Automated Reasoning for Testing Specifications and Programs

Brian Campbell

AR guest lecture, March 2015

Introduction

I'm going to talk about

SMT automated theorem provers

and two applications to testing:

1. Test case generation for an instruction set model
2. High coverage test case generation for programs

SMT solvers

*S*atisfiability *M*odulo *T*heories

These solvers look for values for variables that satisfy a formula.

For example,

$$x > 0 \land y > 0 \land x < 2 \land y < 2 \land (x < 1 \lor y < 1)$$

for $x, y \in \mathbb{R}$ is satisfied by points (x, y) in an L-shape.

Notice the mix of

- boolean logic $(\land, \lor)$
- real arithmetic $(>, <)$

SMT solvers

*S*atisfiability *M*odulo *T*heories

These solvers look for values for variables that satisfy a formula.

For example,

$$x > 0 \land y > 0 \land x < 2 \land y < 2 \land (x < 1 \lor y < 1)$$

for $x, y \in \mathbb{R}$ is satisfied by points (x, y) in an L-shape.

Notice the mix of

- boolean logic $(\land, \lor)$ satisfiability
- real arithmetic $(>, <)$ theory

Solvers usually feature many theories.

Satisfiability

Our core logical 'glue' is satisfiability (SAT):

> *Given a boolean formula (variables, $\wedge$, $\vee$ and $\neg$) can we find an assignment of true and false to the variables that make the formula true?*

Our simple example has a boring boolean structure:

$$\begin{array}{ccccccccccccc} x > 0 & \wedge & y > 0 & \wedge & x < 2 & \wedge & y < 2 & \wedge & (x < 1 & \vee & y < 1) \\ l_1 & \wedge & l_2 & \wedge & l_3 & \wedge & l_4 & \wedge & (l_5 & \vee & l_6) \end{array}$$

Satisfiability

Our core logical 'glue' is satisfiability (SAT):

> *Given a boolean formula (variables, $\wedge$, $\vee$ and $\neg$) can we find an assignment of true and false to the variables that make the formula true?*

Instead, let's have an example stolen from Inf 1 (Fourman):

You may not take both Archeology and Chemistry
If you take Biology you must take Chemistry
You must take Biology or Archeology
If you take Chemistry you must take Divinity
You may not take both Divinity and Biology

$$(\neg A \vee \neg C) \wedge (\neg B \vee C) \wedge (B \vee A) \wedge (\neg C \vee D) \wedge (\neg D \vee \neg B)$$

Complexity of SAT

SAT is NP-complete.

Informally, **NP** is a class of problems that

- are efficient to check solutions of
- have no known algorithm asymptotically better than searching

NP-complete means that any other NP problem can be translated into SAT [Cook,Levin].

1st example of an NP-complete problem.

Complexity of SAT

SAT is NP-complete.

Informally, **NP** is a class of problems that

- are efficient to check solutions of
- have no known algorithm asymptotically better than searching

NP-complete means that any other NP problem can be translated into SAT [Cook,Levin].

1st example of an NP-complete problem.

However, this complexity affects arbitrary SAT problems.
Real instances of SAT perform well.
Widely used in large problems, e.g., verifying digital circuits.

Solving SAT

First of all, translate to Conjunctive Normal Form (CNF):

$$\begin{aligned} \neg(A \wedge (\neg B \vee C)) &= \neg A \vee \neg(\neg B \vee C) \\ &= \neg A \vee (B \wedge \neg C) \\ &= (\neg A \vee B) \wedge (\neg A \vee \neg C) \end{aligned}$$

- Conjunctions outermost
- Disjunctions in the middle
- Negation and variables innermost

That is, every formula has the form

$$\bigwedge_i \bigvee_j l_{i,j}$$

where $l_{i,j}$ are literals — either a variable or a negated variable.

Solving SAT

Our course choice example is already in CNF:

$$(\neg A \vee \neg C) \wedge (\neg B \vee C) \wedge (B \vee A) \wedge (\neg C \vee D) \wedge (\neg D \vee \neg B)$$

The $\wedge$s are normally replaced by a set of *clauses*:

$$\{\neg A \vee \neg C,\ \neg B \vee C,\ B \vee A,\ \neg C \vee D,\ \neg D \vee \neg B\}$$

To solve this, I'll demonstrate the DPLL (Davis-Putnam-Logemann-Loveland) procedure. Its core involves:

- picking a variable from the formula to be true
- propagating this choice to the other clauses
- backtracking when a choice is impossible

Solving the course choice example

We start with

$$\{\neg A \vee \neg C,\ \neg B \vee C,\ B \vee A,\ \neg C \vee D,\ \neg D \vee \neg B\}$$

Choose C.

$$C \quad ; \quad \{\neg A \vee \neg C,\ \neg B \vee C,\ B \vee A,\ \neg C \vee D,\ \neg D \vee \neg B\}$$

Propagating C to the clauses, we must have $\neg A$ for the first clause, the second is true, and D from the third.

$$C, \neg A, D \quad ; \quad \{B \vee A, \neg D \vee \neg B\}$$

Now propagate $\neg A$,

$$C, \neg A, D, B \quad ; \quad \{\neg D \vee \neg B\}$$

But we're in trouble: both D and B are already true.
Backtrack on C.

Solving the course choice example

We start with

$$\{\neg A \vee \neg C,\ \neg B \vee C,\ B \vee A,\ \neg C \vee D,\ \neg D \vee \neg B\}$$

Backtrack on C.

$$\neg C \quad ; \quad \{\neg B \vee C,\ B \vee A,\ \neg C \vee D,\ \neg D \vee \neg B\}$$

Propagate $\neg C$.

$$\neg C, \neg B \quad ; \quad \{B \vee A,\ \neg D \vee \neg B\}$$

Propagate $\neg B$.

$$\neg C, \neg B, A \quad ; \quad \{\}$$

It's satisfiable, with A true and B and C false.

Solving the course choice example

We start with

$$\{\neg A \vee \neg C,\ \neg B \vee C,\ B \vee A,\ \neg C \vee D,\ \neg D \vee \neg B\}$$

Backtrack on C.

$$\neg C \quad ; \quad \{\neg B \vee C,\ B \vee A,\ \neg C \vee D,\ \neg D \vee \neg B\}$$

Propagate $\neg C$.

$$\neg C, \neg B \quad ; \quad \{B \vee A,\ \neg D \vee \neg B\}$$

Propagate $\neg B$.

$$\neg C, \neg B, A \quad ; \quad \{\}$$

It's satisfiable, with A true and B and C false.
Note that D can be whatever we like.

Theories

Satisfaction provides a core of basic logic. The theories will add domain-specific reasoning on top.

Recall our simple example:

$$\begin{array}{ccccccccccc} x > 0 & \wedge & y > 0 & \wedge & x < 2 & \wedge & y < 2 & \wedge & (x < 1 & \vee & y < 1) \\ l_1 & \wedge & l_2 & \wedge & l_3 & \wedge & l_4 & \wedge & (l_5 & \vee & l_6) \end{array}$$

We want the theory implementation to allow us to

- gradually add clauses like $x > 0$,
- propagate information from them amongst the other clauses,
- point out conflicts (and backtrack)
- produce satisfying values

Theories in practice

Some examples of theories found in common SMT solvers:

- Arithmetic
  - For integers, rationals, or reals
  - A variety of operators (+,*,...), inequalities
  - Linear or non-linear fragments
- Uninterpreted functions
- Arrays
- Inductive datatypes, records
- Bitvectors (things like a 32-bit integer)
  - Usual arithmetic
  - Bitwise logical operations (and, or, xor, ...)
- Quantifiers

Huge variation, even within individual tools.
Affects input language, performance, completeness[1].
Standard interchange language, SMT-LIB.

[1] Solvers may report SATISFIABLE, UNSATISFIABLE, or UNKNOWN

Example theory: equality[2]

Suppose we want to handle equality generically:

$$(a = b \vee a = c) \wedge b = d \wedge a \neq d$$

where a, b, c, d are variables of some other theory.

We can represent groups of equal variables efficiently in sets which have a representative element. For example, if we pick $a = b$ the sets might be (taking b to be the representative)

$$\{a, \mathbf{b}\} \quad \{\mathbf{c}\} \quad \{\mathbf{d}\}$$

and when comparing variables we consult the representatives, e.g.,

$$a \neq d \quad \text{is mapped to} \quad b \neq d$$

We also keep a set of picked disequalities, to check we don't violate them.

[2] This example is derived a tutorial by the authors of Z3

Example of the example equality theory

$$(a = b \vee a = c) \wedge b = d \wedge a \neq d$$

where a, b, c, d are variables of some other theory.

Action	Sets	Disequalities	Remainder
Pick $a = b$	$\{a, \mathbf{b}\}$ $\{\mathbf{c}\}$ $\{\mathbf{d}\}$	$\emptyset$	$b = d \wedge b \neq d$
Pick $a \neq d$	$\{a, \mathbf{b}\}$ $\{\mathbf{c}\}$ $\{\mathbf{d}\}$	$\{b \neq d\}$	$b = d$
Pick $b = d$	$\{a, \mathbf{b}, d\}$ $\{\mathbf{c}\}$	$\{b \neq b\}$	Conflict
Backtrack			
Pick $a = c$	$\{a, \mathbf{c}\}$ $\{\mathbf{b}\}$ $\{\mathbf{d}\}$	$\emptyset$	$b = d \wedge c \neq d$
Pick $a \neq d$	$\{a, \mathbf{c}\}$ $\{\mathbf{b}\}$ $\{\mathbf{d}\}$	$\{c \neq d\}$	$b = d$
Pick $b = d$	$\{a, \mathbf{c}\}$ $\{b, \mathbf{d}\}$	$\{c \neq d\}$	Satisfied

These sets-with-representatives are implemented by the *union-find* algorithm.

Examples, counterexamples and proving

We've seen that SMT solvers take some formula over some variables,

$$\phi(x_1, \ldots, x_n)$$

and finds values for $x_1, \ldots, x_n$, i.e., an example where ϕ is true.

If we negate the formula,

$$\neg\phi(x_1, \ldots, x_n)$$

then the SMT solver looks for counterexamples $x_1, \ldots, x_n$ of ϕ.

If the solver reports UNSATISFIABLE for $\neg\phi$, then we've proved

$$\forall x_1, \ldots, x_n.\ \phi(x_1, \ldots, x_n).$$

Useful for integration into systems like Isabelle.

SMT solving in practice

Lots of tools: Z3, CVC4, Yices, Alt-Ergo, ...

Lots of applications:

- Test case generation
- Program verification (in several styles)
- Translation validation (checking compiler optimisations)
- Planning tasks / scheduling
- Mathematics
- Interactive theorem proving assistance
- ...

Test case generation for an instruction set model

I make mistakes when writing software, so I like software verification.

Test case generation for an instruction set model

I make mistakes when writing software, so I like software verification.

But I also make similar mistakes when writing specifications, so I like testing specifications.

Test case generation for an instruction set model

I make mistakes when writing software, so I like software verification.

But I also make similar mistakes when writing specifications, so I like testing specifications.

I even like to do it when I use other people's specifications.

Test case generation for an instruction set model

The ARM Cortex-M0 is a small microcontroller, typically used in low power devices like games controllers.

It has very predictable behaviour, so we wanted to use it in experiments with verifying software execution times.

Anthony Fox in Cambridge cut down his full ARM model to the M0's instructions, added timing information.

But even the simple M0 is complicated: there's a pipeline working on three instructions at once, perhaps there's interference between instructions?

Most testing looks at single instructions — I wanted to test sequences.

The model

The Cortex-M0 instruction set model is specified as functions in the HOL4 interactive theorem prover.

$$\mathsf{NextStateM0} : \mathsf{m0\_state} \rightarrow \mathsf{m0\_state\ option}$$

The state contains registers, memory, flags, ...

The model

There's also a handy function that gives us a theorem about what the model does on any instruction.

For example, `ldr r3,[r0]` which loads register 3 from the address in register 0:

```
[Aligned (s.REG RName_PC,2),  s.MEM (s.REG RName_PC) = 3w,
 Aligned (s.REG RName_0, 4),  s.MEM (s.REG RName_PC + 1w) = 104w]
⊢ NextStateM0 s = SOME (s with
      <| REG := (RName_PC =+ s.REG RName_PC + 2w)
                ((RName_3 =+ s.MEM (s.REG RName_0 + 3w) @@
                             s.MEM (s.REG RName_0 + 2w) @@
                             s.MEM (s.REG RName_0 + 1w) @@
                             s.MEM (s.REG RName_0)) s.REG);
         count := s.count + 2 |>)
```

The hypotheses need to hold for the instruction to execute successfully.

Fuzz testing the model

1. Pick some random instructions
2. run them on the model
3. run them on a chip
4. compare the results

Fuzz testing the model

1. Pick some random instructions
2. run them on the model
3. run them on a chip
4. compare the results

Problem: they'll usually crash

Fuzz testing the model

1. Pick some random instructions
2. run them on the model
3. run them on a chip
4. compare the results

Problem: they'll usually crash

Need to carefully pick values for registers, memory, flags to make them work.

Picking the registers, memory, flags, . . .

The model gave us constraints like

```
Aligned (s.REG RName_0, 4)
```

But when we chain together instructions, it gets complicated:

```
Aligned
  (s.REG RName_2 +
   sw2sw
     (s.MEM (s.REG RName_1 + s.REG RName_2 + 1w) @@
      s.MEM (s.REG RName_1 + s.REG RName_2)) <<~
   w2w ((7 >< 0) (s.REG RName_2)),4)
```

featuring bitvector addition, concatenation, sign extension, shifting and alignment.

A bitvector solver isn't enough: need functions for registers and memory, plus basic logic. SMT

Translation from HOL to SMT

The SMT solver doesn't know what `Aligned` means.
In HOL everything is defined in terms of previous definitions:

```
Aligned (w,n) = (w = n2w (n * (w2n w DIV n)))
```

A naïve translation to SMT involves three theories:

- equality, bitvectors and natural numbers

Equality works well with other theories, but the SMT solver doesn't really support translating between bitvectors and naturals.

Translation from HOL to SMT

The SMT solver doesn't know what `Aligned` means.
In HOL everything is defined in terms of previous definitions:

```
Aligned (w,n) = (w = n2w (n * (w2n w DIV n)))
```

A naïve translation to SMT involves three theories:

- equality, bitvectors and natural numbers

Equality works well with other theories, but the SMT solver doesn't really support translating between bitvectors and naturals.

An important part of the work involved is finding alternative translations:

`Aligned (a, 2)`	iff	bottom bit of `a` is zero
`Aligned (a, 4)`	iff	bottom 2 bits of `a` are zero
...		

These stay within the bitvector theory

Does it work?

1. Pick a random instruction sequence
2. Get formulae from model necessary to avoid crash
3. Use SMT to fill in memory, registers, flags from formulae

```
reg RName_1     = 0x12345678
mem 0x01000000 = 0x23
...
```

4. Run and compare

Found a couple of bugs in model, timing anomalies in chip!

But...

Does it work?

1. Pick a random instruction sequence
2. Get formulae from model necessary to avoid crash
3. Use SMT to fill in memory, registers, flags from formulae

```
reg RName_1    = 0x12345678
mem 0x01000000 = 0x23
...
```

4. Run and compare

Found a couple of bugs in model, timing anomalies in chip!

But...

Beware of daemonic behaviour!

- Automated theorem provers don't know what cheating is

I originally left out formulae to prevent self-modifying code
$\Rightarrow$ so the SMT solver made some

High coverage test case generation for programs

For the processor specification we:

- Don't know how the chip was implemented
- so start from random instructions
- then use SMT to 'fill in the blanks' so that they work

But if we want to test a Java program, we

- have source code
- can monitor execution
- can measure coverage

We want to look at the execution of existing test cases to make new ones.

Concolic testing

Concolic is a portmanteau:

- Trace the concrete execution of an existing test case
- perform a corresponding symbolic execution to work out what to change to divert execution

Symbolic execution remembers how values were calculated

- But unguided symbolic execution is too expensive
- Follow the concrete execution instead
- So can ask an SMT solver for a new input that follows a different path

Concolic testing example

```
private static int foo(int y) { return 2*y; }
public static void testme(int x,int y){
    int z = foo(y);
    if(z==x){
        if(x>y+10){
            System.err.println("Error"); // ERROR
        }
    }
}
```

Existing test case: run `testme(2,1)`

Concrete execution:

`z = foo(1) = 2`

`z == x` is $2 == 2$ is true

`x>y+10` is $2 > 1 + 10$ is false

Terminates successfully

Concolic testing example

```
private static int foo(int y) { return 2*y; }
public static void testme(int x,int y){
    int z = foo(y);
    if(z==x){
        if(x>y+10){
            System.err.println("Error"); // ERROR
        }
    }
}
```

Existing test case: run `testme(2,1)`

Concrete execution:	Symbolic execution:
`z = foo(1) = 2`	`z` $=$ `foo(y)` $=$ `2*y`
`z == x` is $2 == 2$ is true	`z == x` is `2*y == x`
`x>y+10` is $2 > 1 + 10$ is false	`x>y+10`
Terminates successfully	`z=2*y, 2*y == x, !(x>y+10)`

Symbolic execution follows the concrete execution

Concolic testing example

```
private static int foo(int y) { return 2*y; }
public static void testme(int x,int y){
    int z = foo(y);
    if(z==x){
        if(x>y+10){
            System.err.println("Error"); // ERROR
        }
    }
}
```

Existing test case: run `testme(2,1)`

Concrete execution:	Symbolic execution:
`z = foo(1) = 2`	`z` = `foo(y)` = `2*y`
`z == x` is $2 == 2$ is true	`z == x` is `2*y == x`
`x>y+10` is $2 > 1 + 10$ is false	`x>y+10`
Terminates successfully	`z=2*y, 2*y == x, !(x>y+10)`

Symbolic execution follows the concrete execution

Call `2*y == x, !(x>y+10)` path condition

Concolic testing example

```
private static int foo(int y) { return 2*y; }
public static void testme(int x,int y){
    int z = foo(y);
    if(z==x){
        if(x>y+10){
            System.err.println("Error"); // ERROR
        }
    }
}
```

Path condition: `2*y == x, !(x>y+10)`

Want to explore one of the branches:

- Pick one of the clauses in the path condition
- Invert it
- Get rid of irrelevant clauses afterwards
- Ask the SMT solver when it's true

Concolic testing example

```
private static int foo(int y) { return 2*y; }
public static void testme(int x,int y){
    int z = foo(y);
    if(z==x){
        if(x>y+10){
            System.err.println("Error"); // ERROR
        }
    }
}
```

Path condition: `2*y == x, !(x>y+10)`

- Pick `!(x>y+10)`
- New path condition: `2*y == x, x>y+10`
- SMT solver says: try `testme(22,11)`

New test case finds the "Error".

Concolic testing software

Very active research area:

- KLEE: runs LLVM bitcode
- CATG: Java system (source of example)
- SAGE: Microsoft
  - Serious industrial use
  - Finds security bugs in input parsers
- ...

Concolic testing software

Very active research area:

- KLEE: runs LLVM bitcode
- CATG: Java system (source of example)
- SAGE: Microsoft
  - Serious industrial use
  - Finds security bugs in input parsers
- ...

In general there are too many possible paths in the program

- Develop heuristics to guide selection of clauses to invert
- How many times do you want to go around a loop?

Concolic testing software

Very active research area:

- KLEE: runs LLVM bitcode
- CATG: Java system (source of example)
- SAGE: Microsoft
  - Serious industrial use
  - Finds security bugs in input parsers
- ...

In general there are too many possible paths in the program

- Develop heuristics to guide selection of clauses to invert
- How many times do you want to go around a loop?

Also, path conditions must eventually become impossible to solve.

Concolic testing software

Very active research area:

- KLEE: runs LLVM bitcode
- CATG: Java system (source of example)
- SAGE: Microsoft
  - Serious industrial use
  - Finds security bugs in input parsers
- ...

In general there are too many possible paths in the program

- Develop heuristics to guide selection of clauses to invert
- How many times do you want to go around a loop?

Also, path conditions must eventually become impossible to solve.

My current favourite trick: Imperial College researchers are attempting to use a variant of concolic testing to fix unparsable documents.

Summary

Satisfiability Modulo Theories solvers are

- advanced automated theorem provers
- diverse, but generic
- already in widespread research use
- starting to appear in industrial use

I've sketched out two test case generation techniques using them:

1. To fill in background details to make tests work
2. To generate new tests from existing ones