From Program to Logic: An Introduction

Patrice Godefroid and Shuvendu K. Lahiri

Microsoft Research, Redmond, WA, USA

Abstract. We review, compare and discuss several approaches for representing programs by logic formulas, such as symbolic model checking, bounded model checking, verification-condition generation, and symbolic-execution-based test generation.

1 Introduction

The goal of this paper is to provide a short tutorial to several approaches for reasoning about programs using logic. This idea, first pioneered in the 60s by Floyd [14] and Hoare [22], is by now well established. Over the last two decades, various approaches have been developed and refined for encoding programs (or parts of programs) into logic formulas. In this paper, we review and compare four of those approaches:

- *verification-condition generation* [10],
- *symbolic model checking* [4],
- *bounded model checking* [6], and
- *test generation using dynamic symbolic execution* [18].

These four approaches have a lot in common, yet differ by some important details. The main contribution of this paper is to discuss those commonalities and differences.

For this purpose, we consider a simple example of program, and illustrate how each of those approaches encode it using logic. Our running example is inspired from a similar example in [8], and is shown in Figure 1 in a C-like syntax. This program is composed of a single procedure `Foo` which takes as input arguments two bounded integers `a` and `b`, and a boolean value `z`. Procedure `Foo` uses two local integer variables `x` and `y`, and contains an assertion `assert(x <= 3)` which triggers an error whenever it is violated.

The goal of *program verification* is to prove that the assertion is *never* violated *for all* possible program executions, that is, for all possible input values of the program since this program is deterministic. Dually, the goal of test generation, or *bug finding*, is to prove that *there exists* a program execution where the assertion is violated, that is, that there exists an input vector (a set of input values) that triggers an assertion violation.

In other words, "program verification" often refers to checking a *universal* program property of the form *"for all path..."*, while "bug finding" often refers to

```
procedure Foo(a: int, b:int, z:bool)
{
   var x:int;
   var y:int;

0: x := a;
1: y := b;

2: x := x + y;
3: if (x != 1) {
4:    x := 2;
5:    if (z) {
6:       x := x + 1;
      }
   }
7: assert(x <= 3); // error
8:
}
```

Fig. 1. A simple program example.

checking an *existential* program property of the form *"for some path..."*. Program verification and testing are thus dual of each other.

We now describe how the four approaches considered in this paper deal with the simple example of Figure 1. We discuss each approach one by one.

2 Verification-Condition Generation

In this section, we survey some of the work on verification-condition (VC) generation. Informally, VC generation refers to the process of transforming a *bounded program* to a logical formula, where the (partial) correctness of the program is reduced to checking the validity of the resulting formula. A bounded program is a program without loops and procedure calls, and may contain assertions and assumptions. Loops can either be under-approximated by unrolling them a bounded number of times, or over-approximated by *"scrambling"*, i.e., assigning a nondeterministic value to, all the variables that the loop modifies and using a potential loop invariant. Procedure calls may similarly be under-approximated by inlining their bodies, or over-approximated by scrambling all the variables that the procedure may modify and using a potential procedure summary (expressed in terms of preconditions and postconditions). We postpone a discussion of loop invariants and postconditions to Section 6. It suffices to note that checking (unlike inferring) loop invariants and procedure specifications can be reduced to checking the correctness of a bounded program.

Dijkstra [10] proposed a VC generation algorithm based on the concept of *weakest liberal precondition* transformer. One shortcoming of the approach is that

```
// Passive form
procedure Foo(a: int, b: int, z: bool)
{
  A:      assume x@0 == a + b; goto Then1, Else1;
  Then1:  assume x@0 != 1; assume x@1 == 2; goto Then2, Else2;
  Then2:  assume z; assume  x@2 == x@1 + 1; goto B;
  Else2:  assume !z; assume x@2 == x@1; goto B;
  Else1:  assume x@0 == 1; assume x@2 == x@0; goto B;
  B:      assert x@2 <= 3; return;
}

// Block equations
BE_A     : OK_A     = ((x@0 == a + b) ==> (OK_Then1 && OK_Else1))
BE_Then1 : OK_Then1 = ((x@0 != 1) ==> ((x@1 == 2) ==> (OK_Then2 && OK_Else2)))
BE_Then2 : OK_Then2 = (z  ==> ((x@2 == x@1 + 1) ==> OK_B))
BE_Else2 : OK_Else2 = (!z ==> ((x@2 == x@1) ==> OK_B))
BE_Else1 : OK_Else1 = ((x@0 == 1) ==>  ((x@2 == x@0) ==> OK_B))
BE_B     : OK_B     = (True ==> ((x@2 <= 3) && True))

// VC
(BE_Else1 && BE_B && BE_Then1 && BE_Else2 && BE_Then2 && BE_A) ==> OK_A
```

Fig. 2. The VC generated using Boogie.

it can generate VCs exponential in the size of the input program; such cases happen due to repeated substitutions from assignment statements, and presence of conditional statements. Later approaches [13, 8, 3] avoid the exponential blowup by first transforming a program into a variant of the Single Static Assignment (SSA) form [9]. VC generation on the resulting program results in a formula whose size is linear in the size of the SSA form.

In the rest of this section, we describe the VC generation approach used in the Boogie program verifier [3]. This is in many ways a successor to the VC generation algorithm of ESC/Java [13, 12]. However, the algorithm used in Boogie can deal with unstructured programs (programs with arbitrary uses of **goto**), whereas the algorithm in [13] is restricted to structured programs.

The VC generation proceeds in several steps. The first step is to compile away structured control flow statements such as **while** and **if** using **assume** and **goto** `L1, L2, .., Lk` statements (that jumps nondeterministically to any one of the k labels).

This is followed by a *passification* step, which translates the program into an SSA form where assignments are converted into **assume** statements. Specifically, the passification step (a) renames variables (`x@1, x@2` etc.) to ensure that each variable is assigned at most once along each path, and (b) replaces every assignment `x@k := y@j;` with an assume **assume** `x@k == y@j`. The passified form for the example in Figure 1 is shown as the top half of Figure 2. This translation may require additional **assume** statements at the join points (e.g. **assume** `x@2`

`== x@1` at `Else2` in Figure 2); this may lead to a worst-case quadratic blowup in the passification process in theory [13], but is rare in practice.

For the resulting passive program consisting of a set of blocks $\mathcal{B}$ (a sequence of **assert** and **assume** statements terminating with a **goto** statement), the method generates a *block equation* BE_b for each block b in the program. For a block $b \in \mathcal{B}$ consisting of statements s with successor blocks $\{b_1, .., b_k\}$, we introduce a boolean variable OK_b and generate the $BE_b \doteq OK_b \equiv wlp(s, \bigwedge_i OK_{b_i})$. Here wlp refers to Dijkstra's weakest liberal precondition operator that takes a pair of statement s and a formula ϕ. It returns a formula representing the largest set of states from which all successor states satisfy ϕ without failing an assertion inside s. It is defined as: (i) $wlp(\textbf{assume } P, \phi) \doteq P \Rightarrow \phi$, (ii) $wlp(\textbf{assert } P, \phi) \doteq P \wedge \phi$, and (iii) $wlp(s; t, \phi) \doteq wlp(s, wlp(t, \phi))$. Note that wlp is only applied to passive statements (**assert** and **assume**) and therefore does not lead to any exponential blowup. For each block b, OK_b is true if the program is in a state for which all executions starting at block b do not fail an assertion.

The final VC is $\left(\bigwedge_{b \in \mathcal{B}} BE_b\right) \Rightarrow OK_{b_0}$, where $b_0 \in \mathcal{B}$ is the block corresponding to the entry to the procedure. The validity of this formula implies that the bounded program does not violate any assertion. Note that the VC generation works directly on unstructured programs and the VC is linear in the size of the passified program.

3 Symbolic Model Checking

Traditional model checking [7] consists of checking whether a finite-state system satisfies a temporal logic property. This is traditionally done by exhaustively exploring the system's *state space*. The state space of a system is a directed graph where nodes correspond to states and edges correspond to transitions the system can execute to move from state to state. Traditional model checking algorithms build this state graph explicitly, one state at a time, using some search strategy (e.g., a depth-first search).

In [4], an alternative state-space exploration strategy is proposed where state-space exploration is performed *symbolically* using *sets of states* instead of individual states. The key to make this possible is to use efficient representations for sets of states, such as logic formulas or Boolean Decision Diagrams (BDDs).

At a high-level, *symbolic model checking* denotes the following general algorithm:

> Given a logic formula I representing a set of initial states and a logic formula T representing a set of target (bad) states, initially set $S := I$, and then repeat forever:
>
> if $S \wedge T$ is satisfiable, then return "error"
>
> if $Post(S, TR) \Rightarrow S$, then return "safe"
>
> $S := S \vee Post(S, TR)$

Here, $Post(S, TR)$ is defined as the formula $\exists s : S(s) \wedge TR(s, s')$, which represents the set $\{s' | \exists s : s \in S \wedge TR(s, s')\}$ of states reachable in one transition from any state in S, and where TR represents the transition relation of the system.

For the example of Figure 1, TR for procedure Foo could be defined as

$$
\begin{aligned}
&((pc = 0) \wedge (pc' = 1) \wedge (x' = a) \wedge (y' = y) \wedge (z' = z)) \\
\vee&((pc = 1) \wedge (pc' = 2) \wedge (y' = b) \wedge (x' = x) \wedge (z' = z)) \\
\vee&((pc = 2) \wedge (pc' = 3) \wedge (x' = x + y) \wedge (y' = y) \wedge (z' = z)) \\
\vee&((pc = 3) \wedge (pc' = 4) \wedge (x! = 1) \wedge (x' = x) \wedge (y' = y) \wedge (z' = z)) \\
\vee&((pc = 3) \wedge (pc' = 7) \wedge (x = 1) \wedge (x' = x) \wedge (y' = y) \wedge (z' = z)) \\
\vee&((pc = 4) \wedge (pc' = 5) \wedge (x' = 2) \wedge (y' = y) \wedge (z' = z)) \\
\vee&((pc = 5) \wedge (pc' = 6) \wedge (z = true) \wedge (x' = x) \wedge (y' = y) \wedge (z' = z)) \\
\vee&((pc = 5) \wedge (pc' = 7) \wedge (z = false) \wedge (x' = x) \wedge (y' = y) \wedge (z' = z)) \\
\vee&((pc = 6) \wedge (pc' = 7) \wedge (x' = x + 1) \wedge (y' = y) \wedge (z' = z)) \\
\vee&((pc = 7) \wedge (pc' = err) \wedge (x > 3) \wedge (x' = x) \wedge (y' = y) \wedge (z' = z)) \\
\vee&((pc = 7) \wedge (pc' = 8) \wedge (x <= 3) \wedge (x' = x) \wedge (y' = y) \wedge (z' = z))
\end{aligned}
$$

This logic encoding uses two logic variables for each program variable: one variable x for denoting the value of the program variable `x` *before* the transition and one primed variable x' to denote its value *after* the transition. It also uses one special variable pc to represent the *program counter*.

Given a set of initial states represented by the formula $true$ assuming the values of the input variables `a`, `b` and `z` are totally unconstrained, and given a set of target states represented by the formula $pc = err$ where err denotes the program location reached after the assertion on line 7 is violated, the symbolic model checking algorithm can be used to prove that the set of target states is unreachable. Each iteration step can be performed using an automated theorem prover for the logic used to encode I, T, TR and $Post$, until the algorithm converges to a fixpoint where the set S can no longer grow. Or the whole algorithm can be implemented using a theorem prover for that logic extended with fixpoint operators. Termination is guaranteed if the number of reachable states is finite.

Note that the logic needs to be closed under quantifier elimination in order to deal with the existential quantifier in the $Post$ formula. For the example of Figure 1, if `a` and `b` are bounded integers, then this condition holds and the search is guaranteed to terminate. Otherwise, overall results will depend on how existential quantification of (unbounded) integer variables is handled by the theorem prover.

4 Bounded Model Checking

Bounded Model Checking using SAT solvers is another approach for representing programs using logic formulas [6]. It was proposed and developed by researchers from the model checking community, following earlier work on symbolic model checking, but it is also strongly related to earlier work on verification-condition (VC) generation.

In this section, we discuss in more details the tool CBMC, a bounded model checker for ANSI C programs [8]. It works by creating a bounded program by unrolling loops and inlining procedures up to a constant bound. We focus on

the VC generation for the resulting bounded program. We focus on the essential aspects of the VC generation and abstract from particulars of modeling ANSI C programs such as modeling pointers, treating variables as fixed-width vectors, etc.

The VC generation proceeds by first converting an unstructured program into a structured program. The unstructured programs are slightly less general compared to the inputs to the Boogie VC generation, as this approach does not support a jump to the middle of a conditional block. A **goto** `L` statement is replaced by *predicating* all the instructions from the **goto** statement up to the label `L` with the negation of the condition that holds at the **goto** statement.

The resulting structured program only consists of assignments, **skip**, **if**, **assert** and sequential composition. The first step is to convert the program into an SSA form by renaming variables to ensure that each assignment writes to a fresh copy. Figure 3 shows the SSA form for the example in Figure 1. An interesting thing to note is that the resulting SSA program does not include the merge nodes that are introduced at the join points for SSA construction. Therefore the semantics of the SSA form is not the same as the input program. Instead, this form is only an intermediate form that is used for the VC generation.

The VC generation uses two symbolic transformers $\mathcal{C}(s, \phi)$ (to collect the assumptions) and $\mathcal{P}(s, \phi)$ (to collect the assertions), where s is a statement and ϕ is a formula. They are defined recursively as follows:

Statement (st)	$\mathcal{C}(st, \phi)$	$\mathcal{P}(st, \phi)$
if c $\{s\}$	$\mathcal{C}(s, \phi \wedge c)$	$\mathcal{P}(s, \phi \wedge c)$
$s; t$	$\mathcal{C}(s, \phi) \wedge \mathcal{C}(t, \phi)$	$\mathcal{P}(s, \phi) \wedge \mathcal{P}(t, \phi)$
assert c	**true**	$\phi \Rightarrow c$
$v_\alpha := e$	$v_\alpha == (\phi \ ? \ e \ : \ v_{\alpha-1})$	**true**

The final VC that is checked is constructed as $\mathcal{C}(s, \mathbf{true}) \Rightarrow \mathcal{P}(s, \mathbf{true})$. The above definitions of $\mathcal{C}(s, \phi)$ and $\mathcal{P}(s, \phi)$ are fairly standard, except for the case of assignment. For the statement $v_\alpha := e$, v_α is a renaming of the program variable v in the SSA form. For such a renamed statement, $\mathcal{C}(s, \phi)$ denotes that v_α equals e (suitably renamed) when the predicate ϕ evaluates to **true**; otherwise, it remains unchanged and equals the value $v_{\alpha-1}$ that denotes the value before the assignment. This conditional equality allows the VC generation not to require the presence of additional **assume** statements at join nodes. Figure 3 illustrates the final VC generated by CBMC on the input program.

5 Test Generation Using Dynamic Symbolic Execution

In contrast with the previous three approaches, test generation using symbolic execution builds a logic representation of a program *incrementally*, path by path.

```
// SSA form
x@1 := x@0 + y@0;
if(x@1 != 1) {
  x@2 := 2;
  if(z@0) x@3 := x@2 + 1;
}
assert(x@3 <= 3);

// C(s,true)
C := (x@1 == x@0 + y@0) &&
      x@2 == ((x@1 != 1) ? 2 : x@1) &&
      x@3 == ((x@1 != 1 && z@0)? x@2 + 1: x@2)
//P(s,true)
P :=  x@3 <= 3

// VC
C ==> P
```

Fig. 3. The VC using CBMC.

Symbolic execution and test generation can be done statically [23], using static program analysis, or dynamically [24], while executing the program under test. In this paper, we focus the presentation on dynamic test generation since it subsumes static test generation [17], and is the most precise form of test generation known today.

Test generation using dynamic symbolic execution consists of executing the program, typically starting with some random inputs, while performing symbolic execution *dynamically*, collecting symbolic constraints on inputs gathered from predicates in branch statements along the execution, and then using a constraint solver to infer variants of the previous inputs in order to steer the next execution of the program towards an alternative program branch. The process is repeated with the goal of reaching a *specific target* program branch or statement.

Directed Automated Random Testing [18], or DART for short, is a recent variant of dynamic test generation that blends it with model checking techniques with the goal of systematically executing *all* feasible program paths of a program while detecting various types of errors using run-time checking tools (like Purify, for instance). In DART, each new input vector attempts to force the execution of the program through *some* new path, but is *not* guided by one specific target branch or statement. By repeating this process, such a systematic search attempts to force the program to sweep through *all* its feasible execution paths, in a style similar to *systematic testing* and *dynamic software model checking* [15]. Along each execution, a runtime checker (like Purify, Valgrind or AppVerifier) is used to detect various types of errors, such as memory safety violations. Symbolic execution is itself augmented to inject additional symbolic constraints that attempt to trigger specific errors, such as buffer overflows, in alternate program executions [19].

This approach has become quite popular during the last few years, and is sometimes referred to as "execution-generated tests" [5], "concolic testing" [26], or simply "dynamic symbolic execution" [27].

5.1 Constraint Generation with DART

Symbolic execution means executing a program with symbolic rather than concrete values. Assignment statements are represented as functions of their (symbolic) arguments, while conditional statements are expressed as constraints on symbolic values. For a program path w, a *path constraint* pc_w is a logic formula that characterizes the input values for which the program executes along w.

Side-by-side concrete and symbolic executions are performed using a concrete store M and a symbolic store S, which are mappings from *memory addresses* (where program variables are stored) to concrete and symbolic values respectively. A *symbolic value* is any expression e in some theory[1] $\mathcal{T}$ where all free variables are exclusively input parameters. For any program variable v, $M(v)$ denotes the *concrete value* of v in M, while $S(v)$ denotes the *symbolic value* of v in S. For notational convenience, we assume that $S(v)$ is always defined and is simply $M(v)$ by default if no symbolic expression in terms of inputs is associated with v in S. When $S(v)$ is different from $M(v)$, we say that that program variable v has a *symbolic value*, meaning that the value of program variable v is a function of some input(s) which is represented by the symbolic expression $S(v)$ associated with v in the symbolic store.

Consider the example of Figure 1 again. Assume we start with some initial concrete input values, say `a` is initially 5, `b` is initially 3 and `z` is *true*. Initially, every program input is associated with a symbolic variable, denoted respectively by a, b and z, and every program variable storing an input value has its symbolic value (in the symbolic store) associated with the symbolic variable for the corresponding input: thus, the symbolic value for program variable `a` is the symbolic value a, and so on. Initially, the path constraint is simply *true*.

Run 1. Consider a first program run where the input value `a` is 5, `b` is 3 and `z` is *true*. After the first assignment statement `x:=a;` is executed, the concrete value of `x` is 5 while its symbolic value is a. Similarly, after executing the assignment `x:=x+y;`, the concrete value of `x` is 8, while its symbolic value is $a + b$ (assuming the symbol + representing addition is part of the theory $\mathcal{T}$).

At the first conditional statement `x!=1`, since the concrete value of `x` is 8, the current execution takes the `then` branch, and the input constraint $a + b \neq 1$ is added to the current path constraint.

Next, after executing the statement `x:=2;`, the concrete value of variable `x` is 2 and this variable has no longer a symbolic value (since it has been overwritten by the non-input value 2). Next, we reach the second conditional statement `z==true`; since program variable `z` has a symbolic value, which is z, we add the constraint $z = true$ to the path constraint. The concrete and symbolic execution go on until control reaches the assertion. At that point, variable `x`

[1] A theory is a set of logic formulas.

has the concrete value 3 and no symbolic value. The assertion holds, and no constraint is added to the path constraint (since `x` has no symbolic value). The path constraint is at that point $(a + b \neq 1) \wedge (z = true)$.

Run 2. By negating the second constraint $z = true$ in the path constraint, we obtain the new path constraint $(a+b \neq 1) \wedge (z = false)$. By solving this new constraint, we get a new input vector where inputs a and b are unchanged but input z is now $false$. We re-run the program a second time with these new inputs while performing symbolic execution dynamically. During this second program execution, when we hit the assertion, the concrete value of variable `x` is 2 and it has no symbolic value again. No new path constraint can be generated off this execution and the DART algorithm moves on to considering the case below.

Run 3. By negating the first constraint $a+b \neq 1$ in the first path constraint, we obtain another new path constraint $a + b = 1$ which is satisfiable. Assume the constraint solver returns the satisfying assignment where $a = 1$ and $b = 0$. During this third program execution, when we hit the assertion, the concrete value of `x` is 1, so the assertion is satisfied. But this time, `x` has the symbolic value $a + b$, and we also add the constraint $a + b <= 3$ to the current path constraint, which is now $(a + b = 1) \wedge (a + b <= 3)$.

Then, we negate the last constraint of this new path constraint, and obtain the new alternate path constraint $(a+b = 1) \wedge (a+b > 3)$. This path constraint is unsatisfiable, and the whole search stops, thus after executing the program 3 times.

5.2 Compositional Variant

The DART algorithm will thus generate as many tests as they are feasible whole-program paths. In practice, this number can be very large, possibly exponential in the size of the program or even infinite in the presence of a single program loop whose number of iterations is determined by some unbounded input. This *path explosion* can be alleviated by performing symbolic execution and test generation *compositionally* [16].

In compositional symbolic execution, a function summary ϕ_f for a function (or block or whatever program sub-computation) f is defined as a logic formula over constraints expressed in theory $\mathcal{T}$. ϕ_f can be generated by symbolically executing each path of function f, then generating an input precondition and output postcondition for this path, and then bundling together all path summaries in a disjunction. More precisely, ϕ_f is defined as a *disjunction* of formulas ϕ_{w_f} of the form $\phi_{w_f} = pre_{w_f} \wedge post_{w_f}$, where w_f denotes an intraprocedural path inside f, pre_{w_f} is a conjunction of constraints on the inputs of f, and $post_{w_f}$ is a conjunction of constraints on the outputs of f. An input to a function f is any value that can be read by f, while an output of f is any value written by f. ϕ_{w_f} is computed automatically by symbolically executing the intraprocedural path w_f: pre_{w_f} is the path constraint along path w_f but expressed in terms of the function inputs, while $post_{w_f}$ is a conjunction of constraints each of the form $v' = S(v)$ where v' is a fresh symbolic variable created for each program variable v modified during the execution of w_f and where $S(v)$ denotes the symbolic value

associated with v in the program state reached at the end of w_f. At the end of the execution of w_f, the symbolic store is updated so that each such value $S(v)$ is replaced by v'. When symbolic execution continues after the function returns, such symbolic values v' are treated as inputs to the calling context. Summaries can be re-used across different calling contexts [16].

For the example of Figure 1, a summary ϕ_f for the outermost if-then-else block (ending just before the assertion) would be

$$\begin{array}{l}((x = 1) \wedge (x' = x)) \\ \vee((x \neq 1) \wedge (z = true) \wedge (x' = 3)) \\ \vee((x \neq 1) \wedge (z = false) \wedge (x' = 2))\end{array}$$

after logic simplification and where x' denotes a fresh symbolic variable associated to program variable `x` at the end of that block, which represents the (single) output from that block. Given such a summary ϕ_f, when the assertion is hit, a *compositional path constraint*

$$(x = a + b) \wedge \phi_f \wedge (x' <= 3)$$

would be generated, and negating the last constraint in the conjunction would be unsatisfiable, proving that the assertion cannot be violated.

In general, logic function summaries can involve summaries of lower-level functions, or be recursive. There are various ways to represent those logically (e.g., see [16, 1, 21]). Using summarization, compositional path constraints can be shown to be of size polynomial in the size of the program provided the program is loop-free or given a bound on the number of intraprocedural paths being summarized [16].

6 Dealing with Loops

Another fundamental difference between the approaches discussed in the previous sections is how they deal with *program loops*.

Verification-condition generation usually assumes that every program loop is annotated with a loop invariant, either provided by the user or automatically generated. These annotations are used to effectively transform the program into an (usually over-approximate) loop-free form.

For instance, consider the program fragment

```
...
var j:int;
j := 0;
while (i>0) { j++; i--; }
...
```

A precise loop invariant which holds at the beginning of the while loop could be

$$(i \geq 0) \wedge (j = i_0 - i)$$

where i_0 denotes the value of program variable `i` at the beginning of the execution of the loop, while i and j represent respectively the current value of variables `i` and `j` at the beginning of each loop iteration. When the loop terminates, we have $(i = 0) \wedge (j = i_0)$. To analyze the part of the program following the loop, the loop itself can be replaced, or *summarized* (see the previous section), by this postcondition.

A *weaker*, i.e., less precise, loop invariant could simply be $(i >= 0) \wedge (j >= 0)$. Such a weak invariant, where any program variable *modified* during some loop execution can have any value, can always be used for any loop. However, the set of program values satisfying this invariant is an *over-approximation* of the actual set of program states, and may be too imprecise for proving interesting program properties. How to generate automatically precise loop invariants has been discussed in numerous papers (e.g., [11, 25, 2, 28] to name a few).

In symbolic model checking, individual program statements are represented symbolically/logically one by one and the execution of loops is simulated using a fixpoint computation on this logic representation.

In contrast, bounded model checking unrolls loops up to some arbitrary depth (which can be iteratively increased if necessary).

Finally, the DART algorithm unrolls loops as determined by a concrete execution. For instance, in the example above, if i_0 is a whole-program input and is initially 10, the first program execution would execute the loop 10 times, would generate a first path constraint reflecting this, and successive alternate path constraints would then force the loop to be executed a different number of times. Techniques for automatic loop invariant generation and loop summarization can also be exploited in this context [20].

7 Dealing with Imprecision

The previous approaches also differ in how they deal with imprecision in symbolic execution. Indeed, in practice, symbolic execution of large complex programs is rarely fully precise due to external library or system calls, unhandled program instructions, pointer arithmetic, floating-point computations, etc.

In those cases, verification-condition generation, symbolic and bounded model checking would typically *over-approximate* program behaviors, building *may* abstractions suitable for program verification (of universal path properties). For instance, whenever the exact set of possible values of a program variable is unknown, this set is over-approximated, as performed by the "scrambling" operation discussed in Section 2.

In contrast, dynamic test generation typically *under-approximate* program behaviors, building *must* abstractions suitable for test generation (i.e., program verification of existential path properties). For instance, whenever the exact set of possible values of a program variable is unknown, this set is under-approximated, for instance using specific concrete values witnessed in some program execution.

If symbolic execution is precise and generates path constraints that are both sound and complete, static and dynamic test generation coincide. Moreover, the

DART algorithm can then be simply viewed as an iterative way to compute a logic program representation fairly similar to the ones generated by the three previous approaches discussed in this paper, *in the case of loop-free programs*. However, if symbolic execution is imprecise, then dynamic test generation can be shown to be more precise and powerful than static test generation [17]. Test generation is then also different from verification-condition generation, symbolic and bounded model checking.

Finally, bounded model checking also often performs may abstractions for external functions, while bounded loop unrolling is inherently a must abstraction if the loop bound is not large enough. In that case, bounded model checking is neither sound for program verification nor sound for bug finding.

8 Conclusion

The purpose of this paper is to provide a brief tutorial to several approaches for representing programs using logic, as well as to highlight commonalities and differences. We emphasize that there are many other variants of these approaches in the literature (e.g., work on automatic inference of loop and procedure specifications), and that this paper is by no means an exhaustive survey.

A second goal of this paper is to compare the recently developed approach of test generation using (dynamic) symbolic execution with prior work on representing programs using logic. As shown in this paper, this recent work is also closely related to past work on verification-condition generation, symbolic model checking and bounded model checking.

References

1. S. Anand, P. Godefroid, and N. Tillmann. Demand-Driven Compositional Symbolic Execution. In *Proceedings of TACAS'2008 (14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, volume 4963 of *Lecture Notes in Computer Science*, pages 367–381, Budapest, April 2008. Springer-Verlag.
2. T. Ball, O. Kupferman, and M. Sagiv. Leaping Loops in the Presence of Abstraction. In *Proceedings of CAV'2007 (19th Conference on Computer Aided Verification)*, Berlin, July 2007.
3. M. Barnett and K. R. M. Leino. Weakest Precondition of Unstructured Programs. In *Proceedings of PASTE'05 (Program Analysis For Software Tools and Engineering)*, pages 82–87, 2005.
4. J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of LICS'1990 (5th Symposium on Logic in Computer Science)*, pages 428–439, Philadelphia, June 1990.
5. C. Cadar and D. Engler. Execution Generated Test Cases: How to Make Systems Code Crash Itself. In *Proceedings of SPIN'2005 (12th International SPIN Workshop on Model Checking of Software)*, volume 3639 of *Lecture Notes in Computer Science*, San Francisco, August 2005. Springer-Verlag.
6. E. M. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded Model Checking Using Satisfiability Solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

7. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
8. E. M. Clarke, D. Kroening, and K. Yorav. Behavioral Consistency of C and Verilog Programs using Bounded Model Checking. In *Design Automation Conference (DAC)*, pages 368–371. ACM, 2003.
9. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 13(4):451–490, 1991.
10. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.
11. C. Flanagan and K. R. M. Leino. Houdini, an Annotation Assistant for ESC/Java. In *Proceedings of FME'2001 (Formal Methods Europe)*, pages 500–517, 2001.
12. C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *Proceedings of PLDI'2002 (ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation)*, pages 234–245, 2002.
13. C. Flanagan and J. B. Saxe. Avoiding Exponential Explosion: Generating Compact Verification Conditions. In *Proceedings of POPL'2001 (28th ACM Symposium on Principles of Programming Languages)*, pages 193–205, 2001.
14. R. Floyd. Assigning Meaning to Programs. In *Mathematical Aspects of Computer Science*, pages 19–32. XIX American Mathematical Society, 1967.
15. P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of POPL'97 (24th ACM Symposium on Principles of Programming Languages)*, pages 174–186, Paris, January 1997.
16. P. Godefroid. Compositional Dynamic Test Generation. In *Proceedings of POPL'2007 (34th ACM Symposium on Principles of Programming Languages)*, pages 47–54, Nice, January 2007.
17. P. Godefroid. Higher-Order Test Generation. In *Proceedings of PLDI'2011 (ACM SIGPLAN 2011 Conference on Programming Language Design and Implementation)*, pages 258–269, San Jose, June 2011.
18. P. Godefroid, N. Klarlund, and K. Sen. DART: Directed Automated Random Testing. In *Proceedings of PLDI'2005 (ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation)*, pages 213–223, Chicago, June 2005.
19. P. Godefroid, M. Levin, and D. Molnar. Active Property Checking. In *Proceedings of EMSOFT'2008 (8th Annual ACM & IEEE Conference on Embedded Software)*, pages 207–216, Atlanta, October 2008. ACM Press.
20. P. Godefroid and D. Luchaup. Automatic Partial Loop Summarization in Dynamic Test Generation. In *Proceedings of ISSTA'2011 (ACM SIGSOFT International Symposium on Software Testing and Analysis)*, pages 23–33, Toronto, July 2011.
21. P. Godefroid, A. Nori, S. Rajamani, and S. Tetali. Compositional May-Must Program Analysis: Unleashing The Power of Alternation. In *Proceedings of POPL'2010 (37th ACM Symposium on Principles of Programming Languages)*, pages 43–55, Madrid, January 2010.
22. C. A. R. Hoare. An Axiomatic Approach to Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.
23. J. C. King. Symbolic Execution and Program Testing. *Journal of the ACM*, 19(7):385–394, 1976.
24. B. Korel. A Dynamic Approach of Test Data Generation. In *IEEE Conference on Software Maintenance*, pages 311–317, San Diego, November 1990.

25. C. Pasareanu and W. Visser. Verification of Java Programs Using Symbolic Execution and Invariant Generation. In *Proceedings of SPIN'2004 (11th International SPIN Workshop on Model Checking of Software)*, volume 2989 of *Lecture Notes in Computer Science*, Barcelona, April 2004. Springer-Verlag.
26. K. Sen, D. Marinov, and G. Agha. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of FSE'2005 (13th International Symposium on the Foundations of Software Engineering)*, Lisbon, September 2005.
27. N. Tillmann and J. de Halleux. Pex - White Box Test Generation for .NET. In *Proceedings of TAP'2008 (2nd International Conference on Tests and Proofs)*, volume 4966 of *Lecture Notes in Computer Science*, pages 134–153. Springer-Verlag, April 2008.
28. A. Tsitovich, N. Sharygina, C. Wintersteiger, and D. Kroening. Loop Summarization and Termination Analysis. In *Proceedings of TACAS'2011 (17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems)*, April 2011.