SAT/SMT Solvers, Axiomatic Semantics

Outline
- SAT/SMT solvers
- Axiomatic semantics
  - IMP
  - Semantics
  - Verification condition generation
    - What we called “backwards reasoning” in PSoft
- SMT-LIB and Z3

Logical Reasoning
- A lot of recent PL/SE research uses some form of automated logical reasoning
  - Non-standard type inference (SAT, MaxSAT)
  - Software verification (e.g., Dafny uses SMT)
  - Symbolic execution (SMT)
- Program synthesis

Reading
- If you are interested in the field

SAT Solvers
- Decide whether a propositional logic formula is satisfiable (sat) or unsatisfiable (unsat)
  - E.g., \((p \lor q) \rightarrow lp\) is sat or unsat?
  - E.g., \((p \rightarrow q) \rightarrow !(p \land \neg q)\) is sat or unsat?
- A lot of work on SAT solvers
  - Boolean satisfiability is a fundamental NP-complete problem
  - A good SAT solver can “solve” many problems!!!

Variations of SAT
- MaxSAT: Given a formula in Conjunctive Normal Form (CNF), find an assignment that maximizes number of satisfied clauses
  - E.g., \((p \lor q) \land lp \land \neg lq\)
- Partial MaxSAT
  - Hard clauses: clauses that must be satisfied
  - Soft clauses: clauses that may remain unsatisfied
  - Partial MaxSAT: find an assignment that satisfies all hard clauses and maximizes number of satisfied soft clauses
Variations of SAT

- Weighted Partial MaxSAT
  - Hard clauses: clauses that must be satisfied
  - Soft clauses: clauses that may remain unsatisfied
  - Weights: soft clauses have weights
  - Weighted Partial MaxSAT: find an assignment that satisfies all hard clauses and maximizes the weight of satisfied soft clauses
    - E.g., suppose \( (p \lor q) \) is a hard clause, \( lp \) is a soft clause with weight 2, and \( tq \) is soft with weight 1
    - What assignment maximizes \( (p \lor q) \land lp \land tq \)

SMT Solvers

- Satisfiability Modulo Theories (SMT) extends assertions/satisfiability beyond propositional logic
  - Extends with background theories
  - Theory of equality: \( x \neq y \land f(x) = f(y) \)
  - Theory of arithmetic: \( x < y \land (x < y + 0) \)
  - Theory of select/store (arrays): Hoare triple
    \( \{ b.f = 5 \} . a.f = 5 . \text{store}(f,a,5) \Rightarrow \text{select}(f,b) + \text{select}(f,a) = 10 \)

Axiomatic Semantics

- Consider program fragment
  \[ t = x - y; \]
  \[ \text{while} (t > 0) \{ \]
  \[ x = x - 1; \]
  \[ y = y + 1; \]
  \[ t = t - 1; \]
  \[ \} \]
  - We are interested in proving these claims:
    - When \( x > y \), program terminates
    - When \( x > y \), values of \( x \) and \( y \) are swapped

Axiomatic Semantics

- Not easy to prove using theories we studied so far
  - Dataflow analysis
  - Abstract interpretation
  - Type systems
  - E.g., neither gives a convenient way of encoding the assumption \( x > y \) into reasoning and semantics

Axiomatic Semantics

- Key idea:
  - Hoare triple: \( \{ P \} . \text{code} . \{ Q \} \)
  - Semantics of a program construct is defined in terms of logical assertions and the effect of the construct on these assertions
  - Language of assertions
  - Deductive reasoning
### History
- Early years: "An Axiomatic Basis for Computer Programming" by C.A.R. Hoare, 1969
  - Great optimism by Tony Hoare and Edsger Dijkstra
  - Bugs will be a thing of the past!
- If you can prove programs correct, no need to even test!
- Middle ages: "Social Processes and Proofs of Theorems and Programs", De Millo, Lipton and Perlis, 1979
  - Proofs in math work because there is a social process
  - Program proofs are too boring for social process to form
  - Programs change too fast and proofs are too brittle
- A renaissance: Z3, other automated logical reasoning tools
- Some success stories from Microsoft
- There is some optimism again...

### You Already Know The Basics 😊
- Hoare triples \[ \{ P \} \text{stmt} \{ Q \} \]
  - \( P \) is the precondition, \( Q \) is the postcondition
  - Triple is a logical formula: if \( P \) holds before \( \text{stmt} \) execution and \( \text{stmt} \) terminates, then \( Q \) holds afterwards
- E.g., \( \{ x > -1/2 \} x = x + 3 \{ x > 5/2 \} \)
- \( \{ P \} \text{stmt} \{ Q \} \): partial correctness assertion
- \( \{ P \} \text{stmt} \{ Q \} \): total correctness assertion
- We will consider partial correctness

### The IMP Language
- **Expressions**
  \[ e ::= n \mid x \mid e_1 + e_2 \mid e_1 = e_2 \]
- **Commands** (i.e., statements, change state):
  \[ c ::= x := e \mid c_1 ; c_2 \mid \text{if } (e) \text{ then } c_1 \text{ else } c_2 \mid \text{while } (e) \text{ do } c \mid \text{skip} \]
- A big-step operational semantics
- Judgments for expressions: \((e, \sigma) \to n\)
- Judgments for commands: \((c, \sigma) \to \sigma'\)

### Operational Semantics
\[
\begin{align*}
(e, \sigma) & \to n & (c_1, \sigma) & \to \sigma' \quad (c_2, \sigma') & \to \sigma'' \\
(x := e, \sigma) & \to \sigma[x \leftarrow n] & (c_1 ; c_2, \sigma) & \to \sigma'' \\
(e, \sigma) & \to \text{True} & (c_1, \sigma) & \to \sigma' & (e, \sigma) & \to \text{False} & (c_2, \sigma) & \to \sigma'' \\
& & (\text{if } (e) \text{ then } c_1 \text{ else } c_2, \sigma) & \to \sigma' & (\text{if } (e) \text{ then } c_1 \text{ else } c_2, \sigma) & \to \sigma'' \\
(e, \sigma) & \to \text{True} & (c, \sigma) & \to \sigma' & (\text{while } (e) \text{ do } c, \sigma) & \to \sigma'' \\
& & & & (\text{while } (e) \text{ do } c, \sigma) & \to \sigma'' \\
(e, \sigma) & \to \text{False} & (e, \sigma) & \to \sigma' \\
& & (\text{while } (e) \text{ do } c, \sigma) & \to \sigma'
\end{align*}
\]

### Meaning of Assertions
- \( \{ P \} c \{ Q \} \)
  - Let \( P \) be logical assertion
    - E.g. \( x < y \) or \( x + y = 5 \)
    - \( P \) "references" state \( \sigma \)
  - \( \sigma \vdash P \) (read: \( \sigma \) entails \( P \)) means that assertion \( P \) holds on state \( \sigma \)
    - E.g., \( \sigma = [x \leftarrow 5, y \leftarrow 10, z \leftarrow 0] \mid x < y \)
    - Does \( \sigma' = [x \leftarrow 10, y \leftarrow 10, z \leftarrow 0] \mid x < y ? \)
  - \( \{P\} c \{Q\} \) has the following meaning:
    - \( \forall \sigma, \sigma'. (\sigma \vdash P \wedge (c, \sigma) \to \sigma') \implies \sigma' \vdash Q \)

### Static Semantics
\[
\begin{align*}
\{ P[e/x] \} & \xleftarrow{=} \{ P \} \\
\{ P \} c_1 \{ Q \} & \xrightarrow{=} \{ P \} c_2 \{ R \} \\
\{ P \land e \} & \xrightarrow{=} \{ P \} \{ c_1 \} \{ c_2 \} \\
\{ P \text{ if } (e) \text{ then } c_1 \text{ else } c_2 \} & \xrightarrow{=} \{ P \} \{ c_1 \} \{ c_2 \} \\
\{ P \text{ while } (e) \text{ do } c \} & \xrightarrow{=} \{ P \text{ while } (e) \} \{ c \} \\
\end{align*}
\]

Rule of consequence:
\[
\begin{align*}
P \Rightarrow P' & \xrightarrow{=} \{ P' \} c \{ Q' \} \\
Q & \Rightarrow Q \\
\{ P \} & \xrightarrow{=} \{ P \} c \{ Q \}
\end{align*}
\]
Soundness

- For each Hoare triple \( \{ P \} c \{ Q \} \) deduced by the static semantics
  \( \forall \sigma, \forall \sigma'. (\sigma |- P \land (c, \sigma) \rightarrow \sigma' |- Q) \)

- Notice how in each one of our theories, AI, type systems, and AS we have
  - Dynamic semantics
  - Static semantics
  - Soundness (connecting the two)

Example

\( \{ x>y \text{ and } x=x_0 \text{ and } y=y_0 \} \)
\[ t = x - y; \]
\[
\text{while } (t > 0) \{ \]
\[ x = x - 1; \]
\[ y = y + 1; \]
\[ t = t - 1; \]
\[
\} \]
\[
\{ x=y_0 \text{ and } y = x_0 \} \]

Example

\( P \Rightarrow P' \{ P' \} c \{ Q' \} Q' \Rightarrow Q\)
\[ \{ P \} \text{ while } (e) \text{ do } c \{ P \land \neg e \} \]
\[
\{ x>y \text{ and } x=x_0 \text{ and } y=y_0 \} \]
\[ t = x - y; \]
\[
\text{while } (t > 0) \{ \]
\[ x = x - 1; \]
\[ y = y + 1; \]
\[ t = t - 1; \]
\[
\{ x=y_0+t \text{ and } y=x_0-t \text{ and } t \geq 0 \} \Rightarrow \{ x=y_0+t \text{ and } y=x_0-t \text{ and } t \geq 0 \text{ and } \neg (t>0) \} \Rightarrow \{ x = y_0 \text{ and } y = x_0 \} \]
Example

\[ \{P \land e\} \text{c} \{P\}\]

\{ x>y \land x=x_0 \land y=y_0 \}

t = x - y;

\{ x=y_0+t \land y=x_0-t \land t \geq 0 \}

while \((t > 0)\) {
\{ x=y_0+t \land y=x_0-t \land t \geq 0 \land t>0 \} \Rightarrow \{ x=y_0+t \land y=x_0-t \land t-1 \geq 0 \}

x = x - 1;

\{ x=y_0+t-1 \land y=x_0-t+1 \land t-1 \geq 0 \}

y = y + 1;

\{ x=y_0+t-1 \land y=x_0-t+1 \land t-1 \geq 0 \}

t = t - 1;

\{ x=y_0+t \land y=x_0-t \land t \geq 0 \}

\{ x=y_0+t \land y=x_0-t \land t \geq 0 \land !(t>0) \} \Rightarrow \{ x = y_0 \land y = x_0 \} \]

Outline

- SAT/SMT solvers
- Axiomatic semantics
  - IMP
  - Semantics
  - Verification condition generation
    - What we called “backwards reasoning” in PSoft
- SMT-LIB

Weakest Precondition

\[
\begin{align*}
\text{wp}(x:=e,Q) &= Q[e/x] \\
\text{wp}(c_1;c_2,Q) &= \text{wp}(c_1,\text{wp}(c_2,Q)) \\
\text{wp}(\text{if} (e) \text{ then } c_1 \text{ else } c_2, Q) &= \\
&= e \Rightarrow \text{wp}(c_1,Q) \land \text{le} \Rightarrow \text{wp}(c_2,Q) \\
&= e \wedge \text{wp}(c_1,Q) \lor \text{le} \wedge \text{wp}(c_2,Q) \\
\text{wp}(\text{while} (e) \text{ do } c,Q) &= \\
W &= e \Rightarrow \text{wp}(c,W) \land \text{le} \Rightarrow Q
\end{align*}
\]

Verification Condition

- Instead of weakest precondition we compute verification condition (vc). Stronger

\[
\begin{align*}
\text{vc}(\text{while} (e) \text{ do } c,Q) &= \\
&= \text{Inv} \land \text{Inv} \Rightarrow (e \Rightarrow \text{vc}(c,\text{Inv}) \land \text{le} \Rightarrow Q) \\
\text{or}
\end{align*}
\]

The Process

- We want to prove \( \{ P \} \text{c} \{ Q \} \)

- First, we generate verification condition \( P' \) using rules for backwards reasoning

\[
P' = \text{vc}(c,Q) \Rightarrow \text{wp}(c,Q)
\]

- Next, we assert \( P \Rightarrow P' \)

\[
P \Rightarrow P' \Rightarrow \text{vc}(c,Q) \Rightarrow \text{wp}(c,Q) \Rightarrow \text{wp}(c,Q) \Rightarrow c_1; \text{wp}(c,Q) \Rightarrow \text{c} \{ Q \} \Rightarrow \{ P \} \text{c} \{ Q \}
\]
Example

```c
i = 5;
while (i > 0) {
    Inv = { i ≥ 0 }
    i = i – 1;
} { i = 0 }
```

vc breaks into following assertions:

- True => wp(i=5; { i ≥ 0 })

SMT-LIB

- SMT-LIB is a language for specifying input to SMT solvers (e.g., Z3)
  - `(declare-const x Int)` declare an integer constant `x`
  - `(assert (> x 0))` add `x>0` to known facts
  - `(check-sat)` checks if there exist an assignment that makes all known facts true; returns `(sat)` or `(unsat)`
  - `(get-model)` print this assignment

Another Example

```c
{x >= 0 }
i = x;
z = 0;
while (i != 0) {
    z = z + 1;
i = i - 1;
} { x = z }
```

Example

```c
requires: x == 1 || x == -2
ensures: y == 0
```

SMT-LIB

- `(declare-const x Int)`
- `(declare-fun f (Int Bool) Int)`
- `(assert (> a 10))`
- `(assert (< (f a true) 100))`
- `(check-sat)`

Example

```c
vc(...) = ???
x(...y=0) = ??
(x=1 or x=-2) =>
((x>0 and x*x-1=0) or
 (x<=0 and x+4*x=0))
```

SMT-LIB code:

- `(declare-const x Int)`
- `(asssert (and (or (x 1) (x -2)))` (not (or (and (<= x 0) ((+ (+ x 4) x) 0))
  (and (> x 0) (not ("* x) 0))))))
- `(check-sat)`
- `(get-model)`
Example

requires: x == 1 || x == -5
ensures: y == 0
{
    y = x + 4;
    if (x > 0) {
        y = x*x - 1;
    } else {
        y = y + x;
    }
}

SMT-LIB code:
(declare-fun x (Int))
(assert (and (or (= x 1) (= x -5))
    (not (or (and (<= x 0) (= (+ (+ x 4) x) 0))
        (and (> x 0) (= (- (* x x) 1) 0)))))
(check-sat)

Another Example

Is this formula valid?
(x>0 and x+5 > 5) or
(x<=0 and (x=0 => x + x + 5 = 5))

SMT-LIB code:
(declare-fun x (Int))
(assert (not (and (> x 0) (> (+ x 5) 5))))
(assert (not (and (<= x 0) (or (not (= x 0)) (= (+ (+ x x) 5) 5)))))
(check-sat)

Homework I wanted to do…

- Buffer overflow bugs are dangerous
  - A malicious user can hijack program execution
- We were going to write a Ghidra plugin that finds buffer overflows in MBE CTF examples
- Verification condition generation
- Hook into Z3 to verify NO OVERFLOW or find the input that triggers overflow
- Take MBE!

MBE fill_bowl Example

void fill_bowl(char* ingredients, char* bowl, int bowl_size)
{
    printf("How many ingredients do you want to put in the bowl (max %u)\n", (ulong)(bowl_size-1));
    int number;
    scanf("%u", &number);
    if (number > bowl_size)
        number = bowl_size - 1;
    // Copy at most bowl_size characters into the buffer
    for (int i=0; i<= number; i++)
        bowl[i] = ingredients[i]; // buffer overflow.
}

MBE fill_bowl, Ghidra Decompiled

```
int local_24, local_25, i, number;
    i = bowl.size;
    printf("...", (ulong)(bowl.size - 10), (ulong)(bowl.size - 10));
    scanf("%u", &local_25);
    if (i < local_25) {
        local_20 = i + -1;
    }
    local_24 = 0;
    while (local_24 <= local_20) {
        bowl[local_24] = ingredients[local_24];
        local_24 = local_24 + 1;
    }
    return;
```

MBE fill_bowl, Ghidra Decompiled

```
Verify F: ( 40 < local_20 => 40 - 1 <= 40 - 1 AND
40 >= local_20 <= 40 - 1 )
```

Verify { local_20 <= bowl.size - 1 }

```
Verify { local_20 <= bowl.size - 1 }
```