpretation. For example, in recent years a link between ACL2 and HOL-4 has been created. The link is achieved at both a physical level, so that results from one system can be shipped to the other, and at a logical level; the ACL2 logic has been given a formal model in HOL-4. A practical result of this embedding is that, if ACL2 proves a formula P in a theory $\mathcal{T}$ then there is also a HOL proof of P (translated to the HOL theory of ACL2) in $\mathcal{T}$ (also translated to HOL). The embedding thus constitutes a *one-time* heavyweight effort; subsequently, theories and theorems proved in ACL2 may be automatically translated to HOL without having to justify them by re-running proofs. An example of the application of this system is a proof of the correctness of a model-checking algorithm [15].

3 System Trust Explanations

Here are contributions, made after the workshop ended, explaining major ITP systems' approaches to trust.

3.1 Isabelle

(Larry Paulson, with feedback from Burkhart Wolff):

Isabelle uses the LCF approach to soundness. There is a proof kernel although somewhat larger than that of a typical HOL system because it includes an implementation of higher-order unification. Isabelle can generate full proof terms that can be checked independently, although it normally generates minimal proof terms in order to save space. External reasoners can be used as trusted oracles, but all such uses are labelled as such in the proof term even if minimum terms are enabled. Isabelle includes computational reflection, and the results of computations can be accepted as theorems. This involves trusting about 2000 lines of code that describes how to translate executable fragments of higher-order logic into one of several functional programming languages. Of course, the corresponding language implementation must also be trusted. Isabelle proofs do not generate theorems from such computations unless the user asks for this explicitly, and such theorems are always labelled as coming from an oracle.

3.2 ACL2

(Matt Kaufmann and J Moore):

The ACL2 code base consists of approximately 240,000 lines of code – 10.5 MB – as of July, 2010. There is no subset identified as a "trusted" code base. While more than 1/6 of the lines are comment lines, and the source code also contains documentation strings (generating more than 1700 pages if printed), nevertheless this is a large base of code to trust. Trust thus rests on the two individuals who maintain the system, who apply their experience (over 70 years

combined in logic and automated reasoning) and passion towards producing a system that follows its careful logical foundations (see
`http://www.cs.utexas.edu/users/moore/publications/acl2-papers.html#Foundations`).

Users can extend the power of the system, but only (if the two system maintainers have done their job well) in ways that respect its foundations. Such extension facilities include a definitional principle, introduction of (witnessed) constrained functions, and a powerful but safe macro facility. Many trustworthy features are provided to extend the behavior of the automatic theorem prover, including use of previously proved theorems as rules of various sorts, reflection principles, and both static and computed hints. Computation is used during proof.

Users have the power to connect to external tools (see for example
`http://www.cs.utexas.edu/users/sandip/publications/clauseprocessors/main.html`), or even to subvert the system, provided they employ trackable "trust tags" that must be made explicit when verifying ("certifying") a collection of input files ("books"). For maximum trust, ACL2 users should certify all books of a proof development, in an environment without other processes writing to the ACL2 or user space in the file system, and either without the use of trust tags or with explicit inspection of their uses.

3.3 ProofPower

(Rob Arthan):

For the record, the ProofPower trust story is the LCF story, backed up by a formal specification (but no proof) of the HOL language, logic and critical properties of a proof tool. The support for Z in ProofPower is an example of an extension in the sense of a semantic embedding of a new object language with a trust story piggy-backing on the trust story of ProofPower-HOL. The trust story for both ProofPower-HOL and ProofPower-Z includes the fact that the system is interactively programmable, so an evaluator who wants to can really dig in and pick syntax to pieces, which, in principle, mitigates the risk of problems of the sort recently discussed in Freek Wiedijk's (somewhat light-hearted) paper on what he calls "Pollack Consistency" [25]. In practice, the fact that you can easily code up things like reports on which definitions have been proved consistent and which have been given as axioms seems to be appreciated by commercial users. (Evaluators really like checklists!).

3.4 HOL-4

(Konrad Slind):

HOL-4 is an implementation of the so-called LCF style, in which an abstract datatype is used to implement the primitive rules of inference of a logic, higher order logic (HOL) in this case. On top of this kernel, arbitrary ML programming by arbitrarily naive users may be used to construct theorems. If the kernel is a correct implementation, only genuine HOL theorems may thereby result. The kernel is small enough that its correctness may be established by inspection, informal proof, or even formal proof.

Interfacing with an external proof tool is typically performed by translating witnesses provided by the external tool into HOL proofs, which are replayed through the HOL kernel, obtaining the desired proof. However, occasionally it is convenient to directly accept the judgement of an external tool. In those cases, one can create a "tagged" theorem, where the tag designates the external tool that has been applied. Tags accumulate through primitive inference steps, with the result that if a theorem th_1 results from primitive inference on a tagged theorem th_0, then th_1 also becomes tagged. Consequently, only a theorem without a tag can be truly said to have been obtained purely by HOL inference steps.

HOL-4 provides *persistent theories*, in which the results of a theory development can be cached on disk, in a readable format. Since the disk representation of a theory can be maliciously edited, tags are also attached to definitions and theorems imported from a persistent theory.

In summary, one way to believe that a claimed theorem in HOL-4 has a formal proof in the HOL logic is to do the following:

1. Arrange all theories needed to prove the theorem in dependency order;
2. Execute the theories, one at a time, in a single session (to avoid using persistence);
3. Check that the theorem of interest is indeed proved, and also check that it has no tags attached.

Another way to believe that a claimed theorem in HOL-4 has a formal proof is to check a proof object created by a run of HOL-4. Some argue that checking such proof objects provides a stronger level of trust than an LCF-style kernel. Code supporting the generation and checking of proof objects has been implemented several times for HOL-4 but its use is not yet commonplace.

3.5 PVS

(Natarajan Shankar):

PVS is primarily a research platform for experimenting with different modes of inference, the interaction between formal language and inference, as well as issues of trust and automation. Trust in PVS is not merely a matter of ensuring that the inference steps have been correctly implemented. PVS provides an expressive language for capturing mathematical concepts directly and precisely. The automation available in the form of typechecking and theorem proving is directed at making it easy to construct, debug, verify, and maintain proofs.

The PVS specification language is based on a higher-order logic enhanced with predicate subtypes, structural subtypes, dependent types, parametric theories, theory interpretations, and algebraic and coalgebraic datatypes. Typechecking a PVS expression generates proof obligations. These must be proved in order for the expression to be considered well-formed. Subtyping in PVS rules out undefined function applications, division by zero, out-of-bounds array accesses and updates, and misapplied datatype accessors. Theory interpretations can be used to instantiate abstract theories as well as to exhibit models of axiomatic theories. A large fragment of the PVS language is executable. The type system and operational semantics ensure that the execution of a well-typed PVS expression in this fragment cannot trigger a runtime error (other than by exceeding resource limits).

The PVS proof engine is built from a small set of primitive inference steps. Most of the inference steps are small, but a few involve deep combinations of decision procedures and rewriting. Larger proof strategies can be defined using the primitive ones. Several external proof tools for BDD-based simplification and model checking, monadic second-order reasoning, nonlinear arithmetic, and predicate abstraction have been added to PVS. These external tools are trusted, but the code used in defining strategies need not be trusted since the proofs can be expanded down to the primitive inference steps.

3.6 Coq

(Laurent Thery):

Following Curry-Howard's isomorphism, Coq logic is based on a typed lambda-calculus: the Calculus of Inductive Construction where propositions as seen as types and proofs as programs. Its trusted computing base consists in its type-checker only: it is the piece of code that asserts that a given program has a given type, *i.e.*, that the program is a proof of the corresponding proposition. If each proved proposition in Coq comes along with its associated proof-object, this object is not a simple recording of all the primitive logical rules necessary to get to the proof. Because proof-objects are really programs, they may also include computations. This makes it possible to have explicit proof-objects even for large applications. In particular, most of the trusted extensions that have been developed within Coq use this capability of replacing proof steps by computations. This is the so-called *computational reflection*[1]. A call to an external tool is then represented in proof-objects by a call to a certified checker that verifies the certificates or the traces that are generated by the external tools.

4 Further Sources

Although we have given a few pointers into the literature, we have certainly not been exhaustive. However, here are a few other citations that may be useful.

- In [3], Rob Arthan provides a formal specification in ProofPower of the requirements for an implementation of higher order logic.
- In [22], Randy Pollack provides a careful, somewhat philosophical, discussion on the issues surrounding the checking of large proofs.
- (Mentioned in Rob Arthan's description of ProofPower.) In [25], Wiedijk discusses trust aspects surrounding the input and output of an ITP, arguing that seemingly innocuous features of the concrete syntax used by an ITP can affect trust, especially since the input and output ultimately have to be read by humans.

References

1. S.F. Allen, R. Constable, D. Howe, and W. Aitken, *The semantics of reflected proof*, Fifth annual IEEE symposium on Logic in Computer Science (Philadelphia, USA), June 1990, pp. 95–107.
2. Armand, B. Gregoire, Spiwack, and L. Thery, *Extending Coq with imperative features and its application to SAT verification*, Interactive Theorem Proving: ITP 2010 (M. Kaufmann and L. Paulson, eds.), Lecture Notes in Computer Science, vol. 6172, Springer, 2010, WWW: http://hal.archives-ouvertes.fr/docs/00/51/18/05/PDF/fastcoq.pdf, pp. 83–98.
3. Rob Arthan, *On formal specification of a proof tool*, WWW: http://www.lemma-one.com/papers/16.pdf.
4. Stefan Berghofer and Tobias Nipkow, *Proof terms for simply typed higher order logic*, Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000 (J. Harrison and M. Aagaard, eds.), Lecture Notes in Computer Science, vol. 1869, Springer-Verlag, 2000, pp. 38–52.
5. Yves Bertot and Pierre Caste'ran, *Interactive theorem proving and program development: Coq'art: The calculus of inductive constructions*, Texts in Theoretical Computer Science, An EATCS Series, Springer, 2004.
6. Sascha Böhme and Tjark Weber, *Fast LCF-style proof reconstruction for Z3*, Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings (Matt Kaufmann and Lawrence C. Paulson, eds.), Lecture Notes in Computer Science, vol. 6172, Springer, 2010, WWW: http://www.cl.cam.ac.uk/ tw333/publications/boehme10fast.html, pp. 179–194.
7. Richard Boulton, *Efficiency in a fully-expansive theorem prover*, Ph.D. thesis, University of Cambridge, May 1994, Technical Report Number 337, University of Cambridge Computer Laboratory.
8. Samuel Boutin, *Using reflection to build efficient and certified decision procedures*, Proceedings of TACS'97, LNCS, no. 1281, Springer-Verlag, 1997, pp. 515–529.
9. Ana Bove, Peter Dybjer, and Ulf Norell, *A brief overview of Agda—a functional language with dependent types*, Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009. Proceedings, Lecture Notes in Computer Science, vol. 5674, Springer, 2009, WWW: http://wiki.portal.chalmers.se/agda/uploads/Main.Othertutorials/AgdaOverview2009.pdf, pp. 73–78.
10. R. S. Boyer and J S. Moore, *Integrating decision procedures into heuristic theorem provers: A case study of linear arithmetic*, Machine Intelligence **11** (1988), 83–124, WWW: http://www.cs.utexas.edu/ moore/publications/linear.pdf.
11. R.S. Boyer and J S. Moore, *Metafunctions: Proving them correct and using them efficiently as new proof procedures*, Academic Press, 1981.
12. A. Chaieb and T. Nipkow, *Proof synthesis and reflection for linear arithmetic*, Journal of Automated Reasoning **41** (2008), no. 1, 33–59, WWW: http://www4.in.tum.de/ nipkow/pubs/jar08.pdf.
13. J. Davis, *A self-verifying theorem prover*, Ph.D. thesis, University of Texas, Austin, December 2009, WWW: http://www.cs.utexas.edu/users/jared/milawa/Documentation/dissertation.pdf.
14. Georges Gonthier, *A computer checked proof of the Four Colour Theorem*, WWW: http://research.microsoft.com/ gonthier/4colproof.pdf, 2005.
15. Michael Gordon, Matt Kaufmann, and Sandip Ray, *The right tools for the job: Correctness of cone of influence reduction proved using ACL2 and HOL4*, Journal of Automated Reasoning (2010), 1–16, 10.1007/s10817-010-9169-y.

16. John Harrison, *Metatheory and reflection in theorem proving: A survey and critique*, Technical Report CRC-053, SRI Cambridge, Millers Yard, Cambridge, UK, 1995, WWW: http://www.cl.cam.ac.uk/ jrh13/papers/reflect.html.
17. ______, *Towards self-verification of HOL Light*, Proceedings of the third International Joint Conference, IJCAR 2006 (Seattle, WA) (Ulrich Furbach and Natarajan Shankar, eds.), Lecture Notes in Computer Science, vol. 4130, Springer-Verlag, 2006, WWW: http://www.cl.cam.ac.uk/ jrh13/papers/holhol.html, pp. 177–191.
18. Doug Howe, *Computational metatheory in Nuprl*, Tech. Report 88-899, Cornell University, 1988, WWW: http://ecommons.library.cornell.edu/bitstream/1813/6739/1/88-899.pdf.
19. Joe Hurd, *Composable packages for higher order logic theories*, Proceedings of the 6th International Verification Workshop (VERIFY 2010) (M. Aderhold, S. Autexier, and H. Mantel, eds.), July 2010, WWW: http://gilith.com/research/papers.
20. Steven Obua and Sebastian Skalberg, *Importing HOL into Isabelle/HOL*, IJCAR (Ulrich Furbach and Natarajan Shankar, eds.), Lecture Notes in Computer Science, vol. 4130, Springer, 2006, pp. 298–302.
21. Randy Pollack, *On extensibility of proof checkers*, Types for Proofs and Programs (LNCS 996) (Baastad, Sweden), June 1994, WWW: http://homepages.inf.ed.ac.uk/rpollack/export/extensibility.ps.gz.
22. ______, *How to believe a machine-checked proof*, Tech. Report RS-97-18, University of Aarhus Department of Computer Science, 1997, WWW: http://www.brics.dk/RS/97/18/BRICS-RS-97-18.pdf.
23. S. Rajan, N. Shankar, and M.K. Srivas, *An integration of model-checking with automated proof checking*, Computer-Aided Verification, CAV '95 (Liege, Belgium) (Pierre Wolper, ed.), Lecture Notes in Computer Science, vol. 939, Springer-Verlag, jun 1995, WWW: http://www.csl.sri.com/papers/cav95/, pp. 84–97.
24. Tjark Weber, *Validating QBF invalidity in HOL4*, Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings (Matt Kaufmann and Lawrence C. Paulson, eds.), Lecture Notes in Computer Science, vol. 6172, Springer, 2010, pp. 466–480.
25. Freek Wiedijk, *Pollack-inconsistency*, 9th International Workshop on User Interfaces for Theorem Provers, UITP 2010, Proceedings. (C. Sacerdoti Coen and D. Aspinall, eds.), ENTCS, 2010.