How can we improve theorem provers for tool chains?

Martin Riener

The University of Manchester
martin@derivation.org

Abstract

Automated first-order theorem provers have matured as standalone tools to the point that they can be used within a larger infrastructure like Isabelle's Sledgehammer. Nevertheless, there is a significant difference to the spread of SAT solvers, that occur in simple applications like configuration management but are reliably used in tight loops of larger tool chains, not the least in SMT Solvers or instantiation / AVATAR based ATPs. We cannot expect a similar level of integration due to the higher expressiveness of general purpose theorem proving. Nonetheless, here we will identify some aspects that could improve the acceptance in industry.

Automated theorem provers have seen use as back-ends in larger software packages (various hammers, TLAPS) but are rarely used within a tool chain. For decidable theories, SMT solvers certainly have better properties but in this context, we focus mostly on their use as general purpose theorem provers (in other words, any SMT-LIB logic that includes uninterpreted functions and quantifier support). The ability to produce models still distinguishes SMT solvers but the support of full first order logic confronts them with similar problems as other ATPs.

Many of these problems are of a technical nature: the integration of a theorem prover into an industrial project brings several external requirements into play that are easily dealt with in interactive modes. The software might need to run in a certain operating system or avoid optional components under the GPL license. The availability of certain prover features also often depends on such optional components. For example, an arbitrary precision library might only provide integers where its differently licensed alternative also provides unbounded real numbers. In interactive use, we can often restate the problem to avoid real numbers altogether. A human can also investigate the reasons for time outs easier.

We summarize these problems under the titles common availability, standardized interfaces, and reliability. The analysis is centred around CVC4, E Prover, iProver, SPASS, Vampire, veriT, and Z3.

Availability

Most commonly, provers target Linux as the main operating system and are available in source form to be compiled manually. Static Linux binaries are also often available for download at the project homepage, with the exception of Vampire. SPASS and E are packaged in single Linux distributions (Debian/Ubuntu and Fedora respectively) whereas CVC4 and Z3 are widely available. An exception are Alt-Ergo, Zipperposition and Beagle which are usually installed via Opam and sbt, the respective source based package systems for OCaml and Scala.

Apart from Z3, CVC4 and veriT, which offer native binaries, the support for Windows is usually provided via the Cygwin compatibility layer. MacOS is usually supported via the source based package manager Homebrew. Due to Apple's strong discouragement of static linking, this is normally the only way of distribution.

From an industrial perspective, external factors often determine the operating system or way of distribution required. When multiple provers are required, the most common denominator is often only the compilation from source. The introduction of automated builds and testing environments for multiple operating systems is a significant task that is hard to publish on. Although rare, there are research engineer grants that could be used to hire an expert that does this maintenance. Since most provers are under an open source license, one time engineering tasks like the adaptation to a yet unsupported operating system could be announced as a Google Code project.

Interfacing

The most common way of communication with a theorem prover is via plain text. This has the advantages that the input language remains stable and the prover can be easily exchanged for an alternative. The two main input formats, SMT-LIB and TPTP, have been converging with regard to the supported logics and theories: for example, they both define integer and real arithmetic and higher-order logic. There are differences though: only SMT-LIB defines special theories like bit-vectors, data-types or arrays, even though some provers accept them with a custom syntax whereas only TPTP supports polymorphic types and modal operators. The choice of the language therefore restricts which features are efficiently supported.

A further restriction is the fact that no prover supports all features of either SMT-LIB or TPTP. This ties the tool chain even tighter to a particular prover, making it hard to use different encodings of the same problem. On the other hand, the close integration can not be used to share data-structures or avoid repeated parsing of input.

Another distinction regards the output formats. SMT-LIB does not specify a proof format but it has a definition of models in terms of a Herbrand universe extended with abstract values. These abstract values are solely defined by the solver and again tie the tool chain tightly to a particular prover. TPTP's proof format TSTP on the other hand specifies the proof DAG with the single semantic restriction that a conclusion logically follows from its premises. More detailed proofs can be constructed but require additional reasoning. If the proof replay is not tailored to a prover there is also a reasonable chance that it fails. TSTP only specifies a concrete format for finite models. In both cases, an industrial user needs to rely on prover specific behaviour that is often poorly documented.

Some of these issues can be mitigated with an intermediate layer such as pySMT. As an alternative, one can use an API, should it exist and when it has the correct language bindings. The API might change more frequently because it is tied more closely to the implementation.

In each case, only the ATP community can improve the situation. The efforts of harmonizing the SMT-LIB format with TIP seem to have been successful, making the benchmarks collections of both formats available to the other community. Due to the larger syntactic differences, this is not easy to achieve between TPTP and SMT-LIB but it is certainly possible continue integrating the features of the other side. The veriT SMT solver provides an excellent proof format that could be integrated into SMT-LIB. A stand-alone tool that converts between the formats would also be of great help. This could be a stepping stone of making the whole benchmark libraries of both formats available to the other. Currently, there is an overlap due to manual conversion efforts (for instance, the AUFNIRA/nasa is an adaption of the original statements using the array sort instead of an axiomatization). An full problem library that can state the problem in the logic and language required would make such a manual conversion effort unnecessary.

Reliability

The most dreaded behaviour of a theorem prover which has not been mentioned so far is when it reports "unknown", either due to a timeout or when it rejects the problem as outside of its capabilities[1].

This can be due to an unfavourable encoding like expressing equality as a reflexive, symmetric and transitive relation. Other cases are not that obvious: arrays with Boolean values can replace a bit-vector but it depends on the decision procedures that are actually implemented which determine if the encoding is suitable to a problem. Decisions of the latter kind could be integrated into a framework like pySMT (also to compensate for missing theories) but in general this can only be solved by the creator of the tool chain.

Another way to obtain "unknown" is by enabling incomplete proof search strategies, be it particular instantiation techniques or axiom selection. In general, the number of options for theorem provers is immense and their interaction is hard to predict. Unfortunately, this situation is hard to improve. ATP developers can evaluate the effectiveness of a large number of combinations to find those that work well or eliminate some those that don't have any effect. How well this works depends strongly on how representative the benchmarks are for the application. Specialized tool chain developers might be able tune the options themselves but this can not expected from the average user.

A third possibility for a timeout is lack of knowledge about the (possible) model. A superposition based prover will not saturate the clause set generated by $\forall x : Int, y : Int. x < y$ and continue until it times out. Still, some learned similarity measures could select sufficiently distinct generated clauses that could be of interest for choosing a better suited set of options. A similar phenomenon occurs when SMT solvers run in a refinement loop. The clause above almost immediately produces a model $x = 0, y = 1$. A refined model would exclude these model values, obtaining a new model $x = -1, y = 2$ and so on. A human quickly recognizes that the problem has both infinitely many models and counter-models. Part of this information is also generated when the decision procedure – here probably the Simplex algorithm – is run and we should know more about the model theory of that decision procedure. How this knowledge can be preserved in the face of theory combination is not obvious. When we consider the clause $f(x) < f(y)$, whatever model we pick for f overshadows any monotonicity intuitions we had for the simpler case. Nevertheless, it could be beneficial to extract more information from a stopped proof search and feed this back into the refinement loop.

Summary

We have focused on topics that could be considered as obstacles for industrial integration: possible runtime environment restrictions, the need to ship with predetermined parameter schedules or the general hardships of undecidable problems. Nevertheless, it is worth highlighting the impressive advancements that have already been made in that regard. Modern theorem provers come with an extensive manual that includes their inference rules. The level of detail in proofs has become significantly more detailed. Finally, there have already been successful integrations of general first-order provers in widely used software, ranging from backends for interactive proof assistants to software synthesis and test case generation. However we hope that future development can close the gap between the research and its application in industry.

[1]For example, both CVC4 and Z3 report unknown when trying to find a model of $\exists a : Array(Int, Int), b : Array(Int, Int). a \neq b$ but they can both construct a model for the inequality of constants $c \neq d$, which is how existential quantification is defined.