Survey of Dafny and related tools

11.18

Weizhi Feng

Outline

- Dafny
- Boogie
- Smack
- Klee
- Our sample based tool

Dafny

实际工作中怎么验证程序写对了?

Mike He

I am; therefore I fail.

88 人赞同了该回答

有一个语言叫Dafny (F*的兄弟；OOP版本的F-star，语法和主流OOP语言相近)。这个语言提供一个叫做auto-active verifcation的功能，即：给出contract / invariant，然后将它们编译成Z3 SMT solver的constraint扔到Z3上跑SMT试图找到反例。这种验证较为适合在实际工作场景下使用。因为

1. 你不需要懂很多formal verification的东西(知道Hoare Logic和那几个rule就足够了)

2.有很多自动化的地方(例如automatic induction)

3. Several levels of indirection: 足够抽象，能够比较容易的将你想formalize的部分实现出来

Dafny

```
function method maxOf(s : seq<nat>): nat
decreases |s|
ensures forall i :: 0 <= i < |s| ==> s[i] <= maxOf(s)
{
    if |s| == 0 then 0 else max(s[0], maxOf(s[1..]))
}
```

Dafny

题外话：Dafny也可以当作proof assistant:

首先是喜闻乐见的Peano Number

```
datatype Nat = Z | S(Nat)

function add(x : Nat, y : Nat): Nat
decreases x
{
    match x
        case Z => y
        case S(n) => S(add(n, y))
}
```

尝试让Dafny验证加法交换律：

```
ghost method add_comm(x : Nat, y : Nat)
ensures add(x, y) == add(y, x) {}
```

```
ghost method add_comm(x : Nat, y : Nat)
ensures add(x, y) == add(y, x) {
    match x
        case Z => calc {
            add(Z, y); == y; == { addZ(y); } add(y, Z);
        }
        case S(x') => calc {
                add(S(x'), y);
            ==  S(add(x', y)); // definition
            ==  { add_comm(x',y); } // IH
                S(add(y, x'));
            ==  { addS(y, x'); }
                add(y, S(x'));
        }
}
```

Dafny

- Programming language
- Dafny relies on high-level annotations to reason about and prove correctness of code.
  - Lifts the burden of writing bug-free code into that of writing bug-free annotations.
- Language features drawn from:
  - Imperative programming: if, while, :=, class, …
  - Functional programming: function, datatype, …
  - Proof authoring: lemma, calc, refines, inductive predicate, …

命令式编程（Imperative） vs声明式编程 （ Declarative）

- 命令式编程（imperative）： 详细描述路径
  - 下个路口左转
  - 下个有红灯的路口右转
  - 前进100米
  - 在下个路口掉头
  - 前进1500米
  - 到达王府井大街出租车停车区
- 声明式编程（Declarative）： 只告诉目的地
  - 带我到王府井大街。

- Imperative:
  - C, Java
- Declarative:
  - SQL语句

例子二：c#

命令式编程（Imperative)会一步一步的告诉程序该怎么运行

```
List<int> array = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8 };
List<int> results = new List<int>();
foreach(var num in array)
{
    if (num % 2 != 0)
            results.Add(num);
}
```

如果使用声明式编程（ Declarative）则会是这样

```
List<int> array = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8 };
var results = array.Where( num => num % 2 != 0);
```

Dafny Syntax

- Methods
  - One of the basic units of any Dafny program.
- Pre-condition and Post-condition
  - requires/ensures
- Assertions
- Functions
- Loop invariants
- Termination
- Arrays
- Quantifiers
- Predicates
- Framing
- Collections, Modules, Lemmas……

Build-in specifications

Dafny

- Example:

```
method Triple(x: int) returns (r: int)
  ensures r == 3*x
  {
    var y := 2*x;
    r := x + y;
  }
```

'►' shortcut: Alt+B

```
Dafny 2.3.0.10506

Dafny program verifier finished with 1 verified, 0 errors
Program compiled successfully
```

```
method Triple(x: int) returns (r: int)
  ensures r == 3*x
  {
    var y := 2*x;
    r := x + y + y;
  }
```

'►' shortcut: Alt+B

		Description
	1	A postcondition might not hold on this return path.
	2	This is the postcondition that might not hold.

```
Dafny 2.3.0.10506
stdin.dfy(3,2): Error BP5003: A postcondition might not hold on this return path.
stdin.dfy(2,12): Related location: This is the postcondition that might not hold.
```

Dafny

- Example:

```
method Triple(x: int) returns (r: int)
  ensures r == 3*x
  {
    var y := Double(x);
    r := x + y;
  }

method Double(x: int) returns (r: int)
  ensures r == 2*x
  {
    r := x + x;
  }
```

'►' shortcut: Alt+B

```
Dafny 2.3.0.10506

Dafny program verifier finished with 2 verified, 0 errors
Program compiled successfully
```

```
method Triple(x: int) returns (r: int)
  ensures r == 3*x
  {
    var y := Double(x);
    r := x + y;
  }

method Double(x: int) returns (r: int)
  ensures r >= 2*x
  {
    r := x + x;
  }
```

'►' shortcut: Alt+B

		Description
	1	A postcondition might not hold on this return path.
	2	This is the postcondition that might not hold.

```
Dafny 2.3.0.10506
stdin.dfy(3,2): Error BP5003: A postcondition might not h
stdin.dfy(2,12): Related location: This is the postcondit
```

Dafny

- Example-function:

```
function abs(x: int): int
{
    if x < 0 then -x else x
}

```

```
method Abs(x: int) returns (y: int)
{
    if x < 0
        { return -x; }
    else
        { return x; }
}

```

```
function fib(n: nat): nat
{
    if n == 0 then 0 else
    if n == 1 then 1 else
                   fib(n - 1) + fib(n - 2)
}

```

Dafny

- Example-loop invariant:

```
function fib(n: nat): nat
{
    if n == 0 then 0 else
    if n == 1 then 1 else
                    fib(n - 1) + fib(n - 2)
}
method ComputeFib(n: nat) returns (b: nat)
    ensures b == fib(n)
{
    if n == 0 { return 0; }
    var i: int := 1;
    var a := 0;
        b := 1;
    while i < n
        invariant 0 < i <= n
        invariant a == fib(i - 1)
        invariant b == fib(i)
    {
        a, b := b, a + b;
        i := i + 1;
    }
```

Dafny

- Example-termination:

```
function fib(n: nat): nat
   decreases n
{
   if n == 0 then 0 else
   if n == 1 then 1 else
               fib(n - 1) + fib(n - 2)
}
```

Dafny

- Termination- decrease annotation:
- If there is no explicit decreases annotation, it tries to guess one.
  - has special rules for guessing the termination measure.
  - If the loop condition is a comparison of the form A< B, it makes the guess: decrease B-A

```
while i < n
   invariant 0 <= i <= n
{
   // do something interesting
   i := i + 1;
}
```

```
while i < n
   invariant 0 <= i <= n
   decreases n - i
{
   i := i + 1;
}
```

- Recursive functions and methods:

  - Analyzes which functions/methods call each other, to figure out possible recursion.

```
function fac(n: nat): nat
{
   if n == 0 then 1 else n * fac(n-1)
}
```

```
decreases n
```

Dafny

- Termination- decrease annotation- Collatz conjecture:

```
1  method hail(N: nat)
2
3  {
4     var n := N;
5     while 1 < n
6
7     {
8        n := if n % 2 == 0 then n / 2 else n * 3 + 1;
9     }
10 }
11
```

'►' shortcut: Alt+B

		Description	Line	Column
	1	cannot prove termination; try supplying a decreases clause for the loop	5	3

```
1  method hail(N: nat)
2     decreases *
3  {
4     var n := N;
5     while 1 < n
6        decreases *
7     {
8        n := if n % 2 == 0 then n / 2 else n * 3 + 1;
9     }
10 }
11
```

'►' shortcut: Alt+B

```
Dafny 2.3.0.10506

Dafny program verifier finished with 1 verified, 0 erro
Program compiled successfully
```

Dafny

- Sets

```
var s1 := {}; // the empty set
var s2 := {1, 2, 3}; // set contains exactly 1, 2, and 3
assert s2 == {1,1,2,3,3,3,3}; // same as before
var s3, s4 := {1,2}, {1,4};
```

```
assert 5 in {1,3,4,5};
assert 1 in {1,3,4,5};
assert 2 !in {1,3,4,5};
assert forall x :: x !in {};
```

```
assert (set x | x in {0,1,2,3,4,5} && x < 3) == {0,1,2};
```

- Sequences

```
predicate sorted(s: seq<int>)
{
   forall i,j :: 0 <= i < j < |s| ==> s[i] <= s[j]
}
```

- Maps

```
var m := map[4 := 5, 5 := 6]
assert m[4] == 5;
```

Dafny

- Lemmas

```
lemma SkippingLemma(a : array<int>, j : int)
   requires a != null
   requires forall i :: 0 <= i < a.Length ==> 0 <= a[i]
   requires forall i :: 0 < i < a.Length ==> a[i-1]-1 <= a[i]
   requires 0 <= j < a.Length
   ensures forall i :: j <= i < j + a[j] && i < a.Length ==> a[i] != 0
{
   ...
}

index := 0;
while index < a.Length
   invariant 0 <= index
   invariant forall k :: 0 <= k < index && k < a.Length ==> a[k] != 0
{
   if a[index] == 0 { return; }
   SkippingLemma(a, index);
   index := index + a[index];
}
index := -1;
```

(4,3,2,1,1,0):

$a[0] = 4$

Skip(a, 0): $0 \leq i < 0 + a[0]$

Skip: $a[1], a[2], a[3]$

Index = 0+$a[0] = 4$

$a[4] = 1$

Skip(a, 4): $4 \leq i < 4 + a[4]$

Index = 4+$a[4] = 5$

Find $a[5] = 0$

Dafny

- Modules
- A module body can consist of anything that you could put at the toplevel. This includes classes, datatypes, types, methods, functions, etc.
- Import & export:
  - Import: Give access to all declarations;
  - Export: control this more precisely.

```
module Helpers {
  function method addOne(n: nat): nat
  {
    n + 1
  }
}
module Mod {
  import A = Helpers
  method m() {
    assert A.addOne(5) == 6;
  }
}
```

```
module Helpers {
  export Spec provides addOne, addOne_result
  export Body reveals addOne
  export extends Spec
  function method addOne(n: nat): nat
  {
    n + 1
  }
  lemma addOne_result(n : nat)
     ensures addOne(n) == n + 1
  { }
}
```

Dafny

- In addition to proving a correspondence to user supplied annotations, Dafny proves that there are no run time errors, such as index out of bounds, null dereferences, division by zero, etc.

- Formalizing the semantics and proof obligations:
  - Translate it into an intermediate verification language: Boogie.

- Generating verification conditions from the intermediate program:
  - By existing tools (Boogie tool)

Dafny

- Dafny code
  - Desired properties
  - Implementation
  - proof

$\big\downarrow$ Dafny

- Boogie code

$\big\downarrow$ Boogie

- SMT formulas

$\big\downarrow$ Z3

- SAT, UNSAT, or Timeout

Boogie

- An intermediate verification language;

- Also the name of a tool.
  - Accepts the Boogie language as input;
  - Optionally infers some invariants in the given Boogie program;
  - Then generates verification conditions that passed to an SMT solver (Z3).

boogie

Is this property always true?

```
procedure F(n: int) returns (r: int)
  ensures 100 < n ==> r == n - 10;  // This postcondition is easy to check by hand
  ensures n <= 100 ==> r == 91;     // Do you believe this one is true?
{
  if (100 < n) {
    r := n - 10;
  } else {
    call r := F(n + 11);
    call r := F(r);
  }
}
```

home permalink

'►' shortcut: Alt+B

```
Boogie program verifier finished with 1 verified, 0 errors
```

Boogie

- Designed to make the prescription of verification conditions natural and convenient.

- It serves as a common intermediate representation for static program verifiers of various source languages, and it abstracts over the interfaces to various theorem provers.

- Boogie can also be used as a shared input and output format for techniques like abstract interpretation and predicate abstraction

Intermediate Language (Boogie)

- Boogie is used to encode the semantics and proof obligations of Dafny.
- Boogie consists of a mathematical part and an imperative part;
- Mathematical part:
  - Declarations of types, constants, and first-order functions, as well as axioms;

```
type Keyboard;
const Yamaha_DX7: Keyboard;
function keys(Keyboard) returns (int);
axiom (∀ k: Keyboard • 0 ⩽ keys(k) );
axiom keys(Yamaha_DX7) = 61;
```

- Imperative part:
  - Declarations of variables and procedures.

Intermediate Language (Boogie)

proof obligation:
assuming i == 0,
assuming $x + i > 5$,
prove $x > 5$

```
y + 3 > 5
x := y;
x + 3 > 5
i := 3;
x + i > 5
while (i != 0)
    invariant x + i > 5;
{
    (x + 1) + (i - 1) > 5
    i := i - 1;
    (x + 1) + i > 5
    x := x + 1;
    x + i > 5
}
x > 5
```

proof obligation:
assuming i != 0,
assuming $x + i > 5$,
prove $(x + 1) + (i - 1) > 5$

Boogie syntax

- **TYPES**

- **primitive types** t ::= bool | int | bv8 | bv16 | bv32 | real | …
- **array types** | [t]t

- **EXPRESSIONS**

- **variables** e, P ::= x
- **boolean expressions** | true | false | !e | e && e | e ==> e | ...
- **linear integer arithmetic** | ... | -2 | -1 | 0 | 1 | 2 | … | e + e | e – e | e <= e | ...
- **bit vector arithmetic** | e & e | e << e | ...
- **uninterpreted functions** | f(e,…,e)
- **arrays** | e[e] | e[e := e]
- **quantifiers** | (forall x:t :: e) | (exists x:t :: e)

- **STATEMENTS**

- **primitive statements** s ::= x := e | assert e; | assume e; | …
- **compound statements** | s1 s2 | if(e) {s1} else {s2} | while(e) invariant P; {s} | …

Intermediate Language (Boogie)

- Procedures:

Procedures A procedure is declared as follows:

$$\textbf{procedure } P(ins) \textbf{ returns } (outs);\ Spec$$

- Spec:
  - requires Expr: declares a precondition;
  - modifies xs: declares a modifies clause;
  - ensures Expr: declares a postcondition.

Verification Conditions

- A Boogie program is correct if all procedure implementations satisfy their specifications.

- To check it, Boogie performs a syntactic check and generates a verification condition to be discharged by a theorem prover.

- The proof obligations encoded by the verification condition:
  - Arise from the postcondition of the procedure being verified, the preconditions of called procedures, and the conditions in assert statements.

Verifying procedure implementations

- Every procedure implementation is checked to satisfy its specification.
  - The modifies clause is checked syntactically;
  - The pre- and postconditions of the procedure, are checked semantically;

Verifying procedure implementations

- Example :

$$\begin{array}{l}\textbf{procedure } P(ins) \textbf{ returns } (outs);\\ \quad \textbf{requires } Pre;\\ \quad \textbf{modifies } gs;\\ \quad \textbf{ensures } Post;\end{array}$$

  - With an implementation

$$\textbf{var } locals; \quad stmts$$

  - Stmt:

$$\begin{array}{rcl} Stmt & ::= & xs := Exprs;\\ & | & x[Exprs] := Expr;\\ & | & \textbf{havoc } xs;\\ & | & \textbf{if } (Expr)\ \{\ Stmts\ \}\ \textbf{else}\ \{\ Stmts\ \}\\ & | & \textbf{while } (Expr)\ Invs\ \{\ Stmts\ \}\\ & | & \textbf{assert } Expr;\\ & | & \textbf{assume } Expr;\\ & | & \textbf{call } xs := P(Exprs);\end{array}$$

Verifying procedure implementations

- Example :

$$\textbf{procedure } P(ins) \textbf{ returns } (outs); \\ \quad \textbf{requires } Pre; \\ \quad \textbf{modifies } gs; \\ \quad \textbf{ensures } Post;$$

  - With an implementation

$$\textbf{var } locals; \quad stmts$$

  - Verification condition for this implementation of P is given by:

$$Axs \Rightarrow \mathsf{wp}[\![Impl, \textbf{true}]\!]$$

    - Axs: conjunction of axioms declared in the program.
    - Impl:

$$\textbf{assume } Pre; \quad gs' := gs; \quad stmts''; \quad \textbf{assert } Post';$$

    - wp: for any statement S and condition Q on the post-state of S, the weakest precondition of S with respect to Q, denoted wp[S,Q] .

Weakest Precondision

If P = wlp(s, Q), then P is in some sense the "best" (weakest) P such that *(P)* s *(Q)*. Thus, wlp provides an algorithm for computing P.

$y + 3 > 5$
`x:=y;`
$x + 3 > 5$
`i:=3;`
$x + i > 5$

wlp(`x:=y;`, $x + 3 > 5$)
= $(x + 3 > 5)\{x := y\}$
= $y + 3 > 5$

wlp(`i:=3;`, $x + i > 5$)
= $(x + i > 5)\{i := 3\}$
= $x + 3 > 5$

wlp(`x:=y; i:=3;`, $x + i > 5$)
= wlp(`x:=y`, wlp(`i:=3`, $x + i > 5$))
= wlp(`x:=y`, $x + 3 > 5$)
= $y + 3 > 5$

wlp(x := e, *P*) = *P{x := e}*
wlp(assert e, *P*) = *e && P*
wlp(assume e, *P*) = *e ==> P*
wlp(s1 s2, *P*) = wlp(s1, wlp(s2, *P*))
wlp(if(e) {s1} else {s2}, *P*) = *(e ==>* wlp(s1, *P*)*) && (!e ==>* wlp(s2, *P*)*)*

Dafny

- A Dafny program consists of a set of named classes:

$$
\begin{array}{rcl}
\mathit{Program} & ::= & \mathit{Classes} \\
\mathit{Class} & ::= & \textsf{\textbf{class}}\ \mathit{Id}\ \{\ \mathit{Members}\ \} \\
\mathit{Member} & ::= & \mathit{Field} \mid \mathit{Method} \mid \mathit{Function}
\end{array}
$$

Dafny

- translation :

  - The Boogie translation consists of a prelude of declarations, which encodes some properties of all Dafny programs, and the Boogie declarations decl[d] for every Dafny class declaration d.

  - Classes:
    - The prelude declares a type and each class declaration is translated as follows:

$$\textbf{type}\ \mathit{ClassName};$$

$$\begin{aligned}
&\mathsf{decl}[\![\, \textsf{\textbf{class}}\ C\ \{\ mm\ \}\,]\!] \ = \\
&\quad \textbf{const unique}\ \mathit{class}.C\!: \mathit{ClassName}; \\
&\quad \mathsf{decl}^{*}[\![\, mm \,]\!]
\end{aligned}$$

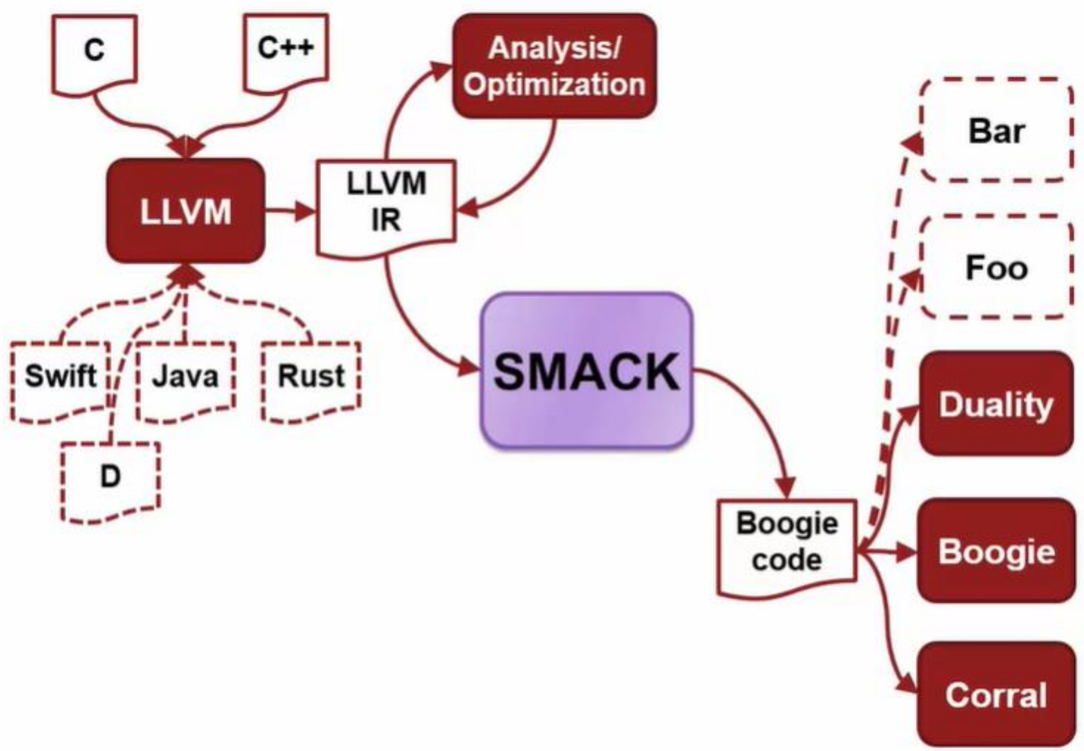
  - Methods:

$$\begin{array}{rcl}
\mathit{Method} & ::= & \textsf{\textbf{method}}\ \mathit{Id}(\mathit{Params})\ \textsf{\textbf{returns}}\ (\mathit{Params})\ \mathit{Specs}\ \{\ \mathit{Stmts}\ \} \\
\mathit{Param} & ::= & \mathit{Id} : \mathit{Type} \\
\mathit{Spec} & ::= & \textsf{\textbf{requires}}\ \mathit{Expr}\ ;\ \mid \textsf{\textbf{modifies}}\ \mathit{Exprs}\ ;\ \mid \textsf{\textbf{ensures}}\ \mathit{Expr}\ ;
\end{array}$$

    - A method is translated into a procedure in Boogie.

Smack

- Translator from LLVM IR into Boogie;
- Modular and extensible software verification ecosystem; (Boogie & LLVM)
- Verifier of assertions in C/C++ programs.



Smack

- Example:

Bit-vector Example

```c
// bitvector_example.c
#include <limits.h>
struct a {
  int i;
  int j;
};

unsigned absolute(int value)
{
  unsigned int r;
  int mask, sz, cb;
  sz = sizeof(int);
  cb = CHAR_BIT;
  mask = value >> sz * cb - 1;
  r = (value + mask) ^ mask;
  return r;
}
```

```c
int main(void)
{
  struct a x = {-10, 20};
  // valid yet unsafe pointer type cast
  char *p = (char *)(&(x.j));
  *p = 1;
  __VERIFIER_assert(x.j == 1);
  // callee contains bit-wise operations
  x.i = __VERIFIER_nondet_int();
  x.i = absolute(x.i);
  __VERIFIER_assert(x.i >= 0);
  return 0;
}
```

```
/usr/local/share/smack/lib/smack.c(37,1): Trace: Thread=1
bitvector_example.c(28,3): Trace: Thread=1  (RETURN from __VERIFIER_assert )
bitvector_example.c(30,9): Trace: Thread=1
bitvector_example.c(32,9): Trace: Thread=1
bitvector_example.c(32,9): Trace: Thread=1  (CALL absolute)
bitvector_example.c(16,3): Trace: Thread=1
bitvector_example.c(16,3): Trace: Thread=1  (value = 2147483648bv32)
bitvector_example.c(17,3): Trace: Thread=1
bitvector_example.c(17,3): Trace: Thread=1  (mask = 4294967295bv32)
bitvector_example.c(18,3): Trace: Thread=1
bitvector_example.c(18,3): Trace: Thread=1  (r = 2147483648bv32)
bitvector_example.c(19,3): Trace: Thread=1
bitvector_example.c(32,9): Trace: Thread=1  (RETURN from absolute )
bitvector_example.c(32,9): Trace: Thread=1
bitvector_example.c(33,3): Trace: Thread=1
bitvector_example.c(33,3): Trace: Thread=1  (CALL __VERIFIER_assert)
/usr/local/share/smack/lib/smack.c(78,3): Trace: Thread=1
/usr/local/share/smack/lib/smack.c(36,21): Trace: Thread=1
/usr/local/share/smack/lib/smack.c(36,21): Trace: Thread=1  (ASSERTION FAILS)
bitvector_example.c(33,3): Trace: Thread=1  (RETURN from __VERIFIER_assert )
bitvector_example.c(33,3): Trace: Thread=1  (Done)
```

Smack

- Example:

Unbounded Loop Example

```c
// unboundedLoop_example.c
int main() {
    long x = __VERIFIER_nondet_long();
    long y = 0;
    long z = 0;
    assume(x > 100);
    for (; y < x; ++y)
        z += 1;
    assert(z != x);
    return 0;
}
```

```
smack.py unboundedLoop_example.c --unroll 10 -v
```

→ Report "No bug"

```
smack.py unboundedLoop_example.c --verifier duality
```

→ Assert failed

KLEE

- KLEE is a dynamic symbolic execution engine built on top of the LLVM compiler infrastructure;

- Automatically generating tests that achieve high coverage on a diverse set of complex and environmentally-intensive programs;

- Also a bug finding tool.

KLEE

- KLEE uses STP solver;
- Coverage-optimized search and Random path search.
- Compared to our tools?
  - Sample a path from CFA (control flow automaton);
  - LLVM based;

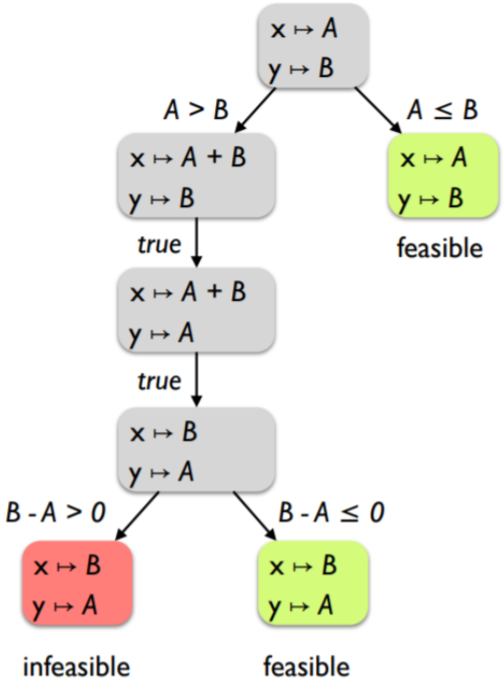
```
def f (x, y):
  if (x > y):
    x = x + y
    y = x − y
    x = x − y
    if (x − y > 0):
      assert false
  return (x, y)
```

Execute the program on *symbolic values.*

Symbolic state maps variables to symbolic values.

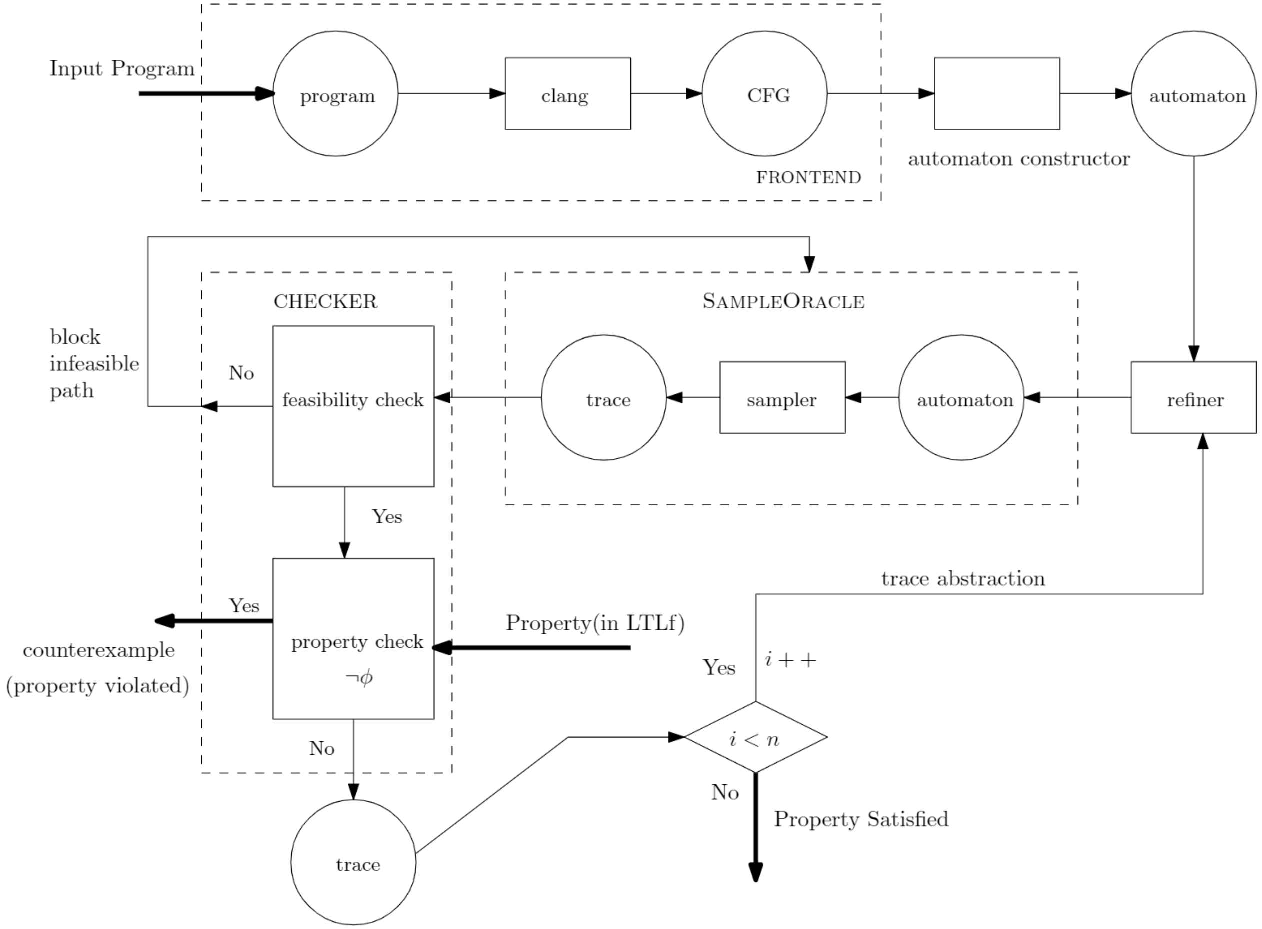
Path condition is a logical formula over the symbolic inputs that encodes all branch decisions taken so far.

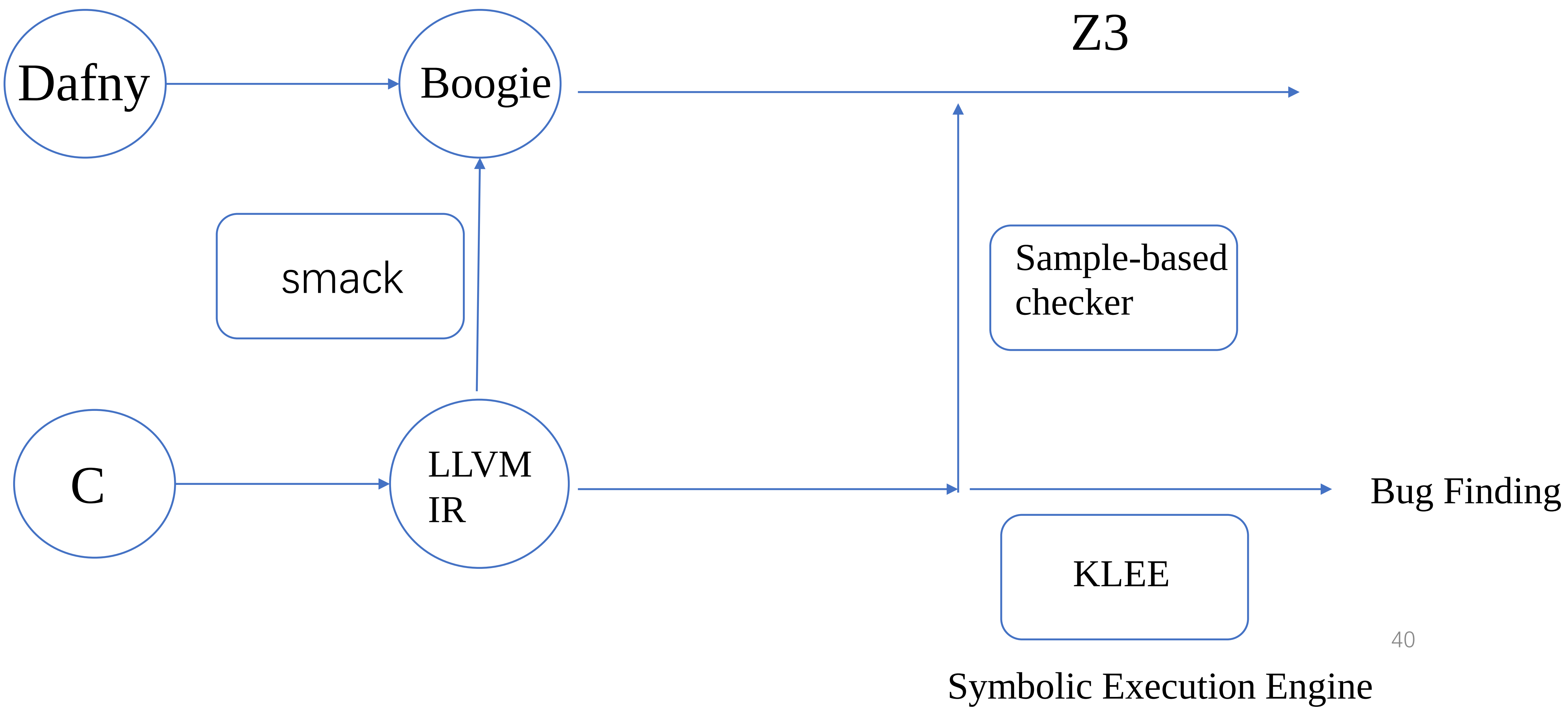
All paths in the program form its *execution tree*, in which some paths are *feasible* and some are *infeasible.*



Sample-based Checker

Flowgraph of the tool




Z3
Dafny
Boogie
smack
Sample-based checker
C
LLVM IR
Bug Finding
KLEE
Symbolic Execution Engine