Abstract Satisfaction

Vijay D’Silva, Leopold Haller, Daniel Kroening
A Tale of Two
Software Verification DPhils
Abstract Interpretation based Program Analysis

Error

Program traces

Abstract representation

Error states do not overlap abstract representation, hence program is safe

Concrete Domain

Interpreter

Abstract Domain

Test for all inputs and memory states

Abstract analysis
Lattice of Boolean Constants

Abstract interpretation operates over lattices
Domain of Constants

Constant Propagation

\[ \text{Var} \rightarrow \text{IntVals} \cup \{?\} \]

Analyse by applying abstract transformers

Efficient, but imprecise
Domain of Intervals

\[ \text{Var} \mapsto \{ [l, u] \mid l, u \in \text{IntVals} \} \]

\begin{align*}
    a, b & \in [1, 1] \\
    a & \in [4, 10], \\
    b & \in [1, 1] \\
\end{align*}

Efficient, but imprecise
How can I get abstract domains to be more precise?
Build a logical formula that encodes program semantics

\[ isTrace(t) \land error(t) \]

Solve satisfiability: Does there exist a \( t \) that makes the above formula true.

Fast SAT solvers exist that can solve this question.
SAT encoding

\[ a_0 = 1 \land b_0 = a_0 \land (c_0 \rightarrow a_1 = 4) \land (\neg c_0 \rightarrow a_1 = 10) \land (a_1 \leq 0 \lor b_0 \leq 0) \]

Translate inequalities and equalities to circuits

Precise, but not scalable
How can I get my SAT solver to be more efficient?
Our initial project

Let’s *combine* SAT solving and abstract interpretation to achieve both *efficiency* and *precision*?
Partial assignments in SAT

The main data-structure in a SAT solver is a partial assignment from variables to truth values.

This assignment is extended using deductions and decisions.

\[
\begin{align*}
    x & \mapsto 1 \\
    y & \mapsto 1 \\
    y & \mapsto 0 \\
\end{align*}
\]

\[
\begin{align*}
    x & \mapsto 1 \\
    y & \mapsto 1 \\
    x & \mapsto 0 \\
    y & \mapsto 0 \\
\end{align*}
\]

\[
\begin{align*}
    x & \mapsto 0 \\
    y & \mapsto 0 \\
\end{align*}
\]

\[
\begin{align*}
    x & \mapsto 0 \\
    y & \mapsto 0 \\
\end{align*}
\]
SAT operates over a lattice

SAT operates the Boolean constants lattice
The Unit Rule

\[ p \mapsto t \]
\[ q \mapsto f \]
\[ r \mapsto f \]

\[ \ldots \land (\neg p \lor q \lor r \lor \neg w) \land \ldots \]
The Unit Rule

$p \leftrightarrow t$
$q \leftrightarrow f$
$r \leftrightarrow f$

\[ \ldots \land (\neg p \lor q \lor r \lor \neg w) \land \ldots \]
The Unit Rule

Unit Rule

\[ p \mapsto t \]
\[ q \mapsto f \]
\[ r \mapsto f \]

\[ \ldots \land (\neg p \lor q \lor r \lor \neg w) \land \ldots \]
The Unit Rule

unit_rule

\[ \ldots \wedge (\neg p \lor q \lor r \lor \neg w) \wedge \ldots \]
The Unit Rule

\[ \ldots \land (\neg p \lor q \lor r \lor \neg w) \land \ldots \]

\text{Unit Rule}

\[ \begin{align*}
p & \rightarrow t \\
q & \rightarrow f \\
r & \rightarrow f \\
w & \rightarrow f
\end{align*} \]
The Unit Rule

\[ p \rightarrow t \]
\[ q \rightarrow f \]
\[ r \rightarrow f \]

\[ \ldots \land (\neg p \lor q \lor r \lor \neg w) \land \ldots \]

\[ w \rightarrow f \]

\[ \text{if}(!p \lor q \lor r \lor !w) \]
\[ \{ \]
\[ \ldots \]
\[ \} \]

Thursday, 17 November 11
The Unit Rule

The unit rule is the **best abstract transformer** over the lattice!

\[ \text{if}(\neg p \lor q \lor r \lor \neg w) \]

\[ \{ \ldots \} \]
Decisions

No deductions are possible on the following formula.

\[ \phi = (w \lor q) \land (\neg w \lor q) \]

Hence a decision is made:

\[ q \mapsto \text{false} \]

From which both of the following can be deduced:

\[ w \mapsto \text{true} \quad \text{and} \quad w \mapsto \text{false} \]

The solver backtracks and learns that q must be true, essentially, we expanded the formula into

\[ (q \land \phi) \lor (\neg q \land \phi) \]
Trace partitioning

Trace partitioning is an well-known refinement technique in abstract interpretation

```c
void foo(int a, int x) {
    if(a < 0)
        x = 1;
    else
        x = -1;
    assert(x != 0);  \( x \in [-1, 1] \) too imprecise!
}
```
Trace partitioning

Trace partitioning is an well-known refinement technique in abstract interpretation

```c
void foo(int a, int x) {
    if(a < 0)
        x = 1;
    else
        x = -1;
    assert(x != 0);
    x ∈ [−1, 1]  too imprecise!
}
```

Apply partitioning:

```c
void foo_part(int a, int x) {
    if(a < 0)
        foo(a,x);  x ∈ [1, 1]  safe
    else
        foo(a,x);  x ∈ [−1, −1]
}
```
Trace partitioning

Trace partitioning is an well-known refinement technique in abstract interpretation

```c
void foo(int a, int x) {
    if(a < 0)
        x = 1;
    else    
        x = -1;
    assert(x != 0);  x ∈ [−1, 1]  too imprecise!
}
```

Apply partitioning:

```c
void foo_part(int a, int x) {
    if(a < 0)
        foo(a,x);  x ∈ [1, 1]  safe
    else
        foo(a,x);  x ∈ [−1, −1]
}
```

Decisions are a well-known program analysis technique!
Summary: SAT = AI

Modern SAT solvers are abstract interpreters

The SAT architecture is an abstract interpreter architecture that automatically and intelligently refines a base domain.
SAT over Interval Domain

Fig. 2. Execution times of Astrée, CBMC, and CDFL; wrong results set to 3600s

Naive implementation of SAT(Intervals) applied to numeric program verification benchmarks.

On average ca. 200x faster than SAT, significantly more precise than mature abstract interpreters.
How can I get my SAT solver to be more efficient?

Choose a domain that’s better suited to your problem than the Boolean constants domain!
How can I get abstract domains to be more precise?

Wrap them in the SAT architecture!
Thanks for your attention