`[]` or `SUCCESS` is Not Enough: Current Technology and Future Directions in Proof Presentation — Extended Abstract —

Johann Schumann, RIACS / NASA Ames
Peter Robinson, QSS / NASA Ames
M/S 269-2, Moffett Field 94035, CA, U.S.A.
email:schumann@ptolemy.arc.nasa.gov
Ph: +1-650-604-0941

1 Introduction

Automated theorem provers for first order logic are now around for several decades. Over the last few years, their deductive power to solve hard problems has increased tremendously. The annual CASC system competitions [Se97] give a clear picture of this situation. However, today's automated theorem provers are restricted "more by general usability than by raw deductive power"[1]. As a result of this, there are only very few serious applications of automated theorem provers.

There are numerous features which a theorem prover lacks for real-world applicability. An automated theorem prover (as it is currently seen) is nothing more than a fast and elaborate search procedure. In that sense, an ATP can compared to a formula-1 race car, cool and fast, but virtually unusable for shopping groceries around the corner. Many important features are missing, or are optimized for speed rather than for applicability. [Sch01] identifies important features which are needed for practical usability like detection of non-theorems, handling of modal/inductive proof tasks, control of the prover, and proof output.

In this paper, we will focus solely on the last point, the presentation of the ATP's result to the user. In the rest of this paper, we will first discuss the general importance of providing feed-back to the user, then we will describe the system ExplainIt!, a part of the deductive synthesis system Amphion/NAV. In the conclusions we will relate proof presentation to other ways of post-processing a proof found by an ATP and stress their role in the future of automated deduction.

[1] M. Kaufmann in his invited talk during CADE 15 [Kau98].

2 Output of an Automated Theorem Prover

For the following, let us consider refutation-based automated theorem prover, using an analytical calculus (e.g., SETHEO [Let92, GLMS94]), or a synthetic resolution-based calculus (e.g., OTTER [McC94]): given a set of clauses such a system tries to find a refutation, and, in case it exists, the prover (hopefully) derives the empty clause `[]` or constructs a closed tableau. For the developer of an ATP, the output of "`UNIT CONFLICT at` t `seconds`", or "`SUCCESS`" together with the run-time is of major importance.

For some applications, such an output can be sufficient (see, e.g., the system NORA/HAMMR [FSS98] for software reuse); for most applications, however, additional feed-back has to be given back to the application system or the user[2]. In general we can distinguish between the following levels of detail (for more details see [Sch01]):

1. *system- and proof-related information* like the number of inference steps in the proof, or the clauses participating in the proof provides very basic feed-back. Nevertheless, such information can be very helpful, e.g., for minimizing domain theories or for knowledge-based applications.

2. *answer substitutions* denote substitutions of variables in the query of a logic program and often carry the "calculated" result. Also for pure theorem proving applications, this kind of feed-back can be very important. For example, in deduction-based synthesis as performed by Amphion, the synthesized program is returned as a variable substitution of the proof obligation (see below).

3. the entire information contained in a successful refutation is present in a *machine-oriented proof.* Usually based upon an internal representation of the proof states such a proof can readily produced by most automated theorem provers. However, such a proof usually cannot be read, even by an expert, because the extreme level of detail obfuscates clear layout and presentation. Machine-oriented proofs, on the other hand are extremely important, because they provide all the necessary information such that the proof can be checked for soundness by a proof checker.

4. a *machine-oriented proof on the source level* has pretty much the same properties as (3), but all terms and formulas are presented on the source (or application) level. Here, all transformations of the original formula (e.g., translation from a different logic, conversion into clausal form, Skolemization), have to be "reversed", usually an extremely difficult task.

5. If the chain of reasoning has to be presented to the user, a *human-readable proof* has to be generated out of the machine's proof. This task is inher-

[2]In this paper, we only discuss feed-back in case, an actual proof could be found. Feed-back in the case where the prover does not terminate, because the formula is not unsatisfiable or too difficult, is a much harder problem with no general solution. For some initial techniques see [Sch01].

ently difficult, because issues of layout, transformation of proofs into a natural deduction style proof, hiding of excessive detail, among others, need to be addressed properly. The ILF system [DW95] (and the stand-alone subsystem ILF-SETHEO [WS97]) translate machine proofs (e.g., found by SETHEO or OTTER) into a nicely formatted LaTeX-document. Other approaches have been developed for Ωmega [Hor99].

The next section describes a deduction-based synthesis system which does not only produce executable code via answer substitution (issue 2), but also generates an *explanation* of the generated code. This explanation is directly extracted from the machine proof. However, it is presented in a human-readable, domain-oriented way. Thus the following description relates to points (4) and (5) from our list.

3 An Example: Amphion/NAV

3.1 The Domain

All vehicles, not bound to road or rail need means to determine their position and attitude. A variety of different techniques have been developed, ranging from simple devices to measure angles (e.g., compass) to complex systems like radio-navigation or the satellite-based global positioning system (GPS). In order to obtain an accurate and reliable position estimate, usually multiple measurements obtained from various sensors are combined by a numerical filter, called Kalman Filter [BH97].

Recent incidents (e.g., Mars Climate Orbiter (MCO) or Mars Polar Lander (MPL)) indicate that problems in generating reliable software for state estimation have not been addressed adequately. Such problems typically arise when certain assumptions (e.g., on measurement noise or sensor failure, see MPL) are not obeyed or the combination of sensors (e.g., same physical units (MCO) or coordinate systems) with the filter is not performed carefully.

Because for this safety-critical domain a variety of architectures exists which require experimentation and prototyping, this domain has been chosen for our *deductive synthesis* system AMPHION/NAV.

3.2 System Architecture

In AMPHION/NAV, the user gives a graphical specification of all sensors and their characteristics using abstract geometrical representations of the objects under consideration (e.g., angle-between-two-points). This specification is then converted into a first-order formula. By traditional deductive synthesis [MW92], the construction of the program corresponds to finding a proof that an object exists which fulfills the specified properties, namely $\mathcal{D} \vdash \forall I\ \exists O\ :\ \textsc{spec}(I, O)$ where SPEC is the specification, $\mathcal{D}$ the domain theory, and I, O are inputs and outputs, respectively.

For the proof search, the resolution-based theorem prover SNARK [SWL+94] is used. Once a proof has been found, the variable substitution for O comprises an applicative term representing the synthesized program. This term is subsequently converted into C++ code which then can be linked into Octave, a Matlab-style numerical environment.

3.3 The ExplainIt! Documentation Generator

Certification procedures for safety-critical applications (e.g., in aircraft or spacecraft) often mandate manual code inspection. This inspection requires that the code is readable and well documented. Even for programs not subject to certification, understandability is a strong requirement as manual modifications are often necessary, e.g., for performance tuning or system integration. Automatically generated programs are often hard to read and understand. In order to overcome this problem, one central component of AMPHION/NAV is ExplainIt! which generates explanations along with the synthesized program. By providing various kinds of documentation, all parts of the synthesized program is documented properly. Furthermore, traceability from the generated program back to the model specification is guaranteed. Both, the synthesized program and the explanations are generated by SNARK: the program is based upon the answer substitution, the explanations are generated from information contained in the proof log.

In AMPHION/NAV, the domain theory is given as a set of first-order axioms, most of them are equations. These equations relate the various objects on different abstraction levels. Each of the axiom is augmented by an *explanation template*. The explanation template consists of one or more pieces of English text, describing the (domain) background behind the corresponding part of the axiom. Templates can contain references to specific term positions in the axiom which are then instantiated or are cause for a (recursive) sub-explanation to be generated. All term-positions are given in the usual way as a path through the term tree.

When SNARK has found a proof, it generates a proof log. This log lists all proof steps together with the names of the used inference and rewriting rules. However, no information is provided on where in the term a rewriting rule has been applied. Therefore, ExplainIt! contains code to reconstruct required parts of the proof [vBRLP98]. Once the entire proof is re-generated, the explanation templates are traversed and its references instantiated. Then the entire explanation (basically an explanation for anything in the synthesized code) is written as an XML document. Figure 1 shows a small portion of the XML document. From this we generate

- a browsable version of the synthesized C++ code. Hotlinks from each part of the code (e.g., variable name, function call) point to corresponding parts in the explanation document.
- a hyper-linked explanation document. This document contains English text which explains the details of synthesized code and provides links

back to the specification.

- a printable (PDF) version of the explanations in form of a standardized design document.

All documents are generated out of the XML document, using XLST for re-arranging the information and for layout. The explanation text in these documents has been assembled from the axiom templates. It thus can be seen as a description of the program design "from first principle". Although the language output is still very rough and definitely needs linguistic post-processing, this kind of explanation contains information in a form as it would be provided by a human domain expert.

In the extended version of this paper we will have a more detailed description of Amphion/NAV and the explanation module.

4 Conclusions

As the explanation module of Amphion/NAV demonstrates, valuable information can be extracted from proofs found by an automated theorem prover. An important prerequisite here is that the information is presented in a format and a way, the domain engineer is used to. In our case, the entire explanation fully hides away the underlying logic and reasoning system used to synthesize the program. The proof, containing hundreds of inference steps is converted in such a way that it relates the input specification with the final product (C++ code).

This technique opens up an entirely new level of traceability between specification and source code. In the design and implementation of safety-critical code (according to the standard document DO-178B), traceability between the different artifacts is of central importance for code reviews and certification. However, this is only one aspect. Current practice of certification of safety-critical code very much requires following rigid processes and relies on standardized documents. Any change in the software requires an update of many documents. Here, the automatic generation of standardized documents from proofs (for synthesis as in Amphion/NAV, but also from verification) can substantially facilitate and accelerate the design process, because consistency between the software artifact and the documentation is guaranteed.

In the future one could even imagine that parts of the lengthy manual certification process is augmented by automatic checks. Proof-carrying code (e.g., [NL98b, NL98a]) might be a small step towards this direction. In all cases, these techniques require that the automated theorem prover is capable of efficiently producing a machine oriented proof. Also, work needs to be done to relate this proof back to the application domain in a way transparent with respect to all formula transformations. These examples make it obvious that future applications require techniques of result- and feed-back presentation which go far beyond traditional, mathematically-oriented proof presentation.

Acknowledgements. Without the Amphion/NAV team (Guillaume Brat,

Mike Lowry, John Penix, Tom Pressburger, Phil Oh, Mahadevan Subramaniam, Jeffrey van Baalen, Jonathan Whittle) this paper would be a hollow position statement.

References

[BH97] Robert Brown and Patrick Hwang. *Introduction to Random Signals and Applied Kalman Filtering.* John Wiley & Sons, 3rd edition, 1997.

[DW95] B.I. Dahn and A. Wolf. Natural Language Representation and Combination of Automatically Generated Proofs. In *Proc. First International Workshop Frontiers of Combining Systems , FroCoS'96, Muenchen, Germany.* (to appear), 1995.

[FSS98] Bernd Fischer, Johann M. Ph. Schumann, and Gregor Snelting. Deduction-based software component retrieval. In *Automated Deduction - A Basis for Applications.* Kluwer, 1998. To Appear.

[GLMS94] Chr. Goller, R. Letz, K. Mayr, and J. Schumann. SETHEO V3.2: Recent Developments (System Abstract) . In *Proc. CADE 12*, pages 778–782, June 1994.

[Hor99] H. Horacek. Presenting proofs in a human-oriented way. In H. Ganzinger, editor, *Automated Deduction — CADE-16*, volume 1632 of *LNAI*, pages 142–156. Springer Verlag, 1999.

[Kau98] M. Kaufmann. Acl 2 support for verification projects — invited talk —. pages 220–238, 1998.

[Let92] Letz, R. et al. SETHEO: A High-Performance Theorem Prover. *JAR*, 8(2):183–212, 1992.

[McC94] William W. McCune. OTTER 3.0 reference manual and guide. Technical report ANL-94/6, Argonne National Laboratory, Argonne, IL, USA, 1994.

[MW92] Zohar Manna and Richard Waldinger. Fundamentals of deductive program synthesis. *IEEE Transactions on Software Engineering*, 18(8):674–704, August 1992.

[NL98a] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Proceedings of the 1998 ACM SIGPLAN Conference on Prgramming Language Design and Implementation (PLDI)*, pages 333–344, 1998.

[NL98b] G. C. Necula and P. Lee. Safe, untrusted agents using proof-carrying code. In G. Vigna, editor, *Safe, Untrusted Agents using Proof-Carrying Code*, volume 1419 of *Lecture Notes in Computer Science*, pages 61–91. Springer-Verlag, Berlin Germany, 1998.

[Sch01] Johann Schumann. *Automated Theorem Proving in Software Engineering*. Springer, 2001. in print.

[Se97] G. Sutcliffe and C. Suttner (eds.). The cade-13 automated theorem proving system competition. *Journal of Automated Reasoning*, 18(2), 1997.

[SWL+94] Mark Stickel, Richard Waldinger, Michael Lowry, Thomas Pressburger, and Ian Underwood. Deductive composition of astronomical software from subroutine libraries. In Alan Bundy, editor, *Proc. 12th International Conference Automated Deduction*, volume 814 of *Lecture Notes in Artifical Intelligence*, pages 341–355. Springer, June/July 1994.

[vBRLP98] J. van Baalen, P. Robinson, M. Lowry, and T. Pressburger. Explaining synthesized software. In *Thirteenth International Conference on Automated Software Engineering*, pages 240–248. IEEE Computer Society Press, 1998.

[WS97] A. Wolf and J. Schumann. ILF-SETHEO: Processing Model Elimination Proofs for Natural Language Output. In *Conference on Automated Deduction (CADE) 14*, 1997.

```xml
<argument type="function" path="(12 1 2 4 3 ANCILLARY)">
<function path="(12 1 2 4 3)"><head>MK-MX</head>
...
<reason axiom="CALC-MEASUREMENT-ROWS-NON-NIL">EACH
    SENSOR MEASUREMENT CORRESPONDS TO A ROW IN THE MEASUREMENT MATRIX
</reason>
<argument type="constant" path="(12 1 2 4 3 1)"><value>2</value></argument>
<function path="(12 1 2 4 3 3 1 1)"><head>X-COORD</head>
<reason axiom="MEASUREMENT-ROW-NON-NIL">EACH ENTRY IN EACH ROW IN THE
    MEASUREMENT MATRIX IS FORMED FROM LINEARIZING AN EQUATION REPRESENTING
    THE MEASUREMENT AROUND A NOMINAL TRAJECTORY
</reason>
<reason axiom="MEASUREMENT-ROW-NON-NIL">I.E. FOR MEASUREMENT Z_I ,
    STATE VARIABLE X_J
</reason>
```

Figure 1: Excerpts from the XML explanation document

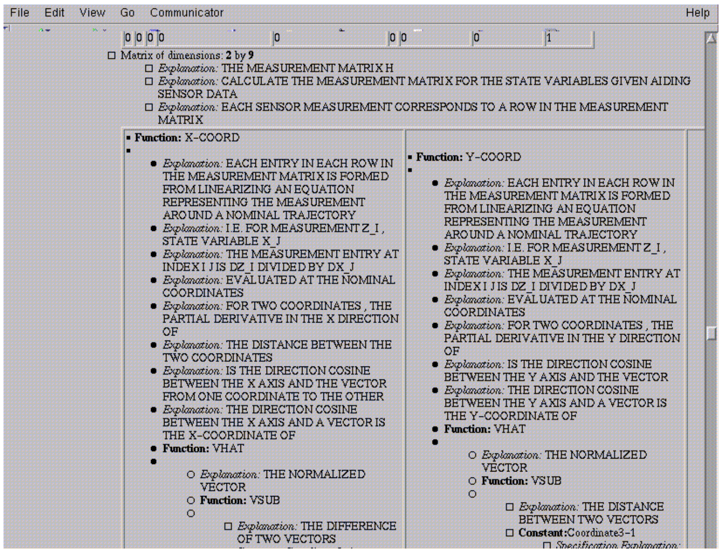

Figure 2: Screen dump of a part of the explanation document