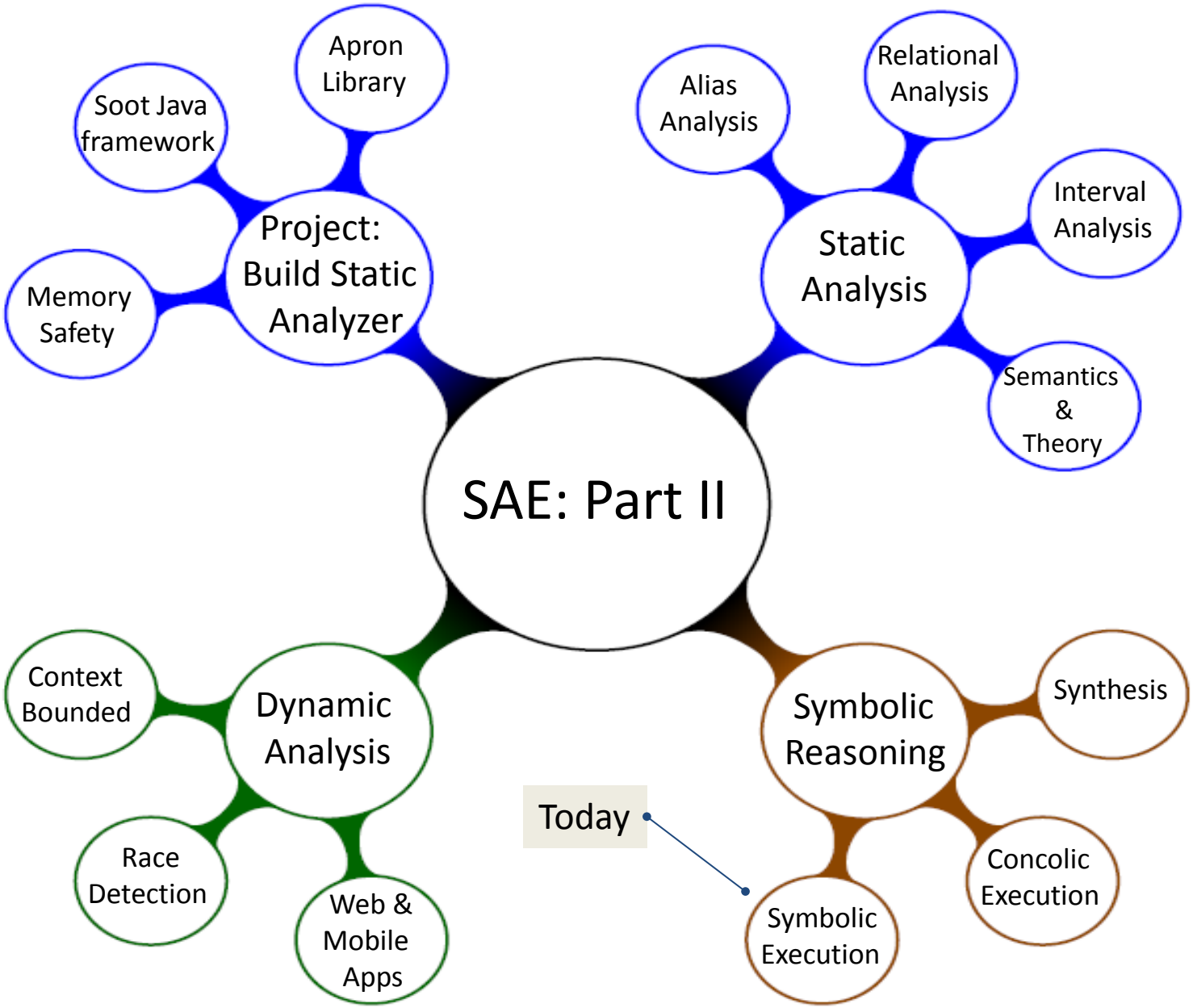
Software Architecture and Engineering: Part II

ETH Zurich, Spring 2014
Prof. Martin Vechev


SAE: Part II
Project: Build Static Analyzer
Apron Library
Soot Java framework
Memory Safety
Static Analysis
Alias Analysis
Relational Analysis
Interval Analysis
Semantics & Theory
Dynamic Analysis
Context Bounded
Race Detection
Web & Mobile Apps
Symbolic Reasoning
Synthesis
Concolic Execution
Symbolic Execution
Today

Static Program Analysis: The Good

- Automatically proves a program is free of errors
  - once transformers are known, that is, the abstract semantics of the language, we can analyze any program
- Based on a solid mathematical foundation
  - well-defined notions of soundness and approximation
- Has been widely used for many problems:
  - flight-control software, algorithm verification, code search, code synthesis, biological systems, etc…

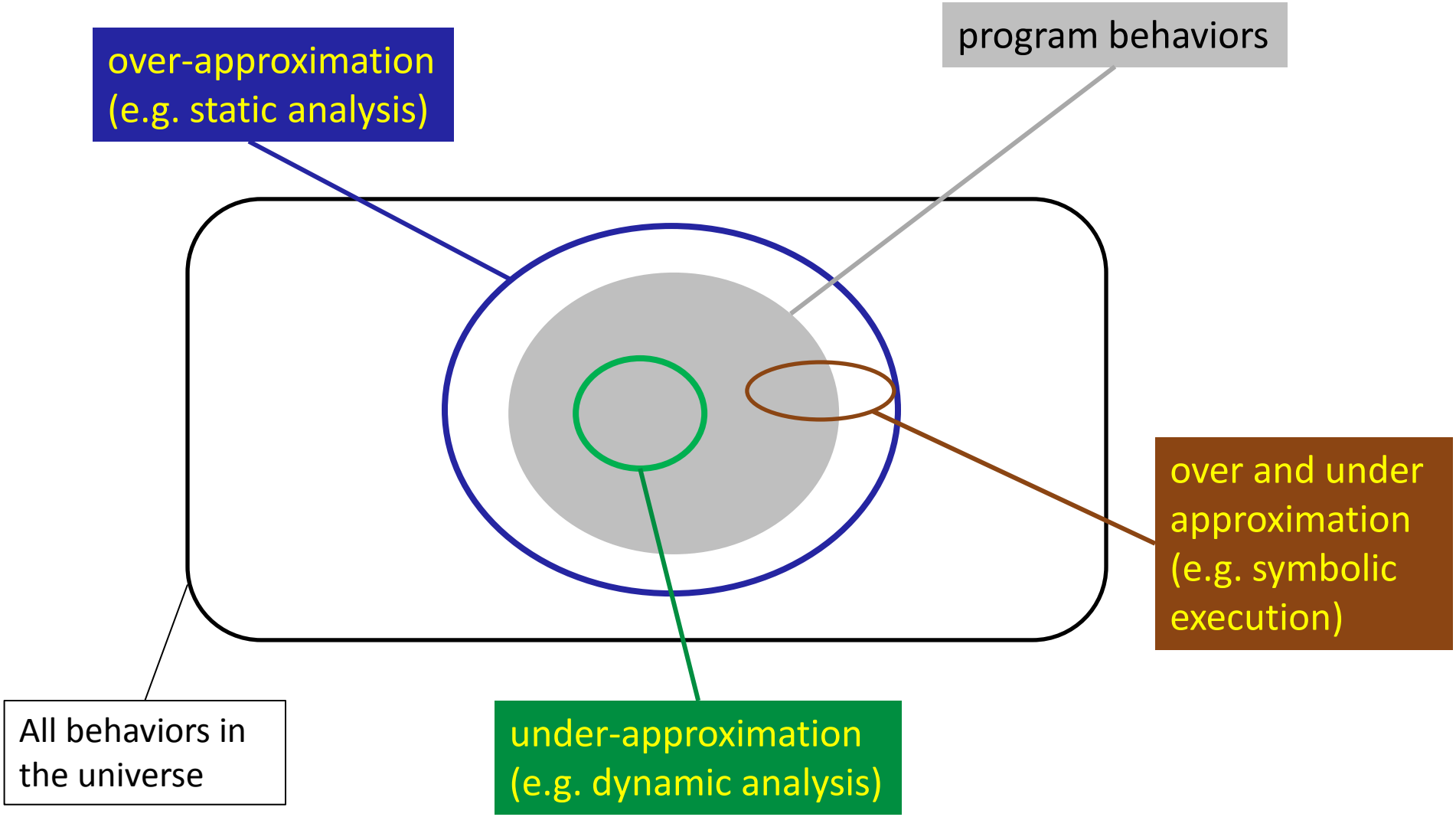
Static Program Analysis: The Bad

- Crafting the abstract transformers may be tricky
  - must worry about soundness and precision
  - although predicate abstraction helps, its key limitation is that the domain is finite

- The abstract transformers can be expensive and not scale to large programs.

- False positives: can be difficult to tell if a reported bug is due to approximation or it is a real error

Approaches to Program Analysis



Next: Classical Symbolic Execution

Symbolic Execution

Symbolic execution is a technique which sits in between testing and static analysis.

It is completely automatic, and aims to explore as many program executions as possible, with the expense that it has false negatives: it may miss program executions, that is, may miss errors.

Hence, symbolic execution is a particular instance of an under-approximation (some versions are actually both under- and over- approximations)

Symbolic Execution: Applications

Symbolic execution is widely used in practice. Tools based on symbolic execution have found serious errors and security vulnerabilities in various systems:

- Network servers
- File systems
- Device drivers
- Unix utilities
- Computer vision code
- …

Symbolic Execution: The Idea

In classic symbolic execution, we associate with each variable a symbolic value instead of a concrete value. We then run the program with the symbolic values obtaining a big constraint formula as we run the program. Hence, the name symbolic execution.

At any program point we can invoke a constraint (SMT) solver to find satisfying assignments to the formula. These satisfying assignments can be used to indicate real concrete inputs for which the program reaches a program point or to steer the analysis to another part of the program.

Constraints: Examples

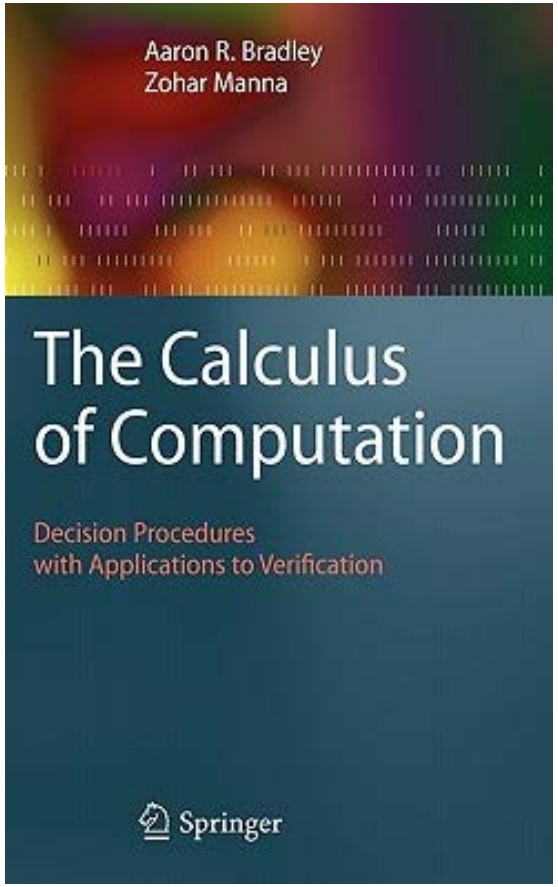
Linear constraint: $5*x + 6 < 100$

Non-linear constraint: $x * y + 12 < 29$

Uninterpreted functions: $f(x) < 30$

A constraint solver, typically an SMT solver, finds satisfying assignments to constraints. An example of SMT solvers are Z3 and Yices.

Logical Fragments of Constraints



What is decidable ?

Theory	Description	Full	QFF
T_{E}	equality	no	yes
T_{PA}	Peano arithmetic	no	no
$T_{\mathbb{N}}$	Presburger arithmetic	yes	yes
$T_{\mathbb{Z}}$	linear integers	yes	yes
$T_{\mathbb{R}}$	reals (with $\cdot$)	yes	yes
$T_{\mathbb{Q}}$	rationals (without $\cdot$)	yes	yes
T_{RDS}	recursive data structures	no	yes
T^{+}_{RDS}	acyclic recursive data structures	yes	yes
T_{A}	arrays	no	yes
$T^{=}_{\mathsf{A}}$	arrays with extensionality	no	yes

Symbolic Execution: Technically

At any point during program execution, symbolic execution keeps two formulas:

symbolic store and a path constraint

Therefore, at any point in time the symbolic state is described as the conjunction of these two formulas.

Symbolic Store

- The values of variables at any moment in time are given by a function $\sigma_s \in \text{SymStore} = \text{Var} \rightarrow \text{Sym}$
  - Var is the set of variables as before
  - Sym is a set of symbolic values
  - σ_s is called a symbolic store

- Example: $\sigma_s : \texttt{x} \mapsto \texttt{x0}, \ \texttt{y} \mapsto \texttt{y0}$

Semantics

- Arithmetic expression evaluation simply manipulates the symbolic values.

- Let σ_s : `x` $\mapsto$ `x0, y` $\mapsto$ `y0`

- Then, `z = x + y` will produce the symbolic store:

  `x` $\mapsto$ `x0, y` $\mapsto$ `y0, z` $\mapsto$ `x0+y0`

That is, we literally keep the symbolic expression `x0+y0`

Path Constraint

- The analysis keeps a path constraint (pct) which records the history of all branches taken so far. The path constraint is simply a formula.
- The formula is typically in a decidable logical fragment without quantifiers
- At the start of the analysis, the path constraint is true
- Evaluation of conditionals affects the path constraint , but not the symbolic store.

Path Constraint: Example

Let σ_s : x ↦ x0, y ↦ y0

Let pct = x0 > 10

Lets evaluate: `if (x > y + 1) {5: … }`

At label 5, we will get the symbolic store σ_s . It does not change. But we will get an updated path constraint:

$$\text{pct} = x0 > 10 \wedge x0 > y0 + 1$$

Symbolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

Lets execute this example
with classic symbolic execution

Symbolic Execution: Example

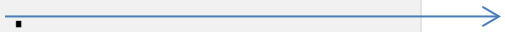
```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

The read() functions read a value from the input and because we don't know what those read values are, we set the values of `x` and `y` to fresh symbolic values called `x0` and `y0`

pct is true because so far we have not executed any conditionals



σ_s : `x` ↦ `x0`, `y` ↦ `y0`

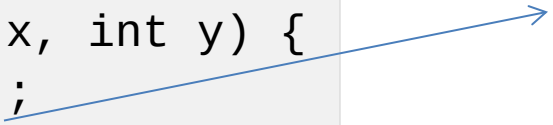
pct : true

Symbolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```



σ_s : x ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : true

Here, we simply executed the function twice() and added the new symbolic value for z.

Symbolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

We forked the analysis into 2 paths: the true and the false path. So we **duplicate** the state of the analysis.

This is the result if x = z:

σ_s : x ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : x0 = 2*y0

This is the result if x != z:

σ_s : x ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : x0 ≠ 2*y0

Symbolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

We can avoid further exploring a path if we know the constraint pct is **unsatisfiable**. In this example, both pct's are satisfiable so we need to keep exploring both paths.

This is the result if x = z:

σ_s : x ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : x0 = 2*y0

This is the result if x != z:

σ_s : x ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : x0 ≠ 2*y0

Symbolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

Lets explore the path when x == z is true.
Once again we get 2 more paths.

This is the result if $x > y + 10$:

σ_s : $x \mapsto x0$, $y \mapsto y0$, $z \mapsto 2*y0$

pct : $x0 = 2*y0 \wedge x0 > y0+10$

This is the result if $x \leq y + 10$:

σ_s : $x \mapsto x0$, $y \mapsto y0$, $z \mapsto 2*y0$

pct : $x0 = 2*y0 \wedge x0 \leq y0+10$

Symbolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

So the following path reaches "**ERROR**".

This is the result if x > y + 10:

σ_s : x ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : x0 = 2*y0
∧
x0 > y0+10

We can now ask the SMT solver for a satisfying assignment to the pct formula.

For instance, `x0 = 40, y0 = 20` is a satisfying assignment. That is, running the program with these inputs triggers the error.

Handling Loops: a limitation

```
int F(unsigned int k) {
  int sum = 0;
  int i  = 0;
  for ( ; i < k; i++)
     sum += i;
  return sum;
}
```

A serious limitation of symbolic execution is handling unbounded loops. Symbolic execution runs the program for a finite number of paths. But what if we do not know the bound on a loop ? The symbolic execution will keep running forever !

Handling Loops: bound loops

```
int F(unsigned int k) {
  int sum = 0;
  int i  = 0;
  for ( ; i < 2; i++)
     sum += i;
  return sum;
}
```

A common solution in practice is to provide some loop bound. In this example, we can bound k, to say 2. This is an example of an under-approximation. Practical symbolic analyzers usually under-approximate as most programs have unknown loop bounds.

Handling Loops: loop invariants

```
int F(unsigned int k) {
  int sum = 0;
  int i  = 0;
  for ( ; i < k; i++)
     sum += i;
  return sum;
}
```

Another solution is to provide a loop invariant , but this technique is rarely used for large programs because it is difficult to provide such invariants manually and it can also lead to over-approximation. This is where a combination with static program analysis is useful (static analysis can infer loop invariants). We will not study this approach in our treatment, but we note that the approach is used in program verification.

Constraint Solving: challenges

Constraint solving is fundamental to symbolic execution as a constraint solver is continuously invoked during analysis. Often, the main **roadblock to performance** of symbolic execution engines is the time spent in constraint solving. Therefore, it is important that:

1. The SMT solver supports as many decidable logical fragments as possible. Some tools use more than one SMT solver.

2. The SMT solver can solve large formulas quickly.

3. The symbolic execution engines tries to reduce the burden in calling the SMT solver by exploring domain specific insights.

Key Optimization: Caching

The basic insight here is that often, the analysis will invoke the SMT solver with similar formulas. Therefore, the symbolic execution system can keep a map (cache) of formulas to a satisfying assignment for the formula.

Then, when the engine builds a new formula and would like to find a satisfying assignment for that formula, it can first access the cache, before calling the SMT solver.

Key Optimization: Caching

Suppose the cache contains the mapping:

Formula:		Solution:
$(x + y < 10) \wedge (x > 5)$	$\rightarrow$	$\{x = 6, y = 3\}$

If we get a weaker formula as a query, say $(x + y < 10)$, then we can immediately reuse the solution already found in the cache, without calling the SMT solver.

If we get a stronger formula as a query, say $(x + y < 10) \wedge (x > 5) \wedge (y \geqslant 0)$, then we can quickly try the solution in the cache and see if it works, without calling the solver (in this example, it works).

When Constraint Solving Fails

Despite best efforts, the program may be using constraints in a fragment which the SMT solver does not handle well.

For instance, suppose the SMT solver does not handle non-linear constraints well.

Let us consider a modification of our running example.

Modified Example

```
int twice(int v) {
  return v * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

Here, we changed the twice() function to contain a non-linear result.

Let us see what happens when we symbolically execute the program now…

Modified Example

```
int twice(int v) {
  return v * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

This is the result if x = z:

σ_s : x ↦ x0,
y ↦ y0
z ↦ y0*y0

pct : x0 = y0*y0

Now, if we are to invoke the SMT solver with the pct formula, it may be unable to compute satisfying assignments, precluding us from knowing whether the path is feasible or not.

Solution: Concolic Execution

Concolic Execution: combines both symbolic execution and concrete (normal) execution.

The basic idea is to have the concrete execution drive the symbolic execution.

Here, the program runs as usual (it needs to be given some input), but in addition it also maintains the usual symbolic information.

Concolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

The read() functions read a value from the input. Suppose we read x = 22 and y = 7.

We will keep both the concrete store and the symbolic store and path constraint.

σ : x ↦ 22,
y ↦ 7

σ_s : x ↦ x0,
y ↦ y0

pct : true

Concolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

σ : x ↦ 22,
y ↦ 7,
z ↦ 14

σ_s : x ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : true

The concrete execution will now take the 'else' branch of z == x.

Concolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

Hence, we get:

σ : x ↦ 22,
y ↦ 7,
z ↦ 14

σ_s : x ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : x0 ≠ 2*y0

Concolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

At this point, concolic execution decides that it would like the explore the “true” branch of x == z and hence it needs to generate concrete inputs in order to explore it. Towards such inputs, it negates the pct constraint, obtaining:

$$pct : x0 = 2*y0$$

It then calls the SMT solver to find a satisfying assignment of that constraint. Let us suppose the SMT solver returns:

$$x0 \mapsto 2,\ y0 \mapsto 1$$

The concolic execution then runs the program with this input.

Concolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

With the input $x \mapsto 2,\ y \mapsto 1$ we reach this program point with the following information:

$\sigma : x \mapsto 2,\ y \mapsto 1,\ z \mapsto 2$

$\sigma_s : x \mapsto x0,\ y \mapsto y0,\ z \mapsto 2*y0$

$pct : x0 = 2*y0$

Continuing further we get:

Concolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

We reach the “else” branch of x > y + 10

σ : x ↦ 2,
y ↦ 1,
z ↦ 2

σ_s : x ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : x0 = 2*y0
∧
x0 ≤ y0+10

Again, concolic execution may want to explore the ‘true’ branch of x > y + 10.

Concolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

We reach the "else" branch of x > y + 10

σ : x ↦ 2,
y ↦ 1,
z ↦ 2

σ_s : x ↦ x0,
y ↦ y0
z ↦ 2*y0

pct : x0 = 2*y0
∧
x0 ≤ y0+10

Concolic execution now negates the conjunct x0 ≤ y0+10 obtaining:

x0 = 2*y0 ∧ x0 > y0+10

A satisfying assignment is: x0 ↦ 30, y0 ↦ 15

Concolic Execution: Example

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

If we run the program with the input:

$$x0 \mapsto 30,\ y0 \mapsto 15$$

we will now reach the ERROR state.

As we can see from this example, by keeping the symbolic information, the concrete execution can use that information in order to obtain new inputs.

Non-linear constraints

Let us return to the problem of non-linear constraints

Non-linear constraints

```
int twice(int v) {
  return v * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

Let us again consider our example and see what concolic execution would do with non-linear constraints.

Concolic Execution: Example

```
int twice(int v) {
  return v * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

The read() functions read a value from the input. Suppose we read x = 22 and y =7.

σ : x ↦ 22,
y ↦ 7

σ_s : x ↦ x0,
y ↦ y0

pct : true

Concolic Execution: Example

```
int twice(int v) {
  return v * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

σ : x ↦ 22,
y ↦ 7,
z ↦ 49

σ_s : x ↦ x0,
y ↦ y0
z ↦ y0*y0

pct : true

The concrete execution will now take the 'else' branch of x == z.

Concolic Execution: Example

```
int twice(int v) {
  return v * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

Hence, we get:

σ : x ↦ 22,
y ↦ 7,
z ↦ 49

σ_s : x ↦ x0,
y ↦ y0
z ↦ y0*y0

pct : x0 ≠ y0*y0

Concolic Execution: Example

```
int twice(int v) {
  return v * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

However, here we have a non-linear constraint $x0 \neq y0*y0$. If we would like to explore the true branch we negate the constraint, obtaining $x0 = y0*y0$ but again we have a non-linear constraint !

In this case, concolic execution simplifies the constraint by plugging in the concrete values for $y0$ in this case, 7, obtaining the simplified constraint:

$$x0 = 49$$

Hence, it now runs the program with the input

$$x \mapsto 49, \quad y \mapsto 7$$

Concolic Execution: Example

```
int twice(int v) {
  return v * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

Running with the input

$$x \mapsto 49, \quad y \mapsto 7$$

will reach the error state.

However, notice that with these inputs, if we try to simplify non-linear constraints by plugging in concrete values (as concolic execution does), then concolic execution we will never reach the else branch of the `if (x > y + 10)` statement.

Concolic Execution: External World

```
int F(char *f) {
  FILE *fp;
  fp = fopen(f, “r”);
  …
}
```

Often, we are not interested in symbolically executing OS code or native libraries. By keeping both the concrete and the symbolic values, we can always invoke system calls or library API with the concrete values and simply continue concrete execution as usual.

Symbolic Execution vs. Abstract Interpretation

We can think of Symbolic Execution as Abstract Interpretation in a particular abstract domain (logical formulas) where we do not perform joins ($\sqcup$).

That is, in symbolic execution, when we are not sure, we under-approximate (e.g. concolic execution), while in abstract interpretation we over-approximate.

Symbolic Execution: Tools

- Stanford's KLEE: http://klee.llvm.org/
- NASA's Java PathFinder: http://javapathfinder.sourceforge.net/
- Microsoft Research's SAGE
- UC Berkeley's CUTE
- EPFL's S2E: http://dslab.epfl.ch/proj/s2e

Summary

- Symbolic Execution is a popular technique for analyzing large programs
  - completely automated, relies on SMT solvers
- To terminate, may need to bound loops
  - leads to under-approximation
- To handle non-linear constraints and external environment, mixes concrete and symbolic execution (called concolic execution)
  - also leads to under-approximation