Applications of Automated Theorem Proving

A Review of Literatures on Automated Theorem Proving

John Hornik
Assignment 5
Research in Computer Science
Lewis University
Romeoville, IL, USA
johnphornik@lewisu.edu

***Abstract*—This paper reviews concepts presented in *Automating Theorem Proving with SMT* [1] and *Problem-Oriented Applications of Automated Theorem Proving* [2]. It begins by a broad summary of [1] followed by a broad summary of [2]. It briefly covers many of the concepts presented in these two literatures. The paper concludes with an application of software in a real-life control system and gives a brief overview of how automated theorem proving would benefit such a system.**

***Keywords*—Automated Theorem Testing; Proof; SMT**

I. Introduction

Automated Theorem Proving is a method for simulating mathematical reasoning using computer programs. In this paper, Automated Theorem Proving (ATP) is presented through two different literatures on the topic, [1] and [2]. [1] focuses on presenting potential proof assistants that are based entirely on satisfiability-modulo-theories (or SMT) solving. The paper centers around Dafny as the main tool implemented. [2] is a paper that focuses on the development of an ATP system which can be applied to three different areas: mathematics, programming, and planning. It also summarizes research in logic and the structure of the proof process of ATP systems.

II. Summary of Automated Theorem Proving with SMT

A. Introduction

The concepts which are presented in [1] focus on utilizing satisfiability-modulo-theories (SMT) in automated theorem proving. SMTs are collection of decision procedures for certain theories. Because they allow for high degrees of automation, there has been an increase in proof assistants implementing SMT solvers as subroutines. The author of the paper presents ideas of proof assistants based entirely on SMT solving. This is done with programming language and auto-active program verifier Dafny. The paper begins with a definition of a type and a function which will be used throughout the rest of the paper. It then goes into a lemma and then proves it by induction, including a proof calculation. The following section presents co-inductive declarations and proofs. After this, the inductive and co-inductive features are combined into a filter function. Finally, the paper concludes with a proof of a theorem regarding filters. This is accomplished by simultaneously applying induction and co-induction.

B. A Type and a Function

The ideas are explained using a heads-or-tails type example. A type `Stream` is defined whose values are infinite lists. Stream is parameterized by the type of its elements, `T`, and has one constructor, `Cons`. The parameters, `head` and `tail`, to the constructor declare destructors. After this, a function is declared which returns a suffix of a given stream. In the authors example, `Tail(s,n)` would return `s.tail`. It is made clear that each function is checked for well-foundedness amongst the recursive calls. This is achieved by implementing a variant function evaluated on entry call. In this case, if the value of this function is smaller for that callee than the caller, well-foundedness follows [1]. On the other hand, if no function is supplied, then the program Dafny guesses one. In this example for the case of `Tail`, Dafny guesses the variant function and checks that the `Tail`'s recursion is indeed well-founded.

C. An Inductive Proof

In this section, a lemma is stated and proved by induction. A lemma is expressed as code with a pre and post condition, or a method. In this case, for values of the method parameters which satisfy the method's precondition, the method will then terminate itself in the state which satisfies the post condition. In the author's example, he attempts to prove that `Tail` and `Tail_Alt` will yield the same result, in reference to the results received from the previous section. This is achieved with the precondition stating that n is a natural number and the post condition giving the property which is sought to be proven. A method body is then supplied and the program verifier is able to run the code and verify that the control paths terminate and establish the post condition. In the actual method, the body provides two code paths: for an $n < 2$ branch, the verifier proves the post condition by unwinding `Tail` and `Tail_Alt` one or two times. An else branch utilizes a **calc** statement as well as a number of equality steps, which are verified. As a result, the verifier is able to check the equality of each step and checks that the pre-condition of the callee is met. It then checks that the variant function is decreased for a call and can then assume the post condition of the callee. It is noted that the final proof is more than Dafny actually needs but is quite comparable to hand-written proof which can be more suitable for humans.

D. Co-recursion and a Co-inductive Proof

Values of a co-datatype are considered as well as their construction and how someone can state and prove properties of these values. In Dafny, a self-call is co-recursive if it is positioned as an argument to a co-datatype constructor. These types of calls are then evaluated into code in a way where the arguments to the constructor are not evaluated until their values are used by the executing program. In order to define a co-inductive datatype, a co-predicate is necessary. These are defined by greatest fix-points as the greatest solutions of the recursive equations to which their definitions give rise [1]. Co-methods are also discussed. These are methods whose purpose is to enable co-inductive proofs [1].

E. A Filter Function

In this section, it is noted that for any stream, it is necessary for the filter function to return the stream consisting of those elements of the stream that satisfy some predicate. The filter function and all related lemmas are parameterized by the predicate [1]. The termination of recursive calls is also discussed. In order to avoid non-termination, one must restrict the input to streams that contain infinitely many elements which satisfy the predicate.

F. A Property of a Filter

The author notes that the interesting property to prove about a Filter is that it returns the subsequence of the stream that consists of exactly those elements that satisfy the predicate [1]. This subsequence can be divided up into the property that the filter returns the correct set of elements. The proof given of the lemma utilizes co-induction and induction together. Co-methods are used to validate the co-predicates. They are implemented with induction, in conjunction with meta-theorem.

III. Summary of Problem-Oriented Applications of Automated Theorem Proving

A. Introduction

[2] largely focuses on an approach to develop an ATP system which can deal with a number of different logics and applications. The basic structure of the proof process of ATP systems is reviewed. Two different approaches to dealing with semantical meta-knowledge originated from non-classical logic are also covered. The paper ends with an outlook to future work.

B. Structuring the Process of Theorem Proving

The inference machine is the core of an ATP system. The author compares it to a microprocessor of theorem proving, where a formula to be proven is preprocessed in order to transform it into a sort of machine language. In the output, the proof is then post processed in order to be user-friendly once again. Because of this, three problems are usually encountered: input transformations, inference machines, and proof presentation. The first is the interface between the application oriented and the inference machine oriented languages. The second is in regards to the machines which do the exploration of the search space which is spanned by the underlying logic. Finally, the third is the interface between the logical calculus of the inference machine as well as the given application [2].

C. Combining Classical and Non-classical Theorem Proving

In this section, two points are presented. The first one takes place at preprocessing time and formalizes the Kripke-semantical notions within the language of classical logic. Its basic idea is to deal with additional reasoning caused by non-classical semantics of the logical connectives by encoding these semantics into a classical first-order formula at pre-processing time. The second extends the classical inference machine by a co-processor which does additional reasoning and checks for each classical inference step [2]. An input formula may be used in a direct way without any translations and normal form transformations, however, when considering non-classical logics, the semantical information can also be processed during the proof procedure. This additional information would then be contained in the resulting proof, making it easier to construct a readable output. For these purposes, the authors develop a uniform framework non-normal form theorem proving. The method discussed consists of a two-step algorithm: a uniform procedure for finding the proofs as well as a uniform transformation procedure converting the proofs into sequent-style systems [2]. During the development phase, the authors unify all logics under consideration into one system of matrix and sequent style calculi. They then obtain the basic structure of the corresponding procedures using the invariant parts as uniform steps. After this, generalized invariant proof and transformation procedures are achieved in which the approach realizes a large degree of flexibility and can easily be extended to other logics [2].

In discussion of non-normal form automated theorem proving, the authors explain that the theoretical foundation for their proof is a matrix characterization of logical validity where a given formula is logically valid if each path through the formula has at least one complementary connection according to the selected logic [2]. It is achieved by introducing the syntactical concept of prefixes to encode the Kripke semantics of the logics into the proof search. An extension of the notion of complementarity is then obtained. The procedure also requires a string-unification procedure for making two prefixes identical [2].

The authors mention that non-normal form automated theorem proving is only the first step in solving problems via ATP methods. After that, is developing a uniform algorithm to transform non-normal form matrix proofs into comprehensible form. This is the basis of the discussion in Converting non-normal form matrix proofs into sequent-style systems. It is noted that this step is essential due to the fact that the efficiency of automated proof methods which are based on matrix characterizations depend on compact representations of the proofs [2].

D. Future Work

The authors of [2] conclude their paper by giving an outlook into future work. They discuss their current work towards a C-implementation of a connection-based proof procedure for intuitionistic logic. They then explain their intentions to investigate how inductive proof methods could be utilized in program synthesis methods. Finally, they discuss their future plans of researching how techniques utilized in systems such as Setheo and KoMeT can be imported into their own system.

IV. Relevant Application of Concepts

In [2], it is mentioned that ATP can be implemented in the automated control of the behavior intelligent agents within a given environment. This sort of application would require the means to automatically generate plans for their actions within the environment. Essentially, these plans would be a series of actions leading from the initial condition to a desired one. This would be ideal in the case of the crystal string saw within the Crystals Department at Siemens Medical Solutions in Hoffman Estates, IL.

Currently, the saw is controlled by an analog based system but implementing a microcontroller and using ATP methods to verify its reasoning would greatly benefit an advancement in the saws capabilities. Breaking the process down into simpler components would be the first step in analyzing this system. A broad description of the saw's operation is as follows:

- After pressing "Start" button, motor begins to spin and saw begins cutting crystal. This is state 1 to state 2.
- Saw continues cutting crystal (continues in state 2).
- Saw aborts and motor turns off on three conditions (returning to state 1): if the "Off" button is pressed, if crystal is fully cut (limit sensor is tripped), or if a timer runs out. The timer function is essential in protecting the integrity of the crystal in cases where the motor moving the saw fails and the motor turning the cutter continues. This can cause serious damage to the crystal and in many cases it would need to be discarded.

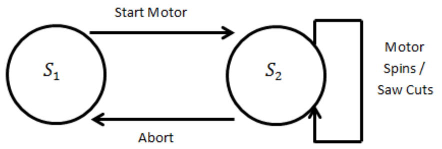

Figure 1

Figure 1 displays a simple Kripke structure of the saw. Essentially, the system can be described with predicate logic, or first-order-logic. The system deals with HIGH/LOW's throughout, however, there are other factors, such as time in the timer, that contribute to the overall system. I believe that this would be a combination of classical logic as well as modal logic, where time is involved. An example of a program for this system is as follows:

```
  digitalWrite(motor1, HIGH);//power up motor1
  digitalWrite(motor2, HIGH);//power up motor2
  sensorStatus =
digitalRead(stringSensor);//this is to read
the status of the sensor.
  if (sensorStatus==HIGH){//if sensor doesn't
read metal plate, then start counter.
          count = counter++;//this block of
code counts up by '1' every second
          Serial.println(count);
          delay(1000);
   }//close conditional statement for HIGH
sensor
  if (sensorStatus==LOW){//if sensor reads
metal plate, reset timer to zero
          counter = 0;
   }//close conditional statement for LOW
sensor
  if (counter==20){
          digitalWrite(motor1, LOW);
          digitalWrite(motor2, LOW);
          exit(0);//ends everything when
timer hits max time of 77 seconds. restart
needed.
   }//close termination condition
```

In the code, it is apparent that the system relies heavily on logic for control. When the motors power on, they continue to run until power is cut (which is beyond the control of the logic), if the limit sensor is switched, or if the timer runs out. The limit sensor (`stringSensor`) is read using the `digitalRead` command. It is then stored in `sensorStatus` and this is where the system determines the state of the sensor. With the Boolean relations of HIGH or LOW implemented into the if-statements, the program can determine which action to take. Similarly, the counter (stored in `count`) enables a timer to begin once the position of a metal plate (for the hall-effect sensor) is no longer being read. This portion of the system will automatically cut power to the saw if the counter runs out – in such circumstances, this is generally the cause of motor gear slipping and the saw cuts away at only one portion of the crystal. An automated theorem proof of this process would omit any errors that can result from a poorly tested system and could potentially save the crystal being sliced, which in turn could save the company thousands of dollars. It would also be cheaper and more efficient that checking the system by hand. In a more complicated design, possibly one implementing machine learning techniques, ATP would also be very beneficial in verifying that all of the operations are correctly implemented in the program. Because this is essentially an embedded system, a tool such as Dafny would not be sufficient enough on its own to verify it. According to [3], FreeRTOS is an adequate tool for testing quality assurance aspects of real-time and embedded systems. Because Dafny was created in order to prove program correctness, it would be able to work closely with FreeRTOS in order to prove the latter's modules.

V. Conclusion

In the summary of [1], Dafny's proof features are exemplified with inductive and co-inductive definitions and proofs as well as human-readable proofs. The examples in [1] are meant to show the degree of automation capable with a tool which implements a SMT solver. In [2], an ATP-System is developed and the overall approach is discussed. The results of research in this area are also summarized in the areas of mechanizing classical and non-classical logics. These two papers prove that automated theorem proving can be an excellent approach to verification of software systems. This is further exemplified in the example of implementing these concepts in a real-world software-based control system.

References

[1] K. Rustan, M. Leino, "Automating Theorem Proving with SMT", Microsoft Research, 2013.

[2] W. Bibel, D. Korn, C. Kreitz, and S. Schmitt, "Problem-Oriented Applications of Automated Theorem Proving", Technische Hochschule Darmstadt.

[3] M. J. Matias, "Program Verification of FreeRTOS Using Microsoft Dafny", Cleveland State University, 2014.