15-414: Bug Catching: Automated Program Verification

Lecture Notes on DPLL(T) & SMT Encodings

Ruben Martins

Carnegie Mellon University
Lecture 19
Thursday, March 28, 2024

1 Introduction

In the previous lecture, we studied the Nelson-Oppen procedure for solving a combination of theories. However, this procedure has some limitations, and in practice, most SMT solvers use DPLL(T) to solve a combination of theories. In this lecture, we will continue explaining the DPLL(T) approach and see some examples of its usage.

This lecture will also show how we can use SMT solvers in practice. In particular, we will show 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 us to solve problems that cannot be encoded to SAT.

Learning Goals

1. DPLL(T) combines a conjunctive theory solver and DPLL to decide formulas in a given first-order theory.
2. Just as conflict clauses were important for DPLL, learning theory lemmas can dramatically improve the performance of DPLL(T).
3. When reaching a conflict DPLL(T) can minimize the learned clause by computing unsatisfiable subformulas.

4. Uninterpreted functions can be used to simplify proofs by replacing (complex) interpreted functions by uninterpreted functions.
5. SMT solvers can be used in practice for program equivalence, graph coloring, and other applications.

2 Review: Boolean abstraction

We define the Boolean abstraction of a Σ-formula φ recursively:

- <literal> ::= <atom>$_T$ | $\neg$ <atom>$_T$
- <formula> ::= <literal> $\quad \mathcal{B}(l_T) \stackrel{\text{def}}{=} P_i$, where P_i is a fresh variable
- <formula> ::= $\neg$ <formula> $\quad \mathcal{B}(\neg F) \stackrel{\text{def}}{=} \neg \mathcal{B}(F)$
- <formula> ::= <formula> $\wedge$ <formula> $\quad \mathcal{B}(F_1 \wedge F_2) \stackrel{\text{def}}{=} \mathcal{B}(F_1) \wedge \mathcal{B}(F_2)$
- <formula> ::= <formula> $\vee$ <formula> $\quad \mathcal{B}(F_1 \vee F_2) \stackrel{\text{def}}{=} \mathcal{B}(F_1) \vee \mathcal{B}(F_2)$
- <formula> ::= <formula>$\rightarrow$ <formula> $\quad \mathcal{B}(F_1 \rightarrow F_2) \stackrel{\text{def}}{=} \mathcal{B}(F_1) \rightarrow \mathcal{B}(F_2)$
- <formula> ::= <formula>$\leftrightarrow$ <formula> $\quad \mathcal{B}(F_1 \leftrightarrow F_2) \stackrel{\text{def}}{=} \mathcal{B}(F_1) \leftrightarrow \mathcal{B}(F_2)$

Given a Σ-formula φ:

$$\varphi : g(a) = c \wedge (f(g(a)) \neq f(c) \vee g(a) = d) \wedge c \neq d$$

The Boolean abstraction of φ is the following:

$$\begin{aligned}
\mathcal{B}(F) &= \mathcal{B}(g(a) = c) \wedge \mathcal{B}(f(g(a)) \neq f(c) \vee g(a) = d) \wedge c \neq d) \\
&= \mathcal{B}(g(a) = c) \wedge \mathcal{B}(f(g(a)) \neq f(c) \vee g(a) = d)) \wedge \mathcal{B}(c \neq d) \\
&= \mathcal{B}(g(a) = c) \wedge \mathcal{B}(f(g(a)) \neq f(c)) \vee \mathcal{B}(g(a) = d) \wedge \mathcal{B}(c \neq d) \\
&= P_1 \wedge (\neg P_2 \vee P_3) \wedge \neg P_4
\end{aligned}$$

Note that we can also define $\mathcal{B}^{-1}$ which maps from the Boolean variables back to the atoms in the original formula. For example $\mathcal{B}^{-}1(P_1 \wedge P_3 \wedge P_4)$ corresponds to the formula $g(a) = c \wedge g(a) = d \wedge c = d$.

We call $\mathcal{B}(\varphi)$ an abstraction of φ since it is an over-approximation of φ with respect to satisfiability. Observe the following properties of this over-approximation:

- If φ is satisfiable then $\mathcal{B}(\varphi)$ is also satisfiable;
- If $\mathcal{B}(\varphi)$ is satisfiable then φ is not necessarily satisfiable:

  $$\varphi : 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$$

  φ is unsatisfiable in the theory of integers ($T_{\mathbb{Z}}$) since x is either 1 or 2 but $f(x) \neq f(1) \wedge f(x) \neq f(2)$ implies that x must be different than 1 and 2. However, the Boolean abstraction $\mathcal{B}(\varphi) = P_1 \wedge P_2 \wedge P_3 \wedge P_4$ is satisfiable.

- If φ is unsatisfiable then $\mathcal{B}(\varphi)$ is not necessarily unsatisfiable:

$$\varphi : 1 \leq x \wedge x \leq 2 \wedge f(x) \neq f(1) \wedge f(x) \neq f(2)$$

  The same example as for the previous case holds for this case as well. φ is unsatisfiable in the theory of integers ($T_{\mathbb{Z}}$) but $\mathcal{B}(\varphi)$ is satisfiable.

- If $\mathcal{B}(\varphi)$ is unsatisfiable then φ is also unsatisfiable.

3 DPLL(T): Combining theory and SAT solvers

The Boolean abstraction provides us with a **lazy** way to solve SMT. Given a Σ-formula φ, we can determine its satisfiability by performing the following procedure:

1. Construct the Boolean abstraction $\mathcal{B}(\varphi)$;
2. If $\mathcal{B}(\varphi)$ is unsatisfiable then φ is unsatisfiable;
3. Otherwise, get an interpretation I for $\mathcal{B}(\varphi)$;
4. Construct $\omega = \bigwedge_{i=1}^{n} P_i \leftrightarrow I(P_i)$;
5. Send $B^{-1}(\omega)$ to the T-solver;
6. If T-solver reports that $B^{-1}(\omega)$ is satisfiable then φ is satisfiable;
7. Otherwise, update $\mathcal{B}(\varphi) := \mathcal{B}(\varphi) \wedge \neg\omega$ and return to step 2.

This procedure terminates when: (i) $\mathcal{B}(\varphi)$ becomes unsatisfiable which implies that φ is also unsatisfiable or (ii) T-solver reports that $B^{-1}(\omega)$ is satisfiable which implies that φ is satisfiable. Note that if $B^{-1}(\omega)$ is unsatisfiable we cannot terminate since there may be another interpretation ω' to $\mathcal{B}(\varphi)$ that would make $B^{-1}(\omega')$ satisfiable. Therefore, we need to exhaust all interpretations for $\mathcal{B}(\varphi)$ before deciding that φ is unsatisfiable. On step 7 we add $\neg\omega$ to $\mathcal{B}(\varphi)$ since if we did not, we would get the same interpretation I for $\mathcal{B}(\varphi)$. We denote $\neg\omega$ as a **theory conflict clause** that prevents the SAT solver from going down the same path in future iterations.

Suppose we want to find if the Σ-formula φ is satisfiable:

$$\varphi : g(a) = c \wedge (f(g(a)) \neq f(c) \vee g(a) = d) \wedge c \neq d$$

We start by building its Boolean abstraction $\mathcal{B}(\varphi)$:

$$\mathcal{B}(\varphi) : P_1 \wedge (\neg P_2 \vee P_3) \wedge \neg P_4$$

Table 1 shows the step 1 of the procedure with the corresponding Boolean abstraction $\mathcal{B}(\varphi)$. Next, we query the SAT solver for an interpretation to $\mathcal{B}(\varphi)$. Assume that the SAT solver returns the following interpretation $I = \{P_1, \neg P_2, P_3, \neg P_4\}$. We construct

Theory solver	SAT solver
	$P_1 \wedge (\neg P_2 \vee P_3) \wedge \neg P_4$

Table 1: $\mathcal{B}(\varphi)$.

Theory solver	SAT solver
$g(a) = c \wedge$ $f(g(a)) \neq f(c) \wedge$ $g(a) = d \wedge$ $c \neq d$	$P_1 \wedge (\neg P_2 \vee P_3) \wedge \neg P_4$ $(\neg P_1 \vee P_2 \vee \neg P_3 \vee P_4)$
$g(a) = d \wedge g(a) = c \rightarrow c = d$ $c \neq d$ unsat	

Table 2: Updated $\mathcal{B}(\varphi)$ after checking that the interpretation $I = \{P_1, \neg P_2, P_3, \neg P_4\}$ does not satisfy φ

$\omega = (P_1 \wedge \neg P_2 \wedge P_3 \wedge \neg P_4)$ and send $\mathcal{B}^{-1}(\omega)$ to T-solver. Note that $\mathcal{B}^{-1}(\omega)$ corresponds to:

$$\mathcal{B}^{-1}(\omega) : g(a) = c \wedge f(g(a)) \neq f(c) \wedge g(a) = d \wedge c \neq d$$

$\mathcal{B}^{-1}(\omega)$ is unsatisfiable since if $g(a) = d$ and $g(a) = c$ then $c = d$ but φ states that $c \neq d$. Therefore, we know that this interpretation is not satisfiable but there may exist another interpretation I that satisfies φ. We update $\mathcal{B}(\varphi)$ with $\neg\omega$ as shown in Table 2 and query the SAT solver for another interpretation.

Assume that the SAT solver returns a new interpretation $I = \{P_1, P_2, P_3, \neg P_4\}$. We construct $\omega = (P_1 \wedge P_2 \wedge P_3 \wedge \neg P_4)$ and send $\mathcal{B}^{-1}(\omega)$ to T-solver. Note that in this case $\mathcal{B}^{-1}$ corresponds to:

$$\mathcal{B}^{-1}(\omega) : g(a) = c \wedge f(g(a)) = f(c) \wedge g(a) = d \wedge c \neq d$$

We can see that $\mathcal{B}^{-1}(\omega)$ is unsatisfiable for the same reason as before. We update $\mathcal{B}(\varphi)$ with $\neg\omega$ as shown in Table 3 and perform another query to the SAT solver.

Assume that the SAT solver returns a new interpretation $I = \{P_1, \neg P_2, \neg P_3, \neg P_4\}$. We construct $\omega = (P_1 \wedge \neg P_2 \wedge \neg P_3 \wedge \neg P_4)$ and send $\mathcal{B}^{-1}(\omega)$ to T-solver. Note that in this case $\mathcal{B}^{-1}$ corresponds to:

$$\mathcal{B}^{-1}(\omega) : g(a) = c \wedge f(g(a)) \neq f(c) \wedge g(a) \neq d \wedge c \neq d$$

We can see that $\mathcal{B}^{-1}(\omega)$ is unsatisfiable since $g(a) = c$ but $f(g(a)) \neq f(c)$. We update $\mathcal{B}(\varphi)$ with $\neg\omega$ as shown in Table 4 and observe that $\mathcal{B}(\varphi)$ becomes unsatisfiable after adding $\neg\omega$. Since $\mathcal{B}(\varphi)$ is unsatisfiable, we can conclude that φ is also unsatisfiable.

3.1 Improving DPLL(T) framework

Consider the Σ-formula φ defined over $T_{\mathbb{Z}}$:

$$\varphi : 0 < x \wedge x < 1 \wedge x < 2 \wedge \ldots x < 99$$

Theory solver	SAT solver
$g(a) = c \wedge$ $f(g(a)) = f(c) \wedge$ $g(a) = d \wedge$ $c \neq d$	$P_1 \wedge (\neg P_2 \vee P_3) \wedge \neg P_4$ $(\neg P_1 \vee P_2 \vee \neg P_3 \vee P_4)$ $(\neg P_1 \vee \neg P_2 \vee \neg P_3 \vee P_4)$
$g(a) = d \wedge g(a) = c \rightarrow c = d$ $c \neq d$ unsat	

Table 3: Updated $\mathcal{B}(\varphi)$ after checking that the interpretation $I = \{P_1, P_2, P_3, \neg P_4\}$ does not satisfy φ.

Theory solver	SAT solver
$g(a) = c \wedge$ $f(g(a)) \neq f(c) \wedge$ $g(a) \neq d \wedge$ $c \neq d$	$P_1 \wedge (\neg P_2 \vee P_3) \wedge \neg P_4$ $(\neg P_1 \vee P_2 \vee \neg P_3 \vee P_4)$ $(\neg P_1 \vee \neg P_2 \vee \neg P_3 \vee P_4)$ $(\neg P_1 \vee P_2 \vee P_3 \vee P_4)$
$g(a) = c \rightarrow f(g(a)) = f(c)$ $f(g(a)) \neq f(c)$ unsat	unsat

Table 4: Updated $\mathcal{B}(\varphi)$ after checking that the interpretation $I = \{P_1, \neg P_2, \neg P_3, \neg P_4\}$ does not satisfy φ. $\mathcal{B}(\varphi)$ becomes unsatisfiable after adding the negation of I.

The Boolean abstraction $\mathcal{B}(\varphi)$ is the following:

$$\mathcal{B}(\varphi) : P_0 \wedge P_1 \wedge \ldots \wedge P_{99}$$

Note that $\mathcal{B}(\varphi)$ has 2^{98} interpretations containing $P_0 \wedge P_1$ and none of them satisfies φ. The procedure described in the previous section will enumerate all of them one by one and add a blocking conflict clause that only covers a single assignment! A potential solution to this issue is to not treat the SAT solver as a black box but instead incrementally query the theory solver as interpretations are made in the SAT solver. If we would perform this integration then we would be able to stop after adding $\{0 < x, x < 1\}$ and would not need to explore the 2^{98} infeasible interpretations. This can be done by pushing the T-solver into the DPLL algorithm as follows:

1. After Boolean Constraint Propagation (BCP), invoke the T-solver on the partial interpretation;
2. If the T-solver returns unsatisfiable then we can stop the search of the SAT solver and immediately add $\neg\omega$ to $\mathcal{B}\varphi$;
3. Otherwise, continue as usual until we have a new partial interpretation.

Recall the example:

$$\varphi : g(a) = c \wedge (f(g(a)) \neq f(c) \vee g(a) = d) \wedge c \neq d$$

And its Boolean abstraction $\mathcal{B}(\varphi)$:

$$\mathcal{B}(\varphi) : P_1 \wedge (\neg P_2 \vee P_3) \wedge \neg P_4$$

DPLL would begin by propagating P_1 and $\neg P_4$ since they are unit clauses. At this point the theory axioms imply more propagations:

$$g(a) = c \rightarrow f(g(a)) = f(c)$$
$$g(a) = c \wedge c \neq d \rightarrow g(a) \neq d$$

Deciding $\neg P_2$ or P_3 would be wasteful, so we can add the **theory lemmas**:

$$(P_1 \rightarrow P_2)$$
$$(P_1 \wedge \neg P_3) \rightarrow \neg P_3$$

This procedure is called **theory propagation** and can guarantee that every Boolean interpretation is T-satisfiable. However, in practice doing this at every step can be expensive and theory propagation is only applied when it is "likely" (using heuristics) to derive useful implications.

Another optimization that can be performed is to minimize the conflict clause ω that we add to $\mathcal{B}(\varphi)$ to contain only the root cause of the issue. Consider again the Σ-formula φ:

$$\varphi : g(a) = c \wedge (f(g(a)) \neq f(c) \vee g(a) = d) \wedge c \neq d$$

Notice that the interpretations $I = \{P_1, \neg P_2, P_3, \neg P_4\}$ and $I' = \{P_1, \neg P_2, P_3, \neg P_4\}$ had the same root cause that lead to φ being unsatisfiable under that interpretation, i.e. $g(a) = d$ and $g(a)$ which implies that $c = d$ but we know that $c \neq d$ which is a contradiction. Can we find the root cause of this issue and learn something stronger than $\omega = (\neg P_1 \vee P_2 \vee \neg P_3 \vee P_4)$? **Yes**, we can minimize ω using unsatisfiable cores!

Definition 1 (Minimal unsatisfiable core)**.** Let φ be an unsatisfiable formula and $\varphi_c \subseteq \varphi$. φ_c is a minimal unsatisfiable core if and only if:

- φ_c is unsatisfiable;
- Removing any element from φ_c makes φ_c satisfiable.

For $I = \{P_1, \neg P_2, P_3, \neg P_4\}$ we have the following $\mathcal{B}^{-1}(\varphi)$:

$$\mathcal{B}^{-1}(\varphi) : g(a) = c \wedge f(g(a)) \neq f(c) \wedge g(a) = d \wedge c \neq d$$

We can compute the minimal unsatisfiable core of $\mathcal{B}^{-1}(\varphi)$ as follows.

1. Drop $g(a) = c$. Is the formula still unsatisfiable? **No!** Then it means this constraint will be part of the minimal unsatisfiable core.

2. Drop $f(g(a)) \neq f(c)$. Is the formula still unsatisfiable? **Yes!** Then it means that we can remove this constraint from the minimal unsatisfiable core.

3. Now we have $g(a) = c \wedge g(a) = d \wedge c \neq d$.

4. Drop $g(a) = d$. Is the formula still unsatisfiable? No, then keep this constraint.

5. Drop $c \neq d$. Is the formula still unsatisfiable? No, then keep this constraint.

We can conclude that our minimal unsatisfiable core is $g(a) = c \wedge g(a) = d \wedge c \neq d$. Therefore, we can learn the clause $\omega' = (\neg P_1 \vee \neg P_3 \vee P_4)$ instead of $\omega = (\neg P_1 \vee P_2 \vee \neg P_3 \vee P_4)$ which would have save one query to the SAT solver in the previous section.

4 SMT Encodings: 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:

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

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

```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.

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

(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.

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.
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 we talk more about it in the next lecture about bounded model checking. 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:

```
(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.

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

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.

5 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. [1]

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.

5.1 Modeling integers as mathematical integers

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

```
(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.

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:

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).

5.2 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 the programs are equivalent to a bit width of 512. The SMT encoding when using bit vectors is shown in the Figure 5.

This formula is much more challenging to be solved than the previous one and will become harder as the bit-width increases. You can 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 why this formula is challenging to solve is because of the multiplication operator when using bit vectors. Can we avoid this issue altogether? What if we consider the multiplication operator as an uninterpreted function?

```
(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.

5.3 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 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.

6 Modeling: SAT vs. SMT

Recall the graph coloring problem that we modeled with SAT in Lecture 15.

To encode the 3-coloring problem of the graph presented in Figure 6 to SAT, we required 15 variables and 41 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)))
```

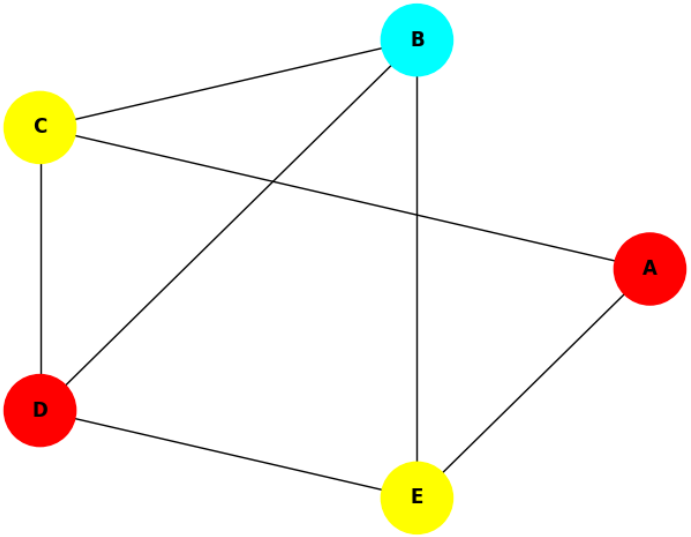

Figure 6: 3-coloring of a graph.

```
(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.