Symbolic Execution as DPLL Modulo Theories

Quoc-Sang Phan

Queen Mary University of London
q.phan@qmul.ac.uk

Abstract

We show how Symbolic Execution can be understood as a variant of the DPLL(\(T\)) algorithm, which is the dominant technique for the Satisfiability Modulo Theories (SMT) problem. In other words, Symbolic Executors are SMT solvers. This view enables us to use an SMT solver, with the ability of generating all models with respect to a set of Boolean atoms, to explore all symbolic paths of a program. This results in a more lightweight approach for Symbolic Execution.

1998 ACM Subject Classification D.2.4 Software/Program Verification

Keywords and phrases Symbolic Execution, Satisfiability Modulo Theories

1 Introduction

Symbolic Execution (SE) [11] is now popular. It is increasingly used not only in academic settings but also in industry, such as in Microsoft, NASA, IBM and Fujitsu [4]. In the success of SE, the efficiency of SMT solvers [9] is a key factor. In fact, while SE was introduced more than three decades ago, it had not been made practical until research in SMT made significant advances [5].

State-of-the-art SMT solvers, e.g. [8, 2], implement the DPLL(\(T\)) algorithm [15] which is an integration of two components as follows. The first component is a propositional satisfiability (SAT) solver, based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [7], to search on the Boolean skeleton of the formula. The second component is a decision procedure, called the \(T\)-solver, to check the consistency w.r.t. the theory \(T\) of conjunctions of literals, each literal is an atomic formula or the negation of an atomic formula. The path conditions generated by a Symbolic Executor, e.g. Symbolic PathFinder (SPF) [14], are also conjunctions of literals. Therefore, when an SMT solver checks such a path condition, only the \(T\)-solver works on it, and the SAT component is not used.

On the other hand, a classical Symbolic Executor [11] can be divided into two components. The first component, dubbed as Boolean Executor hereafter, executes the instructions, and updates the path condition. The second component is a \(T\)-solver (since the SAT solver is not used) to validate the consistency of the path condition. This paper shows that a Boolean Executor does the same work as the DPLL algorithm. Thus, SE is a variant of DPLL(\(T\)). This view is important since it connects two communities and can give an insight for future research.

Based on this new insight, we propose a lightweight approach for SE which uses the DPLL component of an SMT solver to explore symbolic paths instead of a Boolean Executor. Our approach relies on an SMT solver with the ability of generating all models w.r.t. a set of Boolean atoms.
2 Background

Before showing the correspondence between a Symbolic Executor and an SMT solver, we recall some background on Symbolic Execution and the SMT problem.

2.1 Symbolic Execution

Symbolic Execution [11] (SE) is a programming analysis technique which executes programs on unspecified inputs, by using symbolic values instead of concrete data. For each executed program path, SE builds a path condition \( pc \) which represents the condition on the inputs for the execution to follow that path.

```
function Executor(Program P)
{
    PathCondition pc = TRUE;
    InstructionPointer i = NULL;
    update(P, i);
    if (i == RETURN) return;
    while (True) {
        l = chooseLiteral(i);
        pc = pc ∧ l;
        update(P, i);
        if (i == RETURN)
            if (allStatesAreExplored())
                return;
            else backtrack(pc, i);
    }
}
```

Figure 1 A simplified Boolean Executor.

For an \textbf{if} statement with condition \( c \), there are three possible cases: (i) \( pc \vdash c \): SE chooses the \textbf{then} path; (ii) \( pc \vdash \neg c \): SE chooses the \textbf{else} path; (iii) \( (pc \not\vdash c) \land (pc \not\vdash \neg c) \): SE executes both paths: in the \textbf{then} path, it updates the path condition \( pc_1 = pc \land c \), in the \textbf{else} path it updates the path condition \( pc_2 = pc \land \neg c \). The satisfiability of the path condition is checked by SMT solvers, for example the symbolic executor KLEE [3] uses the SMT solver STP [10], while SPF provides a parameter to select one of the SMT solvers: CVC3, Yices, Z3. In this way, only feasible program paths are explored. Test generation is performed by solving the path conditions.

2.2 SMT and DPLL(\( \mathcal{T} \))

\textit{Satisfiability Modulo Theories} (SMT) is the problem of checking the satisfiability of logical formulas over one or more first-order theories \( \mathcal{T} \).

Our setting is standard first-order logic. Boolean variables are called Boolean atoms or simply atoms, and atomic formulas are called theory atoms or \( \mathcal{T} \)-atoms. A \textit{truth assignment} \( \mu \) for a formula \( \varphi \) is a truth value assignment to the \( \mathcal{T} \)-atoms of \( \varphi \). We define a bijective function \( BA \) (Boolean abstraction) which maps Boolean atoms into themselves and \( \mathcal{T} \)-atoms into fresh Boolean atoms. The \textit{Boolean refinement} function \( BR \) is then defined as the inverse of \( BA \), which means \( BR = BA^{-1} \).

At a high level, an SMT solver is the integration of two components: a SAT solver and \( \mathcal{T} \)-solvers. SMT solving can be viewed as the iteration of the two following steps. First, the
SAT solver searches on the Boolean abstraction of the formula, $\varphi^P = BA(\varphi)$, and returns a (partial) truth assignment $\mu^P$. The $\mathcal{T}$-solvers then check the Boolean refinement of the candidate, $BR(\mu^P)$, whether it is consistent with the theories $\mathcal{T}$.

A (finite) disjunction of literals $(l_1 \lor l_2 \cdots \lor l_k)$ is called a clause, and a literal $l_i$ is an atom or its negation. A clause that contains only one literal is called a unit clause. At a high level, DPLL is a stack-based depth-first search procedure which iteratively performs the two following steps: first choose a literal $l_i$ from the remaining clause, and add it to the current truth assignment; then apply Boolean Constraint Propagation (BCP), backtracking if there is a conflict. These two steps are repeated until a model is found or all states are explored without finding a model.

The procedure BCP for a literal $l_i$ removes all the clauses containing $l_i$, and removes $\neg l_i$ from the remaining clauses. If the removal results an empty clause, the search encounters a conflict.

### Illustration of DPLL($\mathcal{T}$)

A complete formal description of first-order theories and the DPLL($\mathcal{T}$) algorithm can be found in, e.g., [15]. Here we briefly introduce necessary preliminaries via a running example as follows.

$$\varphi := (\neg(x_0 > 5) \lor T_1) \land (x_0 > 5) \lor (x_1 = x_0 + 1) \land (\neg(x_1 < 3) \lor T_3) \land ((x_1 < 3) \lor (x_2 = x_1 - 1)) \land ((x_1 < 3) \lor ((y_1 = x_1 + 1))$$

(1)

$\varphi$ is a Linear Arithmetic formula. Boolean variables, $T_1 \ldots T_4$, are called Boolean atoms, and atomic formulas, e.g. $(x_0 > 5)$, are called theory atoms or $\mathcal{T}$-atoms. A first-order formula can
be abstracted into a Boolean skeleton by replacing all the $T$-atoms with new Boolean atoms, which is often called Boolean abstraction. For the example above, we define new Boolean variables $G_1, G_2, A_1, A_2, A_3$ for the Boolean abstraction of $T$-atoms, and the abstraction can be expressed as:

$$
\begin{align*}
BA & := G_1 = (x_0 > 5) \land G_2 = (x_1 < 3) \land \\
A_1 & = (x_1 = x_0 + 1) \land A_2 = (x_2 = x_1 - 1) \land A_3 = (y_1 = x_1 + 1)
\end{align*}
$$

As the result, we obtain a formula $\varphi^P$ ($P$ stands for propositional) as the Boolean skeleton of $\varphi$. Obviously, $\varphi$ is logically equivalent to $\varphi^P \land BA$.

$$
\begin{align*}
\varphi^P & := (\neg G_1 \lor T_1) \land (G_1 \lor T_2) \land (\neg G_1 \lor A_1) \land \\
& (G_2 \lor T_3) \land (\neg G_2 \lor A_2) \land \\
& (G_2 \lor T_4) \land (G_2 \lor A_3)
\end{align*}
$$

The DPLL($T$) algorithm is the integration of the DPLL algorithm with a $T$-solver. The DPLL algorithm searches on $\varphi^P$, returning a conjunction of Boolean literal $\mu^P$. Replacing all the new Boolean atoms, $G_i$ and $A_i$, in $\mu^P$ with their corresponding $T$-atoms, we obtain the conjunction $\mu$ in $T$. The $T$-solver then checks whether $\mu$ is consistent with the theory $T$. Below is the illustration of DPLL($T$) on $\varphi$ (for the limit of space, only decision literals are shown in $\mu^P$):

0. $\mu^P = \text{True} \quad \varphi^P$
1. $\mu^P = G_1 \quad \varphi^P = (\neg G_2 \lor T_3) \land (\neg G_2 \lor A_2) \land (G_2 \lor T_4) \land (G_2 \lor A_3) \land \text{T-solver(}\mu\text{) = Inconsistent}$
2. $\mu^P = G_1 \land G_2 \quad \varphi^P = \text{True} \land \text{T-solver(}\mu\text{) = Inconsistent}$
3. $\mu^P = G_1 \land \neg G_2 \quad \varphi^P = (\neg G_2 \lor T_3) \land (\neg G_2 \lor A_2) \land (G_2 \lor T_4) \land (G_2 \lor A_3) \land \text{Inconsistent}$
4. $\mu^P = G_1 \land G_2 \quad \varphi^P = \text{True} \land \text{T-solver(}\mu\text{) = Consistent}$

The DPLL algorithm tries to build a model using three main operations: decide, propagate, and backtrack [9]. The operation decide heuristically chooses a literal $l$ (which is an unassigned Boolean atom or its negation) for branching. The operation propagate then removes all the clauses containing $l$, and deletes all occurrences of $\neg l$ in the formula; this procedure is also called Boolean Constraint Propagation (BCP). If after deleting a literal from a clause, the clause only has only one literal left (unit clause), BCP assigns this literal to True. If deleting a literal from a clause results in an empty clause, this is called a conflict. In this case, the DPLL procedure must backtrack and try a different branch value.

At step 1, $G_1$ is decided to be the branching literal, and the $T$-solver validates that $(x_0 > 5)$ is consistent. BCP removes the clause $(G_1 \lor T_2)$, and deletes all occurrences of $\neg G_1$. This results in two unit clauses $T_1$ and $A_1$, so they are assigned to True, which means $\mu^P = G_1 \land T_1 \land A_1$. Similarly, at step 2 $G_2$ is chosen, i.e. $\mu^P = G_1 \land T_1 \land A_1 \land G_2$. The $T$-solver checks the conjunction: $\mu = (x_0 > 5) \land T_1 \land (x_1 < 3) \land (x_1 = x_0 + 1)$. This is obviously inconsistent, thus DPLL($T$) backtracks and tries $\neg G_2$, which leads to a consistent model.

Note that DPLL($T$) refers to various procedures integrating DPLL and a $T$-solver. There are procedures with an integration schema different from what we have described here. The interested reader is pointed to [15] for further references.
Symbolic Execution as DPLL Modulo Theories

4 Symbolic Execution as DPLL(\(T\))

Intuitively, a program can be encoded into a (first-order) formula whose models correspond to program traces. Symbolic Executors explore all program traces w.r.t. the set of program conditions, therefore they can be viewed as SMT solvers that return all (partial) models w.r.t. a set of Boolean atoms.

In this paper we only consider bounded programs, since this is the class of programs that SE can analyse. This means every loop can be unwound into a sequence of \(if\) statements. In order to encode a program into a formula, all program variables are renamed in the manner of Static Single Assignment form [6]: each variable is assigned exactly once, and it is renamed into a new variable when being reassigned. In this way, assignments such as \(x = x + 1\) will not be encoded into an unsatisfiable atomic formula. Under these settings, a program \(P\) can be modelled by a Symbolic Transition System (STS) as follows:

\[
P \equiv (S, I, G, A, T)
\]

\(S\) is the set of program states, \(I \subseteq S\) is the set of initial states; each state in the STS models the computer memory at a program point. \(G\) is the set of guards and \(A\) is the set of actions; guards and actions are first-order formulas. An action models the effect of an instruction on the computer memory. Actions that do not update the computer memory (e.g. conditional jumps) are Boolean atoms, the others are \(T\)-atoms. \(T \subseteq S \times G \times A \times S\) is the transition function, \(t_{ij} = (s_i, g_{ij}, a_{ij}, s_j) \in T\) models a transition from state \(s_i\) to state \(s_j\) by taking action \(a_{ij}\) under the guard \(g_{ij}\). After a transition \(t_{ij}: s_i \rightarrow s_j\), the state \(s_j\) is exactly as the state \(s_i\) apart from the variable updated by the action \(a_{ij}\).

One way to encode a transition \(t_{ij}\) into a first-order formula is to present it in the form: \(t_{ij} \equiv g_{ij} \rightarrow a_{ij}\), or equally \(t_{ij} \equiv \neg g_{ij} \lor a_{ij}\). This encoding expresses that satisfying the guard \(g_{ij}\) implies that the action \(a_{ij}\) is performed. In this way, a program trace is defined as a sequence of transitions:

\[
t_{01} \land t_{12} \land \cdots \land t_{(k-1)k} = (\neg g_{01} \lor a_{01}) \land (\neg g_{12} \lor a_{12}) \cdots \land (\neg g_{(k-1)k} \lor a_{(k-1)k})
\]

The semantics of the program is then defined as the set of all possible traces, or equally the set of all possible transitions, which can be represented as the following formula:

\[
\varphi = \bigwedge_{t_{ij} \in T} t_{ij} = \bigwedge_{t_{ij} \in T} (\neg g_{ij} \lor a_{ij}) \quad (4)
\]

Fig. 3 depicts a simple example program and its associated STS. Encoding this STS following (4) results in the formula (1) that we have illustrated with DPLL(\(T\)) in the previous section. We now illustrate this example with SE.

At a high level, a Symbolic Executor can be considered as the integration of two components: a Boolean Executor (BE) to execute the instructions and a \(T\)-solver to check the feasibility of path conditions. For example, SPF has a parameter \texttt{symbolic.dp} to customize which decision procedure to use. If we set this parameter with the option \texttt{no_solver} then SPF solely works on the BE.

A BE can be described as trying to build all path conditions using three main operations: \texttt{decide}, \texttt{update} and \texttt{backtrack}. The operation \texttt{decide} chooses a literal \(l\), a condition (or its negation) of an \(if\) statement, for branching, adding it to the path condition. The operation \texttt{update} then symbolically executes a block of statements, i.e. no branching statement presents, updating the computer memory. When the BE reaches the end of a symbolic path,
While both Symbolic Executors and SMT solvers implement DPLL(T), the main difference between them is the models they return. Symbolic Executors explore all path conditions, each path condition is identified by the values of the Boolean abstractions of its condition constraints. For instance, although there are many values of $x_0$ satisfying the path $\langle x_0 < 5 \rangle \land \neg (x_1 < 3)$, it is identified by $G_1 = \text{true}$ and $G_2 = \text{false}$. On the other hand, SMT solvers in general stop searching when a model is found. However, the solver MathSAT provides a functionality, called All-SMT [2], that given a formula and a set of Boolean atoms, it returns all models of the formula w.r.t. this set. Hence, by asking MathSAT to find all models w.r.t. the set of guards, we can explore all symbolic paths of the program.

We illustrate the approach with our running example in Fig. 3. The formula $\varphi$ in (1) is expressed in SMT-LIB v2 format [1] as follows (for the limit of space variable declarations are omitted):

\begin{verbatim}
void test(int x, int y){
   if(x > 5){
      x++;
      if (x < 3)
         x--;
      else
         y = x + 1;
   }
}
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{fig3.png}
\caption{A simple program and its associated STS. The first if statement is modelled by two transitions $\langle s_0, (x_0 > 5), T_1, s_1 \rangle$ and $\langle s_0, \neg (x_0 > 5), T_2, s_2 \rangle$; the assignment $x++$ is modelled by $\langle s_1, (x_0 > 5), x_1 = x_0 + 1, s_3 \rangle$; similarly for the rest of the program.}
\end{figure}

\section{A lightweight approach for Symbolic Execution using All-SMT}

We consider DPLL in its simplest form, without non-chronological backtracking.

\begin{verbatim}
1 (assert (= (> x0 5) G1))
2 (assert (= (< x1 3) G2))
3 (assert (= (= x1 (+ x0 1)) A1))
4 (assert (= (= x2 (- x1 1)) A2))
5 (assert (= (= y1 (+ x1 1)) A3))
6 (assert (or (not G1) T1))
7 (assert (or G1 T2))
8 (assert (or (not G1) A1))
9 (assert (or (not G2) T3))
10 (assert (or (not G2) A2))
11 (assert (or G2 T4))
12 (assert (or G2 A3))
13 (check-allsat (G1 G2))
\end{verbatim}
Recall that $\varphi = BA \land \varphi^P$ as defined in (2) and (3). Assertions from 1 to 5 are for $BA$, and each assertion from 6 to 12 represents a clause of $\varphi^P$. The final command, `check-allsat`, asks the solver to return all models w.r.t. the set $(G_1, G_2)$. Note that `check-allsat` is not included in standard SMT-LIB, and it is not supported by other solvers, e.g. Z3 [8].

Executing MathSAT on the example above, we obtain three models w.r.t. the set $(G_1, G_2)$ as follows: (True, False), (False, True) and (False, False), which correspond to two feasible symbolic paths: $(x_0 > 5) \land \neg(x_1 < 3)$ and $\neg(x_0 > 5)$. As we did not model integer overflow rules, the path $(G_1, G_2)$ as (True, True), i.e. $(x_0 > 5) \land (x_1 < 3)$, is not feasible (this path is feasible when the overflow in the assignment $x_1 = x_0 + 1$ changes a big integer into a small one). The path $\neg(x_0 > 5)$ is listed twice, since in this case the value of $G_2$, i.e. $(x_1 < 3)$, is irrelevant, but the solver still considered it in two cases True and False.

### 6 Discussions and Future Work

In this paper, we show the correspondence between a Symbolic Executor and an SMT solver implementing the DPLL(\text{T}) framework [15]. Therefore, the claims in this paper do not apply for the bit vector theory, since its solvers do not implement the DPLL(\text{T}) framework. Bit vector formulas are solved by flattening into propositional formulas, then the resulting formulas are checked by a SAT solver [10].

A common problem for both SE and SMT is that there is a considerable amount of redundancy in the queries to the $\text{T}$-solver. Because the BE and its counterpart in SMT, the SAT component, build the conjunctions of literals in an incremental manner. To address this problem, previous work in SE and SMT used different approaches. In SE, the BE often uses the $\text{T}$-solver as a black box, thus most research on constraint redundancy problem has focused on caching techniques. For example, KLEE [3] has a counterexample cache that maps of sets of constraints to counterexamples, Green [16] can store the constraints offline, reusing them in different runs of the Symbolic Executor. In SMT solver, there is a tight synergy between the SAT component and the $\text{T}$-solver, and both are incremental and backtrackable. Therefore, the $\text{T}$-solver do not start from scratch if the constraint is similar to the previous query.

An immediate direction for investigation is whether the caching techniques in KLEE and Green can improve the efficiency of SMT solvers. Conversely, it would be interesting to investigate if techniques developed for SMT can be exploited for SE, for example to develop a concurrent Symbolic Executor based on previous work in concurrent SMT solver [17].

### 7 Conclusion

We show the correspondence between Symbolic Execution and Satisfiability Modulo Theories. This correspondence is important, as it enables us to migrate techniques developed in one community to the other. Moreover, it can give an insight for future research. In our previous work, we proposed a DPLL-based algorithm to tackle the quantitative information flow problem, and implemented it using SPF [13, 12].

Based on the correspondence above, we propose a lightweight approach for SE, using the DPLL component of an SMT solver to explore symbolic paths. The main limitation of our approach is path redundancies, which is because the off-the-shelf All-SMT functionality is not tailored for SE. Hence, future work also includes extending the open-source SMT solver Z3 so that it can function as SE.
References