15-414: Bug Catching: Automated Program Verification

Lecture Notes on SMT Encodings

Ruben Martins

Carnegie Mellon University

Lecture 18
Thursday, April 8, 2021

1 Introduction

In the previous lecture, we studied the congruence closure algorithm to solve formulas in the theory of equality with uninterpreted functions. In this lecture, we will see how we can use SMT solvers to prove the equivalence of programs. Uninterpreted functions can be used to help to verify programs by abstracting complex functions that may be hard to reason about. When compared to SAT, encoding using SMT is much easier and allows the combination of different theories. In practice, when the domain is finite and can be encoded compactly to SAT, then SAT solvers are usually faster than SMT solvers. However, SMT is more general and allows to solve problems that cannot be encoded to SAT. [1]

2 Proving equivalence of programs

Replacing functions with uninterpreted functions in a given formula is a common technique for making it easier to reason about (e.g., to prove its validity) At the same time, this process makes the formula *weaker* which means that it can make a valid formula invalid. This observation is summarized in the following relation, where φ^{UF} is derived from a formula φ by replacing some or all of its functions with uninterpreted functions:

$$\models \varphi^{UF} \to \varphi$$

[1]Lectures notes based on [BM07] and [KS16].

```c
int power3(int in)
{
   int i, out_a;
   out_a = in;
   for (i = 0; i < 2; i++)
     out_a = out_a * in;
  return out_a;
}
```

(a)

```c
int power3_new(int in)
{
   int out_b;

   out_b = (in * in) * in;

   return out_b;
}
```

(b)

Figure 1: Two C functions. We can simplify the proof of their equivalence by replacing the multiplication operator by an uninterpreted function.

Uninterpreted functions are widely used in calculus and other branches of mathematics, but in the context of reasoning and verification, they are mainly used for simplifying proofs. Under certain conditions, uninterpreted functions let us reason about systems while ignoring the semantics of all functions, assuming they are not necessary for the proof.

Assume that we have a method for checking the validity of a Σ-formula in T_{E}. Relying on this assumption, the basic scheme for using uninterpreted functions is the following:

1. Let φ denote a formula of interest that has interpreted functions. Assume that a validity check of φ is too hard (computationally), or even impossible.

2. Assign an uninterpreted function to each interpreted function in φ. Substitute each function in φ with the uninterpreted function to which it is mapped. Denote the new formula by φ^{UF}.

3. Check the validity of φ^{UF}. If it is valid then φ is valid. Otherwise, we do not know anything about the validity of φ.

As a motivating example consider the problem of proving the equivalence of two C functions shown in Figure 1. In general, proving the equivalence of two programs is undecidable, which means there is no sound and complete to prove such an equivalence. However, in this case, equivalence can be decided since the program does not have unbounded memory usage. A key observation about these programs is that they have only bounded loops, and therefore it is possible to compute their input/output relations. The derivation of these relations from these two programs can be as follows:

1. Remove the variable declarations and "return statements".

2. Unroll the **for** loop.

3. Replace the left-hand side variable in each assignment with a new auxiliary variable.

$$\begin{aligned} out0\_a =& in0\_a \land \\ out1\_a =& out0\_a * in0\_a \land \\ out2\_a =& out1\_a * in0\_a \end{aligned}$$

(a) (φ_a)

$$out0\_b = (in0\_b * in0\_b) * in0\_b$$

(b) (φ_a)

Figure 2: Two formulas corresponding to the programs (a) and (b) in Figure 1.

4. Whenever a variable is read, replace it with the auxiliary variable that replaced it in the last place where it was assigned.

5. Conjoin all program statements.

These operations result in the two formulas φ_a and φ_b which are shown in Figure 2. This procedure to transform code into a first-order formula is known as *static single assignment* (SSA) and was first introduced in Lecture 15. Even though generalizing SSA to programs with "if" branches and other constructs can be challenging, we restrict ourselves to a limited form of SSA to illustrate how uninterpreted functions can be used to abstract the multiplication operator.

To show that these programs are equivalent with respect to their input-outputs, we must show that the following formula Φ is valid:

$$in0\_a = in0\_b \land \varphi_a \land \varphi_b \rightarrow out2\_a = out0\_b$$

Showing the validity of Φ is equivalent to show the unsatisfiability of $\neg\Phi$. We can show that $\neg\Phi$ is unsatisfiable by using SMT solvers.

3 Using SMT solvers

SMT solvers take as input a formula in a standardized format (SMT2-Lib format). A detailed description of the SMT2-Lib format is available at:

http://smtlib.cs.uiowa.edu

SMT solvers support a variety of theories, namely: the theory of arrays with extensionality, the theory of bit vectors with an arbitrary size, the core theory defining the basic Boolean operators, the theory of floating-point numbers, the theory of integer number, and the theory of reals. [2]

If you want to try SMT solving, we recommend doing the z3 tutorial at:

https://rise4fun.com/z3/tutorial

[2]Further details on each theory are available at http://smtlib.cs.uiowa.edu/theories.shtml.

```
(declare-fun out0_a () (Int))
(declare-fun out1_a () (Int))
(declare-fun in0_a () (Int))
(declare-fun out2_a () (Int))
(declare-fun out0_b () (Int))
(declare-fun in0_b () (Int))
(define-fun phi_a () Bool
  (and (= out0_a in0_a) ; out0_a = in0_a
      (and (= out1_a (* out0_a in0_a)) ; out1_a = out0_a * in0_a
        (= out2_a (* out1_a in0_a))))) ; out2_a = out1_a * in0_a
(define-fun phi_b () Bool
  (= out0_b (* (* in0_b in0_b) in0_b))) ; out0_b = in0_b * in0_b *
      in0_b
(define-fun phi_input () Bool
  (= in0_a in0_b))
(define-fun phi_output () Bool
  (= out2_a out0_b))
(assert (not (=> (and phi_input phi_a phi_b) phi_output)))
(check-sat)
```

Figure 3: SMT encoding of Φ using mathematical integers to model integers.

and trying z3 online at:

https://rise4fun.com/z3/

Before using SMT solvers to show that $\neg\Phi$ is unsatisfiable, we must decide how we will model integers since this will restrict the underlying theories used by the SMT solver.

3.1 Modeling integers as mathematical integers

If we model integers as mathematical integers then the SMT solver will use the theory of integers and will be able to show that both programs are equivalent. Figure 3 shows the SMT encoding of Φ when using integers: You can try this encoding online at:

https://rise4fun.com/Z3/BLQpl

When modeling integers as mathematical integers, we can prove the equivalence of these programs quickly and without any issues. However, integers are not represented as mathematical integers in C. If we want to model integers as the ones being used in C then we should model them using bit vectors (of size 32 or 64).

3.2 Modeling integers as bit vectors

Modeling integers as bit vectors has the advantage of capturing the C model and being able to detect potential overflows. However, using the bit vector theory is not as efficient as using the theory of integers. In particular, assume we want to show that

```
(declare-fun out0_a () (_ BitVec 512))
(declare-fun out1_a () (_ BitVec 512))
(declare-fun in0_a () (_ BitVec 512))
(declare-fun out2_a () (_ BitVec 512))
(declare-fun out0_b () (_ BitVec 512))
(declare-fun in0_b () (_ BitVec 512))
(define-fun phi_a () Bool
  (and (= out0_a in0_a) ; out0_a = in0_a
      (and (= out1_a (bvmul out0_a in0_a)) ; out1_a = out0_a * in0_a
        (= out2_a (bvmul out1_a in0_a))))) ; out2_a = out1_a * in0_a
(define-fun phi_b () Bool
  (= out0_b (bvmul (bvmul in0_b in0_b) in0_b))) ; out0_b = in0_b *
      in0_b * in0_b
(define-fun phi_input () Bool
  (= in0_a in0_b))
(define-fun phi_output () Bool
  (= out2_a out0_b))
(assert (not (=> (and phi_input phi_a phi_b) phi_output)))
(check-sat)
```

Figure 4: SMT encoding of Φ using bit vectors to model integers.

the programs are equivalent to a bit width of 512. The SMT encoding when using bit vectors is shown in the Figure 5. You can try this encoding online at:

https://rise4fun.com/Z3/ibsw3

This formula is much more challenging to be solved than the previous one and will become harder as the bit-width increases. For example, if you try it online you will get an out-of-memory error. You can also try it on your own computer (since you should have z3 installed) by running the following command:

```
$ z3 -smt2 formula
```

where the formula is a file with the contents of Figure 5. The reason for the memory blowup is the multiplication operator when using bit vectors. Can we avoid this issue altogether? What if we consider the multiplication operator as an uninterpreted function?

3.3 Using uninterpreted functions

If we consider an uninterpreted function f that takes as input two bit vectors and returns a bit vector then we can replace the bit vector multiplication operator (bvmul) by f. If we are able to prove that this formula is unsatisfiable, then we can conclude that the original formula is also unsatisfiable and we are able to show the equivalence between the two programs when representing integers by bit vectors of width 512. This formula is much easier to be solved than the one using bit-vector multiplication since

```
(declare-fun out0_a () (_ BitVec 512))
(declare-fun out1_a () (_ BitVec 512))
(declare-fun in0_a () (_ BitVec 512))
(declare-fun out2_a () (_ BitVec 512))
(declare-fun out0_b () (_ BitVec 512))
(declare-fun in0_b () (_ BitVec 512))
(declare-fun f ((_ BitVec 512) (_ BitVec 512)) (_ BitVec 512))
(define-fun phi_a () Bool
  (and (= out0_a in0_a) ; out0_a = in0_a
      (and (= out1_a (f out0_a in0_a)) ; out1_a = out0_a * in0_a
        (= out2_a (f out1_a in0_a))))) ; out2_a = out1_a * in0_a
(define-fun phi_b () Bool
  (= out0_b (f (f in0_b in0_b) in0_b))) ; out0_b = in0_b * in0_b *
      in0_b
(define-fun phi_input () Bool
  (= in0_a in0_b))
(define-fun phi_output () Bool
  (= out2_a out0_b))
(assert (not (=> (and phi_input phi_a phi_b) phi_output)))
(check-sat)
```

Figure 5: SMT encoding of Φ using an uninterpreted function for multiplication.

we abstracted the multiplication function and the SMT solver will not need to reason about what f does but only that it is a function. You can try this encoding online at:

https://rise4fun.com/Z3/V7Sf

4 Modeling: SAT vs. SMT

Recall the graph coloring problem that we modeled with SAT in Lecture 13.

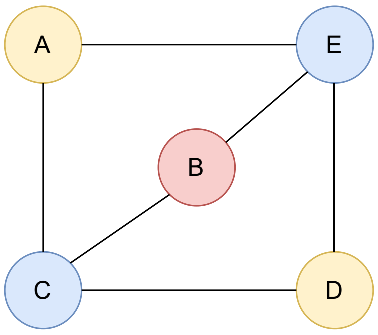

Figure 6: 3-coloring of a graph.

To encode the 3-coloring problem of the graph presented in Figure 6 to SAT, we required 15 variables and 38 clauses. However, when encoding this problem to SMT, we can see this can be done in a more compact and simpler way. Since we can encode variables with integers, we can have the integer domain represent the possible colors.

```
(declare-fun A () Int)
(declare-fun B () Int)
(declare-fun C () Int)
(declare-fun D () Int)
(declare-fun E () Int)
(assert (not (= A E)))
(assert (not (= A C)))
(assert (not (= B E)))
(assert (not (= B C)))
(assert (not (= B D)))
(assert (not (= C D)))
(assert (not (= D E)))
(assert (and (>= A 0) (<= A 2)))
(assert (and (>= B 0) (<= B 2)))
(assert (and (>= C 0) (<= C 2)))
(assert (and (>= D 0) (<= D 2)))
(assert (and (>= E 0) (<= E 2)))
(check-sat)
(get-model)
```

SMT formulas when written in SMT-LIB format also have the advantage that they are easier to read than CNF formulas since variables can have names and restrictions are more readable. When modeling problems to logic, unless the performance is critical, SMT is often more used than SAT.

5 Summary

- Uninterpreted functions can be used to simplify proofs by replacing (complex) interpreted functions by uninterpreted functions;
- Let φ^{UF} be φ with all its interpreted functions replaced by uninterpreted functions. Then:
$$\models \varphi^{UF} \rightarrow \varphi$$
- We can use SMT solvers to check if two programs are equivalent:
  - If we represent integers as mathematical integers then the problem is relatively easy to be solved;
  - If we represent integers as bit vectors then the problem becomes more challenging because of bit vector multiplication;
  - If we abstract bit vector multiplication with uninterpreted functions then we can achieve scalability and prove that two programs are equivalent for any bit width.
- Modeling problems with SMT is easier than with SAT.

References

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[KS16] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2016.