Project Proposal

Matthew Gregoire
Kaki Ryan
Martin Meng

***Abstract*—This document will outline our project proposal for COMP 790 this semester. For our project we plan to prove three properties of symbolic execution that were first introduced in the seminal paper on the subject [1]. The three properties are:**

1) **The path condition never becomes identically false.**
2) **Path conditions associated with any two terminal leaves are distinct.**
3) **Symbolic execution defined for the Integer language in the paper is commutative.**

This proposal will provide some motivation and background as to why symbolic execution is important, precisely define the above properties, and provide an outline of our project's goals.

I. Introduction

For our project we will be diving deeper into the properties of symbolic execution defined by [1]. This is a powerful method for verifying correctness of programs which, in essence, replaces fixed inputs provided to a program with symbolic variables.

A symbolic execution engine is the tool that drives program execution while keeping track of the "execution state." For most engines, this will consist of two main components.

1) The path condition: a boolean formula describing the conditions satisfied by branches taken along a path.
2) A symbolic store with mappings between program variables and symbolic expressions.

We frame our reasoning about the different possible executions of a program with execution trees. An execution tree is a way of representing different paths through the program as taken by some symbolic execution engine. Each node in the tree maps to an executed statement, and the transitions between statements are directed arrows, symbolizing a change in execution state. Each node represents one such state, with information such as the path condition and current variable values.

This tree can become much more complex when different control flow structures become involved. For example, an *if* statement will result in a fork in the execution tree, with a resultant node for the TRUE and the FALSE branches. These execution trees have two properties that we hope to prove in the course of our project:

1) The path condition never becomes identically false.
2) Path conditions associated with any two terminal leaves are distinct.

Additionally, symbolic execution for the Integer language defined in [1] is commutative. This means that regardless of whether the program is executed symbolically, and then symbols are replaced with values, or vice versa, the end result will be the same. The final portion of our project will be focused on proving this.

The remainder of this proposal is structured as follows: in Section II we provide some background on symbolic execution and motivation for our project. Then in Section III we will describe the properties we will set out to prove and the timeline in which we hope to do so.

II. Background and Motivation

A. Symbolic Execution

Now we will give a basic overview of the history and core ideas of symbolic execution.

Developers employ a wide range of methods to track down bugs in their software code, from unit testing and code reviews to static analysis, but none are perfect. Deeply buried and hard-to-find bugs can be hidden away in rarely-used pieces of code, or can result from program states that are seldom reached. These factors make it almost impossible to completely test and formally verify a complex software system, even when execution is fully deterministic. Developers can use static analysis techniques to reason about certain possible executions of a piece of code, but these methods have their own issues. If programmers miss test cases, static analysis can lead them to believe that their faulty code is correct. And exhaustively checking every possible program execution

state is not a scalable solution for most systems. Traditional verification techniques relying on proofs are guaranteed to work, but are resource-intensive and largely unfamiliar to developers. Symbolic execution strikes a middle ground between these exhaustive static analysis tools and traditional verification proofs.

Symbolic execution is a technique that was first introduced in 1976 as a way to check if certain properties could possibly be violated in a software program. For example, we might want to ensure that a NULL pointer is never dereferenced, or that there are no divide-by-zero operations attempted. Symbolic execution generalizes software testing by representing inputs with symbols, where each symbol corresponds to a set of input values. This means each symbolic execution stands in for many program runs with concrete values: in particular, one symbolic execution represents exactly the set of runs whose values satisfy the path condition (as defined in Section I). Each executable program statement generates formulas and new expressions over the input symbols. When a branching statement is reached, the symbolic execution engine will fork, considering each possible path in turn.

In general, as the execution progresses, taking branches will update the path constraints, and assignment statements will update the symbol mappings in the current state. At the end of a line of execution or after some constrained amount of time, we typically use a satisfiability modulo theories (SMT) solver to verify if the path constraint has a satisfying assignment. This solver helps us determine whether a path is realizable or if some property was violated.

B. Related Work

In this section, we will give details about similar proofs done, both related to symbolic execution and in Coq more generally.

To the best of our knowledge, none of the properties described in the King paper have been proven formally in any proof assistant software, Coq or otherwise. The basic informal argument for the first property they provide is as follows:

When we take the THEN or ELSE branches of an if statement, for example, we add the necessary boolean expression to our path condition. However, when we execute a non-forking statement, the path condition remains the same since no new constraints need to be satisfied for that path to be valid. Thus, the path condition can never become identically false since it is initialized to be true and the only updates performed to it are adding further constraints. We start out with true, and perform "and" operations with the current path condition and new satisfiable boolean expressions. Note that this only holds when all of the conditions we append to the path condition are satisfiable.

Our project and proofs will relate exclusively to the simple Integer language presented in [1]. There has been some work to prove properties independent of the language used for execution. For example, [2] presented a "Generic Framework for Symbolic Execution," which set out to provide a language-independent theory and set of properties about symbolic execution. The authors define symbolic execution as the application of rewrite rules based on a language's semantics and provide a tool that uses their "K" framework. They use "CinK", a kernel of the C++ programming language, as the running example throughout their paper, and give a formal definition of CinK with functional symbols on page 27 of their paper. The authors also prove its correctness in Coq in order to formalize a verification of the Knuth-Morris-Pratt string matching algorithm written in CinK. Under future work, they listed potentially scripting a process to translate their "Reaching Logic" formulas into Coq. But none of these goals deal specifically with formalizing proofs for the three properties above in the Integer language in particular.

Gillian is another language-independent framework developed for use with symbolic analysis tools [3]. Their work uses the power of symbolic execution and other formal verification methods to prove correctness of software. Instead of directly proving properties of their framework in Coq like we are planning to do, they use Coq-verified modules as part of their workflow. The tool the authors developed uses a "fully parametric metatheory" that aims to unify symbolic execution across different target languages. As an example, they present their methodology towards verification of a C program. They define a symbolic memory model using an OCaml module and also make use of the CompCert compiler which was written and proven in Coq.

A general, extensible, reusable formal memory (GERM) framework for Coq has been developed to support different formal verification specifications [4]. They present an extension to the Curry-Howard isomorphism that combines symbolic execution and theorem proving. Their goal here is to improve the automation in higher-order logic theorem prover tools. Their objective was using symbolic execution to develop theorem proving methods, but we will be using Coq to formalize and prove properties laid out in [1].

III. Project Proposal

A. Description of the problem(s) and subgoals

Now we will describe the three subgoals we hope to prove by the end of the semester.

Minimum goal: Prove that the path condition never becomes identically false. This is equivalent to saying that for each terminal leaf that marks the end of an execution path, there exists a non-symbolic input that, when fed back into the program, will trace the same path.

The minimum goal is logically quite simple; indeed, it only corresponds to roughly a paragraph of reasoning in [1]. However, before starting to prove this goal we will need to translate most concepts relating to symbolic execution and the Integer pseudocode language into Coq. This translation effort will be the bulk of the work necessary to prove our minimum goal.

Standard goal: Prove that path conditions associated with any two terminals leaves are distinct. In other words, two paths which came from a common root have some unique node where their paths diverged.

This property is also related to property one. For example, if you take the true branch of some forking statement, and the expression p is conjoined to the path condition, the path condition for the execution run taking the false branch now includes the negation of p. Since the path condition never becomes identically false, this must be property must hold.

Reach goal: The symbolic execution defined in [1] is commutative. This means that if you execute a program with some concrete inputs, you will get the same results as symbolically executing the program and instantiating the symbols at the end.

The commutativity of symbolic execution is one of the main reasons for its success and widespread adoption. Symbolic execution captures the same behaviors as standard execution of a program, and as such is a natural extension of it. Additionally, the commutativity property allows us to prove the correctness of a path by executing it symbolically. Essentially, using the nomenclature in [1], place an ASSUME statement at the beginning of the program and a PROVE statement at the end, then begin regular symbolic execution. If the PROVE is true at the end, the path is correct.

B. Potential Challenges

One early problem we will need to overcome relates to translating symbolic execution concepts (execution state, path conditions, branching, etc.) into Coq. These are not simply properties about computing primitives, but meta-properties about transitions between computation states. We'll need to determine how much state we'll need to keep track of in Coq. In particular, we'll need to determine how much of the Integer pseudocode language will be required in order to prove our three goals above. We hope to avoid writing a full compiler for Integer in Coq, and perhaps we can simulate the required operations natively in Coq's syntax. But we will need to determine the level of abstraction at which we implement Integer operations.

C. Timeline

The following is our estimated timeline for completing the project. This may change as the semester goes on, but hopefully will be a good baseline.

1) Translate all of the terminology, notation and properties into Coq by: 3/07/21
2) Complete proof for minimal goal by: 3/15/21
3) Complete proof for standard goal by: 3/28/21
4) Complete proof for reach goal by: 4/15/21
5) Complete progress report by: 3/30/21
6) Complete final report by: 4/25/21
7) Prepare final presentation by: 5/3/2021

D. Who Did What

Kaki and Matthew contributed equally to this proposal. The three of us, including Martin, have been involved in the scoping out of the project and hopefully will all work together on the implementation, too.

References

[1] James C. King. 1976. Symbolic execution and program testing. Commun. ACM 19, 7 (July 1976), 385–394. http://www.cs.umd.edu/class/fall2014/cmsc631/papers/king-symbolic-execution.pdf

[2] Andrei Arusoaie, Dorel Lucanu, Vlad Rusu. A Generic Framework for Symbolic Execution. [Research Report] RR-8189, Inria. 2014, pp.27. https://hal.inria.fr/hal-00766220v6/document.

[3] José Fragoso Santos and Petar Maksimović and Sacha-Élie Ayoun and Philippa Gardner. Gillian: Compositional Symbolic Execution for All. 2020. https://arxiv.org/pdf/2001.05059.pdf.

[4] Zheng Yang and Hang Lei. A general formal memory framework in Coq for verifying the properties of programs based on higher-order logic theorem proving with increased automation, consistency, and reusability. 2018. https://arxiv.org/pdf/1803.00403.pdf.