SMT for Termination of LLVM

Mohammad Touhidur Rahman
Supervision: Prof. Dr. Jürgen Giesl

SS 2017

Abstract

This paper will discuss the application of SMT solvers in the process of proving the termination of a C or LLVM program. For this proving, we need a *symbolic execution graph* which over-approximates all possible runs of a program. We will also need an abstract domain which is required for the construction of this *symbolic execution graph*. Several rules will be shown here for the construction of this graph. SMT solvers are required to check conditions for the applicability of these rules. An *integer transition system* is produced from this *symbolic execution graph*. This *integer transition system* can be used to prove termination by using standard techniques.

Keywords— LLVM, C programs, Termination, Symbolic Execution, SMT solver

1 Introduction

In this paper, we will be discussing the topic from [17]. We will present the concepts of [17] in a more simplified manner. It will also be less technical and very much shorter.

From [17], before diving into the process of proving the termination of C programs, let us first consider the standard C implementation of `strlen` [1, 8]. It is a program for computing the length of a string at the pointer `str`. Usually, strings are represented as a pointer `str` to the heap in C. This heap contains the allocated memory that consists of memory cells up to the first one containing the value 0 and it forms the value of the string.

```
int strlen(char* str) {char* s = str; while(*s) s++; return s-str;}
```

It is required to manage the interaction between addresses and the values they point to in order to analyze algorithms on such data. For proving the termination of C programs with low-level memory access, it is required to ensure the *memory safety*. The *memory safety* violation may lead to non-termination. The `strlen` algorithm ensures memory safety and terminates, because there exists an address `end` $\geq$ `str` (which is the integer property of `end` and `str`) such that `*end` is 0 (a pointer property of `end`) and all addresses `str` $\leq$ `s` $\leq$`end` are allocated. Other typical programs with pointer arithmetic operate on arrays (which are just sequences of memory cells in C). Based on [17], we will discuss the approach to prove memory safety and termination of algorithms on integers and pointers. In [17], an abstract domain is formulated based on *separation logic* [15] to track both integer properties which relate allocated memory addresses with each other, as well as pointer properties about the data stored at such addresses.

The platform-independent intermediate representation (IR) of the LLVM compilation framework [4, 13] is considered to analyze the programs. This will help us to avoid the complexities of C. The rest of the paper will be presented in the following sequential manner

- We will start with a short introduction of LLVM, SMT Solvers, and symbolic execution. These topics will be presented in the *Preliminaries*.
- Then, based on [17], we will discuss
  1. The formulation of the abstract domain.
  2. The rules for generating the *symbolic execution graph* using this abstract domain.
  3. The usage of SMT solvers on these rules. In this way, the applicability of these rules can be checked.
  4. The generation of *integer transition system* from this *symbolic execution graph*.

2 Preliminaries

Here, we will give a short introduction of terms like: *LLVM, SMT solvers and symbolic execution.*

2.1 LLVM (*Low-Level Virtual Machine*)

LLVM is a modern compilation framework. According to Wikipedia [5]: *The LLVM compiler infrastructure project (formerly Low Level Virtual Machine) is a "collection of modular and reusable compiler and toolchain technologies" [3] used to develop compiler front ends and back ends.*

One of the main features of LLVM is that it can act as middle layers of a complete compiler system, i.e., by taking the intermediate representation (IR) code from a compiler and using an optimizer to generate an optimized IR. This IR is our main concern here. It is the core of LLVM. It is an assembly language type programming language. IR is strongly typed and provides a RISC (*reduced instruction set computing*) instruction set. There are three different forms of IR: *in-memory compiler IR, on-disk bitcode representation and human readable assembly language representation.* We will be concerned with the last one [5, 4].

To say it from our paper's perspective, LLVM is capable to represent all high-level languages in its own assembly language like represention [4]. So, there is an LLVM representation of any C program which we will analyze for our purpose. As an example, consider a simple function for addition [6].

```
int addition(int x, int y) {
  return x + y;
}
```

Listing 1: C program

```
define i32 @additon(i32 %x, i32 %y) {
  entry: %tmp = add i32 %x, %y
         ret i32 %tmp }
```

Listing 2: LLVM equivalent

The LLVM equivalent of the addition function shows a little bit of syntax of LLVM IR. Here, i32 is a type for 32-bit integers. For the full reference of the syntax, see the LLVM Language Reference Manual.

2.2 SMT (*Sat Modulo Theories*) Solvers

Approaches for solving SMT problems are presented in [14]. From [14], in an SMT solver, we check the satisfiability of some formula based on some backgound theory.

Suppose there is some theory T which is a set of closed first-order formulas, and a formula F. An SMT solver will check the satisfiability of this formula F based on this theory T, i.e., whether F is *T-satisfiable* or not. There are two techniques of SMT solvers.

Eager SMT techniques: The formula F is translated into a propositional CNF formula by using a satisfiability-preserving transformation. Then a SAT solver comes into play to check the satisfiability of this CNF formula.

Lazy SMT techniques: Firstly a SAT solver is used to check the satisfiability of the Boolean Abstraction of the formula F, and then a theory solver is used to check the T-satisfiability of the corresponding sub-formulas that were abstracted away from F.

2.3 Symbolic Execution

Symbolic execution is the way of executing the program by assuming symbolic values instead of concrete values for the program variables. It finds all feasible executable paths of the program [11]. For example, let us take a look at the program in Listing 3 [7].

Here, we do not want this program to assert false. In case of a concrete execution, one would take concrete values of the variables a, b and c. Suppose, we take 2, 3 and 4 for a, b and c, respectively, and then run it along the program to see its assertion which is actually *true*. But, in case of *symbolic execution* we will take *symbolic values* of a, b and c. Suppose, we take α, β and γ symbolic values for the variables a, b and c. Figure 1 [7], shows the executed result.

```
void symbolic(int a, int b, int c) {
    int x = 0, y = 0, z = 0;
    if (a) { x = -2; }
    if (b < 5) {
        if (!a && c) { y = 1; }
        z = 2; }
    assert(x+y+z!=3) }
```

Listing 3: Simple C Program

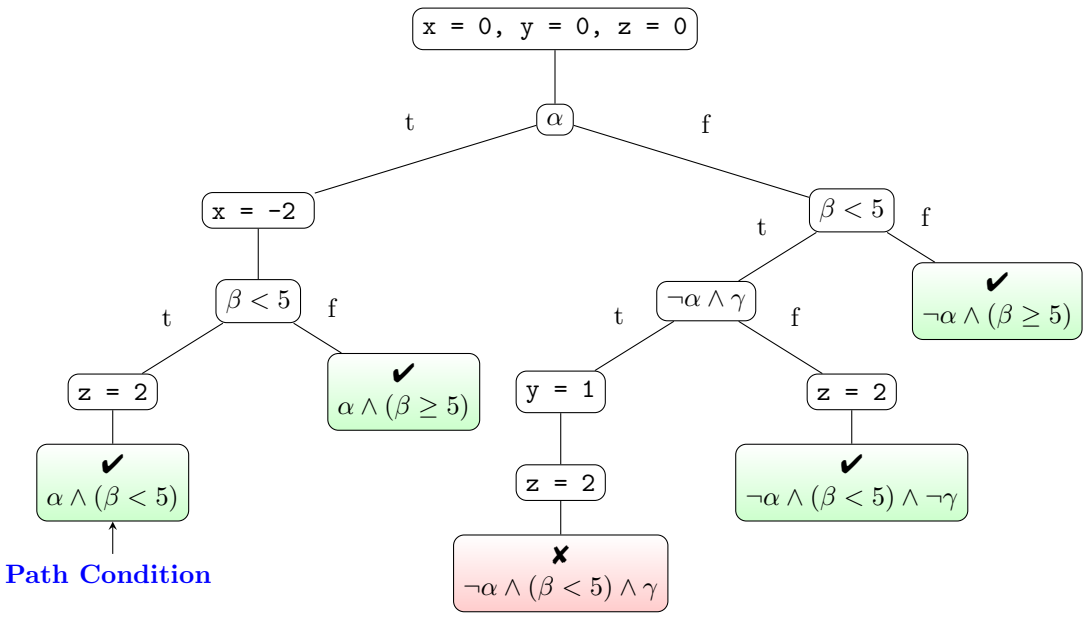

Figure 1: Symbolic Execution

Here, edges are representing *true* or *false* of the *if* condition. There is a path condition for each path which actually is one of the main concerns here. These path condition constraints are being checked using an SMT solver for satisfiability and ultimately we

can get all the feasible program paths. It is important to note that usually the *symbolic execution* creates an infinite tree for programs with loops. To avoid this, a *generalization* process is presented in [17]. This process tries to "identify" (or to generate) similar states and re-use previous states instead of always generating new ones. In this way, we obtain a finite *symbolic execution graph* instead of an infinite *symbolic execution tree*.

3 Abstract Domain for Symbolic Execution

This section will discuss the formulation of concrete LLVM states and *abstract* states that represent sets of concrete states based on [17].

Here, only types of the form `i`n (for n-bit integers), `i`n* (for pointers to values of type `i`n), `i`n**, `i`n***, etc are considered. Also the integer overflows are disregarded and it is assumed that variables are only instantiated with signed integers appropriate for their type. Let us take a look at the LLVM equivalent code in listing 4 for our previous `strlen` function. We will explain some of this LLVM code[1] during the construction of the *symbolic execution graph*.

An LLVM state consists of five components. These are *the program position (p), a partial function that maps local program variables to symbolic values (LV), the knowledge base (KB), the allocation list (AL) and a set of "points-to atoms" (PT).*

```
define i32 @strlen(i8* str) {
entry: 0: c0 = load i8* str
       1: c0zero = icmp eq i8 c0, 0
       2: br i1 c0zero, label done, label loop
loop:  0: olds = phi i8* [str,entry],[s,loop]
       1: s = getelementptr i8* olds, i32 1
       2: c = load i8* s
       3: czero = icmp eq i8 c, 0
       4: br i1 czero, label done, label loop
done:  0: sfin = phi i8* [str,entry],[s,loop]
       1: sfinint = ptrtoint i8* sfin to i32
       2: strint = ptrtoint i8* str to i32
       3: size = sub i32 sfinint, strint
       4: ret i32 size                    }
```

Listing 4: LLVM equivalent code for `strlen` function

The program position (p) is a pair (`b`, j) denoting the current *program position*. Here, `b` is the name of the current basic block and j is the index of the next instruction, for example, `entry, loop` and `done` are basic blocks in our `strlen` function.

The partial function for local variables (LV) gives information about current values of the local program variables. If V_p is the set of local program variables then $LV : V_p \rightharpoonup V_{sym}$ where "$\rightharpoonup$" denotes partial functions and V_{sym} is an infinite set of symbolic variables with V_{sym} and V_p having no common elements, i.e., $V_{sym} \cap V_p = \{\}$. For the construction of *symbolic execution*, this V_{sym} will be used.

The knowledge base (KB) is a set of quantifier-free first-order formulas. These formulas express integer arthmetic properties of V_{sym}. The knowledge base constrains V_{sym} to unique values for *concrete* states and to several possible values for *abstract* states.

The allocation list (AL) maintains the memory allocated by `malloc`. It contains expressions of the form $[\![v_1, v_2]\!]$ for $v_1, v_2 \in V_{sym}$ which indicates that all the addresses between v_1 and v_2 have been allocated.

A set of "points-to" atoms (PT) has atoms that look like $v_1 \hookrightarrow_{\texttt{ty}} v_2$, where v_1, v_2 $\in V_{sym}$ and `ty` is an LLVM type. It represents that address v_1 is pointing to the value v_2 of type `ty`.

[1]This LLVM program corresponds to the code obtained from strlen with the Clang compiler [2].To ease readability, variables are written without "%" in front (i.e., it is written "str" instead of "%str" as in proper LLVM) and added line numbers.

There is another special state called ERR to represent the error state. It is required for modeling possible memory safety violations which can happen through accessing memory that is not allocated.

Here, the mapping of LV is identified with the equations $\{\mathtt{x} = LV(\mathtt{x}) \mid \mathtt{x} \in V_p,\ LV(\mathtt{x})$ is defined$\}$ and LV is extended to a function from $V_p \uplus \mathbb{Z}$ to $V_{sym} \uplus \mathbb{Z}$ where $LV(n) = n$ for all $n \in \mathbb{Z}$. In a *well-formed* LLVM state, PT only stores information about allocated addresses. For example, the initial abstract state of `strlen` function will look like:

$$(\ (\mathtt{entry}, 0), \{\mathtt{str} = u_{\mathtt{str}}\}, \{z = 0\}, \{[\![u_{\mathtt{str}}, v_{end}]\!]\}, \{v_{end} \hookrightarrow_{\mathtt{i8}} z\}\ ) \qquad (\dagger)$$

If we compare it with (p, LV, KB, AL, PT) then p represents the begining of the `entry` block and $LV(\mathtt{str}) = u_{\mathtt{str}}$. AL contains $[\![u_{\mathtt{str}}, v_{end}]\!]$ because of an earlier call of `malloc` which allocates the memory on the heap where v_{end} represents the address of the last index. PT shows that v_{end} has a value z. That is why KB contains $\{z = 0\}$.

Two formulas are introduced for defining the semantics of abstract states: a first-order formula $\langle a \rangle_{FO}$ and a formula $\langle a \rangle_{SL}$ from a fragment of *separation logic* [15]. Here, $\langle a \rangle_{FO}$ stores the information about KB along with the consequences of AL and PT and represents all the addresses with positive numbers. SMT solvers are used to check the validity of statements about $\langle a \rangle_{FO}$ which is needed in order to apply symbolic execution rules. Below, we can see how $\langle a \rangle_{FO}$ will look like [17]:

Definition 1 (*Representing States by FO Formulas*) The set $\langle a \rangle_{FO}$ is the smallest set with

$$\begin{aligned}\langle a \rangle_{FO} = & KB \cup \{1 \leq v_1 \wedge v_1 \leq v_2 \mid [\![v_1, v_2]\!] \in AL\} \cup \\ & \{v_2 < w_1 \vee w_2 < v_1 \mid [\![v_1, v_2]\!], [\![w_1, w_2]\!] \in AL, (v_1, v_2) \neq (w_1, w_2)\} \cup \\ & \{v_2 = w_2 \mid (v_1 \hookrightarrow_{\mathtt{ty}} v_2), (w_1 \hookrightarrow_{\mathtt{ty}} w_2) \in PT \text{ and } \models \langle a \rangle_{FO} \Rightarrow v_1 = w_1\} \cup \\ & \{v_1 \neq w_1 \mid (v_1 \hookrightarrow_{\mathtt{ty}} v_2), (w_1 \hookrightarrow_{\mathtt{ty}} w_2) \in PT \text{ and } \models \langle a \rangle_{FO} \Rightarrow v_2 \neq w_2\} \cup \\ & \{v_1 > 0 \mid (v_1 \hookrightarrow_{\mathtt{ty}} v_2) \in PT\}.\end{aligned}$$

Based on this FO formula, *concrete states* are defined as abstract states of a particular form. In a *concrete state*, the memory contents are defined uniquely. Let us analyze our initial abstract state (†) for *concreteness*. From [17], (†) will be a concrete state iff (a) $\langle \dagger \rangle_{FO}$ is satisfiable, (b) all the allocated addresses (e.g., addresses from $u_{\mathtt{str}}$ to v_{end}) have unique contents, (c) due to byte-wise representation of memory data, only statement of the form $w_1 \hookrightarrow_{\mathtt{i8}} w_2$ in PT is allowed for concrete states and as LLVM stores values in two's complement, each byte will store a value from $[-2^7, 2^7 - 1]$, i.e., $v_{end} \hookrightarrow_{\mathtt{i8}} z$ is allowed and z has value from $[-2^7, 2^7 - 1]$, (d) and all the occurring symbolic variables, e.g., $u_{\mathtt{str}}$, v_{end} and z have unique values. Moreover, ERR is also a concrete state. A concrete state for (†) may look like the one below with arbitrary concrete values:

$$(\ (\mathtt{entry}, 0), \{\mathtt{str} = u_{\mathtt{str}}\}, \{z = 0, u_{\mathtt{str}} = 10, v_{end} = 20\}, \{[\![u_{\mathtt{str}}, v_{end}]\!]\}, \{v_{end} \hookrightarrow_{\mathtt{i8}} z\}\ )$$

Another important note is that, from [17], a state $a \neq ERR$ means that a is memory-safe, i.e., all the addresses in AL are allocated. Let $\rightarrow_{LLVM}$ be LLVM's evaluation relation on concrete states, i.e., $a \rightarrow_{LLVM} \overline{a}$ means that state $\overline{a}$ is reached after executing one LLVM instruction in state a. If $\rightarrow^+_{LLVM}$ is the transitive closure of $\rightarrow_{LLVM}$ then a state $a \neq ERR$ is *memory safe* iff there is no evaluation $a \rightarrow^+_{LLVM} ERR$.

The separation logic formula $\langle a \rangle_{SL}$ is an extension of $\langle a \rangle_{FO}$. Two partial functions are used for defining the semantics of separation logic. These are (as, mem) where as:

$V_P \rightharpoonup \mathbb{Z}$ represents the values of the program variables and mem: $\mathbb{N}_{>0} \rightharpoonup \{0,..., 2^8 - 1\}$ represents the values of the heap at the allocated memory addresses as unsigned bytes. For handling the symbolic variables in formulas, *instantiations* are used. An *instantiation* is a function $\sigma : V_{sym} \to \mathcal{T}(V_{sym})$ where $\mathcal{T}(V_{sym})$ is a set of all arithmetic terms. The formula $\langle a \rangle_{SL}$ is used to define the concrete states represented by an abstract state a. In [17], $\langle a \rangle_{SL}$ is formally defined along with the semantics of separation logic.

Now, for every *concrete state* $c \neq ERR$ one can infer an interpretation (as^c, mem^c). Then it can be defined that all concrete states c where (as^c, mem^c) is a model of some (concrete) instantiation of state a are represented by a (well-formed) abstract state a. Now, it can be said that our abstract state (†) of the `strlen` program represents all concrete states $c =$ ((`entry, 0`), LV, KB, AL, PT) where mem^c stores a string at the address as^c(`str`).[2] [17]

4 Construction of the Symbolic Execution Graph and Usage of SMT Solvers

Based on [17], a couple of symbolic execution rules for LLVM instructions will be discussed along with the usage of SMT solvers to check the applicability of these symbolic execution rules.

4.1 Symbolic Execution Rules

Here, the analysis starts from the abstract state (†). The symbolic execution graph for the `strlen` program is shown in Fig. 2. For good visual representation and reducing redundancy, some modifications are made to some symbols we discussed earlier. These are: the allocation list $AL = \{[\![u_{\texttt{str}}, v_{end}]\!]\}$ is removed as it remains the same for all states, some parts of LV, KB, and PT are expressed with "...". For $v_{end} \hookrightarrow_{\texttt{i8}} z$ and $z = 0$, $v_{end} \hookrightarrow_{\texttt{i8}} 0$ is written, etc.

The `strlen` program starts with loading the character at address `str` to `c0`. For defining the rules, *p:ins* denotes that *ins* represents the *instruction* at position p. In the rules, $\langle a \rangle$ is used instead of $\langle a \rangle_{FO}$. Let us now take a look at the first rule.

`load` from allocated memory (*p*: "`x = load ty* ad`" with `x, ad` $\in V_p$)

$$\frac{(p, LV, KB, AL, PT)}{(p^+, LV[\texttt{x} := w], KB, AL, PT \cup \{LV(\texttt{ad}) \hookrightarrow_{\texttt{ty}} w\})} \quad \text{if}$$

- there is $[\![v_1, v_2]\!] \in AL$ with $\models$ ($\langle a \rangle \Rightarrow (v_1 \leq LV_1(\texttt{ad}) \wedge LV_1(\texttt{ad}) + size(\texttt{ty}) - 1 \leq v_2)$) where this formula $\langle a \rangle \Rightarrow$ (....) is checked for validity by an SMT solver,
- $w \in V_{sym}$ is fresh

Here, the `load` rule is defined with the form p: "`x = load ty* ad`". It represents that the value of type `ty` at the address `ad` is assigned to the variable `x`. Firstly, it is checked that the addresses `ad,....,ad +` $size$`(ty) - 1` are allocated which is our first condition here. The 2nd condition is for the newly derived state. In the new state, the program position gets updated , i.e., if the progam was in position $p =$ (`b,` i), it moves to $p^+ =$ (`b,` i `+ 1`) of the same basic block. A new symbolic variable is introduced using LV(`x`) $= w$ where w is fresh. In the rule, it is written as LV[`x` $:= w$]. So, LV[`x` $:= w$](`x`)

[2] The reason is that then there is an address $end \in \mathbb{N}_{>0}$ with $end \geq as^c$(`str`) such that $mem^c(end) = 0$ and mem^c is defined for all numbers between as^c(`str`) and end. Hence if a is the state in (†), then $mem^c \models \sigma(\langle a \rangle_{SL})$ holds for any instantiation σ with $\sigma(u_{\texttt{str}}) = as^c(\texttt{str})$, $\sigma(v_{end}) = end$, and $\sigma(z) = 0$.

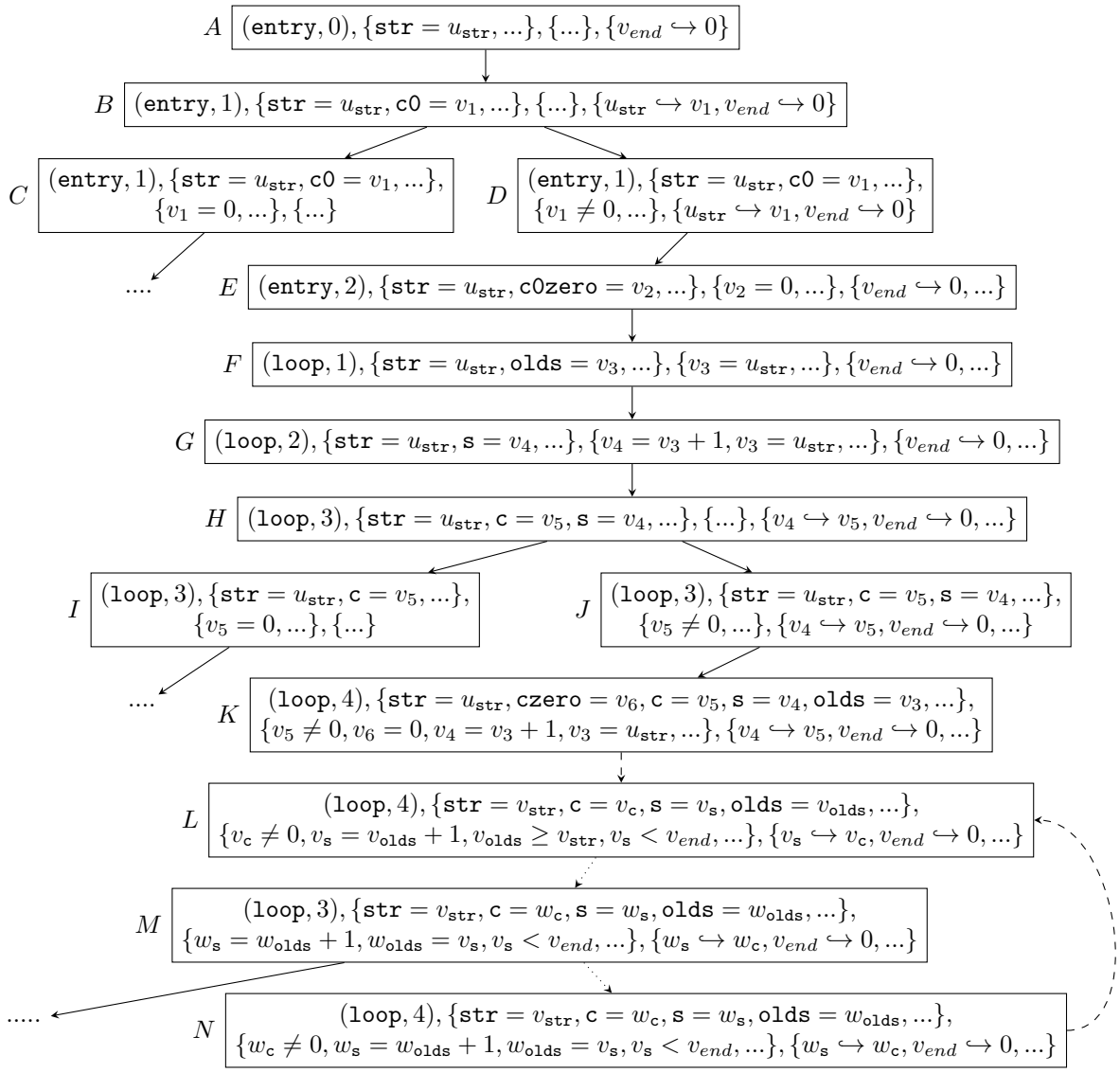

Figure 2: Symbolic Execution Graph of `strlen`

$= w$ and for $\mathtt{y} \neq \mathtt{x}$, it is $LV[\mathtt{x} := w](\mathtt{y}) = LV(\mathtt{y})$. Also PT is updated with $LV(\mathtt{add}) \hookrightarrow_{\mathtt{ty}} w$. If the formula $LV(\mathtt{add}) \hookrightarrow_{\mathtt{ty}} u$ is already in PT then $\langle a \rangle$ implies $w = u$, where a is the state before the execution step, i.e., above the horizontal line. For example, if we take a look at B in Fig. 2, after applying this `load` rule, the program position is updated, a new symbolic variable v_1 is added by using $LV[\mathtt{c0} := v_1](\mathtt{c0}) = v_1$ and PT is updated with new expression $u_{\mathtt{str}} \hookrightarrow v_1$.

Similarly, rules for the `store` instruction can be formalized. See [17] for the formalized rule. When `load` tries to access non-allocated address, then the *ERR* state is reached.

`load` from unallocated memory (*p*: "`x = load ty* ad`" with `x, ad` $\in V_p$)

$$\frac{(p, LV, KB, AL, PT)}{ERR} \quad \text{if}$$

- there is no $[\![v_1, v_2]\!] \in AL$ with $\models (\ \langle a \rangle \Rightarrow (v_1 \leq LV(\mathtt{ad}) \wedge LV(\mathtt{ad}) + size(\mathtt{ty}) - 1 \leq v_2)\ )$ where this formula $\langle a \rangle \Rightarrow (....)$ is checked for validity by an SMT solver

The next instructions in the `strlen` program are `icmp and br`. Both of them check if the first character `c0` is 0. If it evaluates to *true*, then we are at the end of the string and the program jumps to the block `done`. Now, the rule for integer comparison is introduced. For "`x = icmp eq ty` t_1, t_2", it is checked if the state has enough knowledge to decide whether the values t_1, t_2 of type `ty` are equal. Here, value 1 and 0 for *true* and $false$ repectively, is assigned to `x`.

`icmp eq` (***p*: "`x = icmp eq ty` t_1, t_2" with** `x` $\in V_p$ **and** $t_1, t_2 \in V_p \cup \mathbb{Z}$ **)**

$$\frac{(p, LV, KB, AL, PT)}{(p^+, LV[\texttt{x} := w], KB \cup \{w = 1\}, AL, PT)} \quad \text{if } \models (\ \langle a \rangle \Rightarrow (LV(t_1) = LV(t_2))\ ) \text{ and } w \in V_{sym} \text{ is fresh}$$

$$\frac{(p, LV, KB, AL, PT)}{(p^+, LV[\texttt{x} := w], KB \cup \{w = 0\}, AL, PT)} \quad \text{if } \models (\ \langle a \rangle \Rightarrow (LV(t_1) \neq LV(t_2))\ ) \text{ and } w \in V_{sym} \text{ is fresh}$$

here ($\langle a \rangle \Rightarrow (LV(t_1) = LV(t_2))$) is checked for validity by an SMT solver

So KB needs to have enough information for the evaluation of the `icmp` condition. The abstract state needs to be refined if the information is not present. It is done by extending the knowledge base. So, the following rule is defined, which transforms an abstract state into *two* new ones [17] [3]

refining abstract states (*p*: "`x = icmp eq ty` t_1, t_2" with `x` $\in V_p$**,** $t_1, t_2 \in V_p \cup \mathbb{Z}$**)**

$$\frac{(p, LV, KB, AL, PT)}{(p, LV, KB \cup \{\varphi\}, AL, PT) \quad | \quad (p, LV, KB \cup \{\neg\varphi\}, AL, PT)} \quad \text{if } \not\models (\langle a \rangle \Rightarrow \varphi) \text{ and } \not\models (\langle a \rangle \Rightarrow \neg\varphi)$$

here φ is $LV(t_1) = LV(t_2)$ and $\langle a \rangle \Rightarrow \varphi$ is checked by an SMT solver

Now, if we take a look at B of Fig. 2 for the evaluation of "`c0zero = icmp eq i8 c0, 0`", we see that it is checked if the value of the `c0` character of the string `str` is 0. As the KB of state B does not have enough information, refinement is done, i.e., state B is refined to states C and D. The new information for D is `c0` $= v_1$ and $v_1 \neq 0$. The successor of C is not displayed here as it leads to a program end.

So we have got the image of how these rules look like. See [17] for all the other rules for `strlen` program.

Now, recall that we mentioned the *generalization* idea for programs with loops. Let us take a brief look on it from [17]. If the evaluation process reaches the program position (`b`, j) twice and the mappings of LV in the two states have the same domain, then the *generalization* of states is done to achieve *finite* symbolic execution graphs. We see dashed edges from state K to L and N to L in Fig. 2. These dashed edges represent *generalization*, i.e., both the states K and N are represented by the more general state L. Semantically, if $\models \langle a \rangle_{SL} \Rightarrow \mu(\langle a' \rangle_{SL})$ holds for some instantiation μ, then state a' is a generalization of a state a. So, the generalization rule [17] states that if $a = (p, LV, KB, AL, PT)$ and $a' = (p, LV', KB', AL', PT')$ are two states with a having an incoming evaluation edge[4] and $LV(\texttt{x}) = \mu(LV'(\texttt{x}))$, $\models \langle a \rangle \Rightarrow \mu(KB')$, $\mu(AL') \subseteq AL$ and $\mu(PT') \subseteq PT$ for some instantiation μ, then a' is a generalization of a. Here, an SMT solver is used on $\langle a \rangle \Rightarrow \mu(KB')$ for validity to check whether the knowledge in the more special state implies the knowledge in the more general state.

[3] Analogous refinement rules can also be used for other conditional LLVM instructions, e.g., conditional jumps with `br` or other cases of `icmp`.

[4] Evaluation edges are edges that are not refinement or generalization edges.

4.2 Usage of SMT Solvers

It is very clear from the above rules that we are getting FO formulas for conditions of these rules and these formulas only contain integers and first-order connectives. Therefore, these formulas can be checked using ordinary SMT solvers over the integers. These formulas are checked for validity. If φ is one of these formulas then we can give $\neg\varphi$ to an SMT solver and prove unsatisfiability of $\neg\varphi$ to prove validity of φ, i.e., ($\models \varphi$). If these conditions containing FO formulas hold, then the corresponding symbolic execution rules will be applicable. It can be clearly said that an SMT solver is the main engine for the applicability of these symbolic execution rules.

In [17], AProVE [10, 16] is used for the implementation. It uses the SMT solvers Yices [9] and Z3 [12] in the back-end.

5 Integer Transition Systems for Proving Termination

In our *symbolic execution graph*, the *ERR* state is not reached which proves that `strlen` function is memory safe. Now, for proving termination, from [17], an *integer transition system* is extracted from the symbolic execution graph. There are already existing tools like AProVE which can be used to prove its termination. The extraction step essentially restricts the information in abstract states to the integer constraints on symbolic variables. It is often enough for the termination proof to use this conversion of memory-based arguments into integer arguments.

In the `strlen` program, the termination is proved by showing that `s` is increased as long as it is smaller than v_{end}, which is the end of the string.

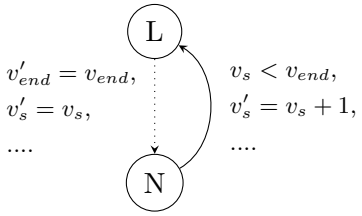

Figure 3: ITS of the cycle in Fig. 2

An ITS is a graph containing nodes and edges where nodes represent the abstract states and edges are *transitions*. Symbolic execution graphs are converted to ITSs by transforming every edge into a *transition*. For example, Fig. 3 shows the ITS of the cycle within states L and N of Fig. 2. Take a look at [17] for technical details on how a symbolic execution graph is converted to ITS. There is a theorem [17] on termination of LLVM Programs. The theorem states:

Theorem 1 (Termination of LLVM Programs) *Let $\mathcal{P}$ be an LLVM program with a complete symbolic execution graph $\mathcal{G}$. If $\mathcal{I}_\mathcal{G}$(the ITS from $\mathcal{G}$) is terminating, then $\mathcal{P}$ is also terminating for all LLVM states represented by the states in $\mathcal{G}$.*

6 Conclusion

This paper is fully based on [17]. In the current paper, the importance of SMT solvers for proving the termination of LLVM programs is discussed. For the usage of SMT solvers, the symbolic execution rules are formalized where SMT solvers check the applicability of these rules. So, the main focus of this paper was to discuss the formulation of these symbolic execution rules. In the final section, we discussed little bit about the termination based on *integer transition system* which can be generated from the *symbolic exeution graph*. It can be said that ordinary SMT solvers on integer arithmetic are enough to analyze the termination and memory safety of programs over the heap which is really remarkable.

References

[1] http://fxr.watson.org/fxr/source/lib/libsa/strlen.c?v=OPENBSD accessed on 2017-06-27.

[2] Clang compiler. http://clang.llvm.org accessed on 2017-06-27.

[3] The LLVM compiler infrastructure project. retrieved 2016-03-11.

[4] LLVM reference manual. http://llvm.org/docs/LangRef.html.

[5] LLVM wiki. https://en.wikipedia.org/wiki/LLVM.

[6] Simple LLVM example:. http://releases.llvm.org/2.6/docs/tutorial/JITTutorial1.html.

[7] Symbolic execution example. https://www.cs.umd.edu/class/fall2011/cmsc631/lectures/sym.pdf accessed on 2017-06-23.

[8] Wikibooks C Programming. https://en.wikibooks.org/wiki/C_Programming/ accessed on 2017-06-27.

[9] B. Dutertre, L. de Moura. The Yices SMT solver. Tool paper at. at http://yices.csl.sri.com/tool-paper.pdf accessed on 2017-06-27.

[10] J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving termination of programs automatically with AProVE. In: Proceedings of IJCAR'14.

[11] J. C. King and IBM Thomas J. Watson Research Center and Yorktown Heights, NY. Symbolic execution and program testing. 7, July 1976.

[12] L. de Moura, N. Bjørner. Z3: An efficient SMT solver. . In: Proceedings of TACAS'08.

[13] C. Lattner and V. Adve. LLVM: A compilation framework for life long program analysis & transformation. In: Proceedings of CGO'04.

[14] R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *J. ACM, 53(6)*, 2006.

[15] P. O'Hearn, J. Reynolds, and H. Yang. Local Reasoning about Programs that Alter Data Structures. In: Proceedings of CSL'01.

[16] T. Ströder, C. Aschermann, F. Frohn, J. Hensel, and J. Giesl. AProVE: Termination andmemory safety of C programs (competition contribution). In: Proceedings of TACAS'15.

[17] T. Ströder, J. Giesl, M. Brockschmidt, F. Frohn, C. Fuhs, J. Hensel, P. Schneider-Kamp, and C. Aschermann. Automatically proving termination and memory safety for programs with pointer arithmetic. *Journal of Automated Reasoning*, 58(1):33-65, 2017.