Logos Machines

A Review of Automatic Theorem Provers and Their Applications

Devin Gonier
Department of Computer Science
Lewis University
Romeoville, IL
DevinCGonier@lewisu.edu

***Abstract*—This article introduces automatic theorem provers and some of their design features. Specifically, articles which discuss verification functions based in logically expressive syntax and translation between non-classical logics are reviewed. These discussions are made concrete with the case study of verifying a system for gamifying internal business communications.**

Keywords—Model Checking, Automatic Theorem Provers, Non-Classical Logic, Paraconsistent Logic, Quantum Logic, Gamification, Internal Business Communication Gamification

INTRODUCTION

The Greek word *logos* could be defined as 'reason', 'word', or 'plan', but probably no translation quite captures the subtlety of its far-reaching impact for philosophy and science over the centuries. Logos represents the pursuit of deeper *a priori* structures and foundations for reasoning, and consequentially appears in the suffix (-ology) of the name for many disciplines. The discipline of logic is perhaps its most direct descendent, and its rigorous development, from Aristotle to Frege to contemporary non-classical logicians, has had tectonic and indiscriminate impacts on the development of most disciplines. Some of the history of logic could be seen as an attempt to prove what most already find to be intuitive, such as Frege's attempt to provide a logical foundation for mathematical concepts like addition. Other discoveries are less than intuitive, such as Godel's incompleteness theorem, recent discoveries in quantum physics that have engendered the field of quantum logic, or novel approaches to old problems such as paraconsistent interpretations of the Liar's paradox.

Much of the (especially early) part of computer science dealt with ways to mechanize the formal principles of logic. Today, this pursuit continues. Automatic Theorem Provers are tools used by computer scientists primarily to verify software using logical analysis. They have grown in complexity over the years, and now can handle multiple logics when performing a single analysis. Such developments are still young, and there remains great opportunity for incorporating novel developments in logic. This will make programs not only more correct, but also smarter. In the first section, two articles are reviewed that discuss Automatic Theorem Provers. This is followed by a look at a concrete example of a product used in business applications. Finally, the paper concludes with some comments about the future significance of developing logical approaches to verification.

I. AUTOMATIC THEOREM PROVERS: A LITERATURE REVIEW

A. Review: Automating Theorem Proving with SMT

Satisfiability is an approach to verification of software that determines whether or not certain functions satisfy propositions which are formally specified in logical notation. Many problems can be satisfied with Boolean solvers (SAT), but some require more expressive logical syntax such as various arithmetic operations, arrays, datatypes, and equality expressions [2]. In such cases where notation needs to be more expressive, SMT (Satisfiability-Modulo-Theory) engines are an alternative to Boolean restricted methods. These engines verify according to a "background theory (e.g. the theory of equality, of integer numbers, of real numbers, of arrays, of lists, and so on) [1]." In "Automating Theorem Proving with SMT" Leino discusses SMT engines and some specific functions which they perform to help in automating the verification process of software and/or mathematical proofs. Software verifiers that use SMT engines are advantageous because they can verify software by expressing "necessary proof ingredients in program centric declarations like preconditions or loop invariants" [3]. Leino's discussion largely centers around a specific SMT solver called Dafny and some of its more unique capabilities and qualities. Leino focuses on a few specific functions of Dafney in streamlined automatic verification that involve expressive logical notations amenable to SMT engine approaches. Furthermore, Leino emphasizes the efficiency of such operations, and the potential advantages of using these verification methods as a background application during development [3].

The first attribute of Dafney discussed is its handling of *co-inductive* datatypes. This type of data can be infinite, and thus requires unique specifications so as to not create liveness problems for software. For example, a function which recursively squares itself ($x=x^2$) without a termination parameter would approach infinity. With Dafney such a function is acceptable because one can specify the function as being a co-datatype stream, and then using parameters *heads* and *tails*, specify the starting and ending point of the function when it is called. Such analysis needs to factor in property restrictions of the numbers parameterized. For example, in a codatatype function that recursively decreases, if there are restrictions on the return being only natural numbers, there

will be errors that result, as eventually those results could become negative [3].

Another attribute of Dafney is the ability to construct inductive proofs by specifying pre- and post- conditions. Preconditions are specified using the *requires* command and postconditions are specified using the *ensures* command. The author uses a *lemma* comparison between two tail functions (*tail* and *tail_alt*) as an example to demonstrate equivalence given pre- and post- conditions. The *lemma* is then instantiated in a systematic way using the *calc* function, which is readable to users afterwards. Recursive and inductive proofs can also be applied to co-datatypes using other functions, and for some such functions their co-recursive nature does not require the specification of a terminating command [3].

Finally, the author discusses how Dafney evaluates filter functions. These functions can filter a stream according to specific predicate specifications. For example, one can filter a stream to ensure that values are only increasing, or only certain types of numbers are returned. These processes involve commands like *Ord*, *AlwaysAnother*, and *StepsToNext*, to specify pre- and post- conditions appropriately. By combining such capabilities with inductive proof techniques such as the *lemma* method, it is possible to evaluate sequences, arrays, or other datatypes to determine whether or not certain properties hold even if those sequences are potentially infinite [3].

B. *Review: Problem Oriented Applications of Automated Theorem Proving*

While Leino's evaluation of Dafney as an SMT automatic theorem prover was mostly aimed at evaluating computer programs via more expressive logics, the automatic theorem provers (ATP) discussed by Bibel et al. are more focused on dealing with mathematical logic proofs, especially ones involving non-classical logics. However, other applications do exist for the type of ATPs discussed in Bibel et al. such as automatically synthesizing "computer programs from a given specification" and automating control "of the behavior of intelligent agents within a given environment" [1]. The primary focus of the article by Bibel et al. is on the simultaneous use of non-classical logics, and their respective transformations. The three logics discussed are intuitionistic logic, which the author most associates with program synthesis, modal logics, which can be useful in the development of time and belief concepts, and classical logic, which is used for mathematics and other types of computations. The relationship between these different logics is complex and difficult to distinguish, which is why translations and relations between them is useful [1].

In the ATPs discussed in this article, there are three main stages of computation. The first is input transformation, which involves leveraging logic morphisms or normal form transformation techniques to translate different logics into a unified language which the machine can evaluate. The second stage is the inference machine itself, in which the actual "exploration of the search space" occurs [1]. The third stage is the presentation of proofs, in which the various translations are recomposed and placed into a format readable by logicians creating the proofs. Perhaps the most important components of this process are the initial transformations themselves, as relating logics is extremely complicated. If successfully automated, it may not only create more expressive results but also certain efficiencies for various applications. Relating intuitionistic, modal, and classical logics can be very useful in modeling complex processes like knowledge extension, where one considers what could be known in relation to what is currently known [1].

ATP technologies have impressive potential, as they can relieve some of the rigorous burdens of proof construction experienced by mathematical logicians and consequentially may also serve a creative role in the development of theories that will impact research in a variety of fields. One wonders though if this technology is still ultimately restricted to certain kinds of logics. For example, it may be much easier to translate many intuitionistic claims into classical claims because the differences between the two logics is perhaps less severe than classical logic and other logics like quantum logic or paraconsistent logic. Arguably, it is the latter though that is more important for the development of computer science in the near future. For example, the growing field of quantum computing has created a demand for quantum algorithms that can take advantage of quantum effects. This is complicated by the fact that traditional computing techniques depend upon classical logical reasoning, which diverge significantly from quantum logic. Superposition and entanglement are phenomena that are being leveraged to make exponential gains in computing relative to their classical counterparts, yet are deeply counter-intuitive and often contradictory with classical approaches. If techniques for relating or translating other non-classical logics like quantum logic and paraconsistent logic develop in addition to the ones discussed in this article, then there is perhaps even greater potential for ATP integration in the future of algorithmic processes [1].

II. ATP Verification of Gamified Internal Communication Strategies

It is sometimes difficult to visualize the applications of these technologies in industry. In fact, for some non-classical logics, pervasive industrial applications may be far off. Nonetheless, it is important to realize that even basic programs and goals can benefit from formal verification techniques that depend upon concepts discussed by these researchers. To make these abstract concepts more concrete, the remaining paper will focus on the usefulness of theorem provers on a specific application. The application chosen for this paper is the development of private communication forums in large business enterprises. This technology will be case studied with Automobile Association of America's (AAA) branches within the Auto Club Enterprises (ACE) club.

Since previous similar attempts at growing communications within ACE have been less than sustainable, our hope is to create a program that gains momentum early on and remains sustainable. For that reason, this technology will gamify communications by assigning points for certain kinds of idea-sharing. Those points will then be used to win contests that result in recognition and monetary rewards. There are

however, many ways this program could fail without the correct initial specifications. Many managers who would participate are less than tech-savvy, and are unlikely to have already participated actively in other forms of social media. Additionally, since there are financial awards and possible career advancements involved, it's important to prevent behaviors which may be seen afterwards as cheating or gaming the system. For these two reasons, there are three categories of specifications that must be taken into account when designing such a forum: Game Rules, Participation Rules, and Search Rules. Here I will highlight a couple rules in each category and how these could be logically specified for verification purposes.

In the first category (Game Rules), a point is given for new topical posts and when credit is given by other users for that post. Credit can be given by a simple upvote which each user has an infinite supply of or by the assignment of an appreciation badge such as "Big Idea!" or "I'm Going to Do it in My Branch!", for which each user has a finite supply. Whether or not a post is topical is determined by the use of keywords specified in the original topic question. For example, if the topic of the forum question is, "How do you coach your employees on Trip-Tiks?", posts that don't include words like 'Trip Tik,' 'Employee,' or 'Coach' will get flagged to be evaluated by a moderator. If flagged, the post is not necessarily excluded outright, but will be evaluated by moderators to determine its topicality. This is to help minimize spamming or irrelevant posting. Each user is limited in the number of badges they can award. For example, a user may begin the game with 10 "Big Idea" badges and 15 "I'm Going to Do it in My Branch!" badges, for which they must use sparingly when evaluating other posters. A user cannot give more than one of any badge to a single post. So, one logical property to be evaluated would be, "No user may give more than one of a single type of badge for a single post." This general rule could be formalized using the following expressions:

- All badge points of badge-type *x* for a post *p* from user *z* are unique.
- All badge point assignments of badge-type *x* by user *z* for any post *p* may be only given once.
- After assignment by user *z*, the number of badge points (other than upvotes) belonging to user *z* of badge-type *x* will be reduced by one.

Here there are three key variables to be evaluated: the post itself, the badge or upvote assignment, and the user doing the assigning and receiving the assigned credit. These statements can all be expressed in second order logic using basic set theory and classical logic notation.

In the second category (Participation Rules), the goal is to minimize threat of improper use of identifiable information. For example, no full names may be used, no users without login access may post or read posts, and no posts can be made anonymously. Fortunately, such specifications are fairly standard and turnkey solutions of this type can be found without too much difficulty. The last category (Search Rules) are designed to optimize the ability for users to find information posted by other users. This will involve tagging restrictions. Any post made in response to a forum topic will automatically be tagged according to that topic, and other tags will be restricted to a list that prevents redundancies. The desired effect is that users who wish to find ideas may easily search by forum topic, tag, or badge reward. The variables in this category: are tags (their quantity and redundancy), topics, and how badges and upvotes prioritize search results.

Each of these specifications could be initially setup in a model checker that runs on an automatic theorem prover. The model checker could then evaluate the effectiveness of software according to these rules by posting in various ways, assigning upvotes and badges in various ways, and doing so from various user perspectives. Finally, traces of behaviors can be matched up to final scores to evaluate whether or not behaviors appropriately correspond to scoring intentions, searches could be done to evaluate the ease of finding relevant information, and initial specifications are not violated by the code. Using ATP's as part of development or verification afterwards will prevent cheating, and will ensure the usefulness of the forum as a repository for ideas.

CONCLUSION

The verification of software used to automatically manage forums specifically designed to gamify internal business communication is but a small example of the potential use of automatic theorem provers. The act of automating software synthesis according specifications and verification of new software will continue to be important components in the development of autonomous intelligent systems, which are able to modify their own architecture. Important historical computer scientists such as John Mccarthy have recognized the importance of logic in designing intelligent systems and creating more efficient and complex programs for solving sophisticated problems. The ability to deal with complex logical syntax and to relate different logics together will help unlock this potential in the years to come. Automatic theorem provers are not only useful in verifying software, but could also be the key to more powerful revolutionary designs that exponentially expand the potential for complex computation.

REFERENCES

[1] Bibel, W., D. Korn, C. Kreitz, and S. Schmitt. "Problem-Oriented Applications of Automated Theorem Proving." Fachgebeit Intellektik (n.d.): n. pag. Web.

[2] Editors of Encyclopædia Britannica. "Logos." Encyclopædia Britannica. Encyclopædia Britannica, Inc., n.d. Web. 03 May 2017. <https://www.britannica.com/topic/logos>.

[3] Leino, K. Rustan M. "Automatic Theorem Proving with SMT." (n.d.): n. pag. Microsoft Research, 19 May 2013. Web.

[4] Moura, Leonardo De, Bruno Dutertre, and Natarajan Shankar. "A Tutorial on Satisfiability Modulo Theories." Computer Aided Verification Lecture Notes in Computer Science (2007): 20-36. Web.

[5] Tinelli, Cesare. "Foundations of Satisfiability Modulo Theories." Logic, Language, Information and Computation Lecture Notes in Computer Science (2010): 58. Web.

[6] Thomason, Richmond. "Logic and Artificial Intelligence." Stanford Encyclopedia of Philosophy. Stanford University, 27 Aug. 2003. Web. 03 May 2017. <https://plato.stanford.edu/entries/logic-ai>