SMT-Style Program Analysis with Value-based Refinements

Vijay D’Silva    Leopold Haller    Daniel Kröning

NSV-3
July 15, 2010
Outline

Imprecision and Refinement in Abstract Interpretation

SAT Style Abstract Analysis

Value-based Refinement for Intervals
Imprecision in Abstract Interpretation

- Abstract interpretation sound but not complete.

- Incompleteness manifests in imprecision during the analysis.
Imprecision in Abstract Interpretation

- Abstract interpretation sound but not complete.

- Incompleteness manifests in imprecision during the analysis.

Example: Domain of Intervals
Imprecisions in the Domain

Imprecision in join

```plaintext
x:=*;
if(x > 5)
y := -1;
else
  y := 1;
assert(y != 0);
```

The disjunction $y = 1 \lor y = -1$ cannot be expressed as an interval.
**Imprecisions in the Domain**

**Imprecision in join**

```plaintext
x := *;
if (x > 5)
    y := -1;  \rightarrow \quad y \in [-1, -1], x \in [6, \infty]
else
    y := 1;

assert(y != 0);
```

The disjunction $y = 1 \lor y = -1$ cannot be expressed as an interval.
Imprecisions in the Domain

Imprecision in join

\begin{align*}
x &:= \ast; \\
\text{if}(x > 5) &\quad y := -1; \quad \rightarrow \quad y \in [-1, -1], x \in [6, \infty] \\
\text{else} &\quad y := 1; \quad \rightarrow \quad y \in [1, 1], x \in [-\infty, 5] \\
\text{assert}(y \neq 0);
\end{align*}
Imprecisions in the Domain

Imprecision in join

\[
\begin{align*}
  x & := *; \\
  \text{if} (x > 5) & \quad y := -1; \quad y \in [-1, -1], x \in [6, \infty] \\
  \text{else} & \quad y := 1; \quad y \in [1, 1], x \in [-\infty, 5] \\
  \text{assert}(y \neq 0); & \quad y \in [-1, 1]
\end{align*}
\]

The disjunction \( y = 1 \lor y = -1 \) cannot be expressed as an interval.
Imprecisions in the Domain

Imprecision in transformer

\[
x := y; \\
\text{if}(x > 5) \\
\text{assert}(y > 5);
\]
Imprecisions in the Domain

Imprecision in transformer

\[ \begin{align*}
  x &:= y; \\
  \text{if}(x > 5) & \quad \rightarrow \quad T \\
  \text{assert}(y > 5); \end{align*} \]

Intervals cannot express relational information.
Imprecisions in the Domain

Imprecision in transformer

\[\begin{align*}
x & := y; \\
\text{if}(x > 5) \\
\text{assert}(y > 5); & \quad \rightarrow \quad x \in [6, \infty]
\end{align*}\]
Imprecisions in the Domain

Imprecision in transformer

\[ x := y; \quad \rightarrow \quad T \]
\[ \text{if}(x > 5) \]
\[ \text{assert}(y > 5); \quad \rightarrow \quad x \in [6, \infty] \]

Intervals cannot express relational information.
Imprecisions in the Analysis

Imprecision in widening

while(x < 50000) {
    x++;
    if(y < x)
        y++;
}

Precision can be lost in the analysis
Refinement of widening studied by, e.g., Gulavani et. al (TACAS 2008), Wang et al. (CAV 2007)
Refining Abstract Domains

Global domain refinement

Global domain refinement potentially expensive.

How can we locally refine an abstract domain?
Refining Abstract Domains

- Global domain refinement
- More powerful domain
  - Octagons
  - Polyhedra
  - ...
Refining Abstract Domains

- Global domain refinement
- More powerful domain
  - Octagons
  - Polyhedra
- Disjunctive completion
  - $\leq 0$
  - $\neq 0$
  - $\geq 0$
  - $-0$
  - $0$
  - $+0$

Global refinements potentially expensive.

How can we locally refine an abstract domain?
Refining Abstract Domains

- Global refinements potentially expensive.
- How can we locally refine an abstract domain?
Trace Partitioning

- Trace partitioning allows for flexible and local refinement
Trace Partitioning

- Trace partitioning allows for flexible and local refinement
  - Consider separately different sets of traces through a program
  - Similar to case splits in a mathematical proof.
Trace Partitioning

- Trace partitioning allows for flexible and local refinement
  - Consider separately different sets of traces through a program
  - Similar to case splits in a mathematical proof.

![Control-flow based trace partitioning]

```plaintext
x := *
[x > 5] y := -1
assert(y != 0)
[x <= 5] y := 1
```
Trace Partitioning

- Trace partitioning allows for flexible and local refinement
  - Consider separately different sets of traces through a program
  - Similar to case splits in a mathematical proof.

```
x := *
[x > 5]
y := -1
assert(y != 0)
y = -1
[x <= 5]
y := 1
```
Trace Partitioning

- Trace partitioning allows for flexible and local refinement
  - Consider separately different sets of traces through a program
  - Similar to case splits in a mathematical proof.

Control-flow based trace partitioning

\[ x := * \]
\[ \begin{align*}
[x > 5] & \quad y := -1 \\
[x \leq 5] & \quad y := 1
\end{align*} \]
\[ \text{assert}(y \neq 0) \]
\[ y = -1 \]
\[ y = 1 \]
Trace Partitioning

- Wide range of partitionings possible
  - control flow,
  - values of variables,
  - number of iterations through a loop, etc.

```java
x := y;
if (x > 5) assert (y > 5);
assume (y > 5);
y > 5 assume (y <= 5);
⊥
```
Trace Partitioning

- Wide range of partitionings possible
  - control flow,
  - values of variables,
  - number of iterations through a loop, etc.

Value-based partitioning

```
x := y;
if(x > 5)
  assert(y > 5);
```
Trace Partitioning

▶ Wide range of partitionings possible

▶ control flow,
▶ values of variables,
▶ number of iterations through a loop, etc.

Value-based partitioning

```plaintext
assume(y > 5);
  x := y;
  if(x > 5)
    assert(y > 5);

y > 5
```
Trace Partitioning

- Wide range of partitionings possible
  - control flow,
  - values of variables,
  - number of iterations through a loop, etc.

Value-based partitioning

```plaintext
assume(y > 5); assume(y <= 5);

x := y;
if(x > 5)
  assert(y > 5);

y > 5

⊥
```
Finding Partitioning Functions

- Trace partitioning allows one to refine the precision of an analysis down to explicit exploration of all traces.
Finding Partitioning Functions

- Trace partitioning allows one to refine the precision of an analysis down to explicit exploration of all traces.

*The main question is:*
Finding Partitioning Functions

- Trace partitioning allows one to refine the precision of an analysis down to explicit exploration of all traces.

The main question is:

_How can we find a good partitioning?_
Finding Partitioning Functions

- Trace partitioning allows one to refine the precision of an analysis down to explicit exploration of all traces.

*The main question is:*

*How can we find a good partitioning?*

- Precise enough to prove the property, and
- abstract enough to be efficient.
Finding Partitioning Functions

- Leino and Logozzo (APLAS 2005): Value-based trace partitionings based on counter examples

- Gulavani et al. (TACAS 2008): DAG-based Exploration of control-flow paths inside loops with splitting on demand.

- Gulwani et al. (PLDI 2009): Control-flow refinement for bounds analysis.

- Harris et al. (POPL 2010): Satisfiability Modulo Path Programs
Value-based Trace Partitionings

- If the abstract transformer \( \hat{F} \) is too imprecise, find a set of transformers \( \hat{F}_1, \ldots, \hat{F}_k \), such that

\[
\bigcup_{1 \leq i \leq k} \gamma(\mu X. \hat{F}_i(X)) \supseteq \mu X. F(X)
\]
Value-based Trace Partitionings

- If the abstract transformer $\hat{F}$ is too imprecise, find a set of transformers $\hat{F}_1, \ldots, \hat{F}_k$, such that

$$\bigcup_{1 \leq i \leq k} \gamma(\mu X. \hat{F}_i(X)) \supseteq \mu X. F(X)$$

- This can be done by clipping the analysis by an abstract element:

$$\hat{F}_i = \hat{F} \cap a_i$$
Value-based Trace Partitionings

If the abstract transformer \( \hat{F} \) is too imprecise, find a set of transformers \( \hat{F}_1, \ldots, \hat{F}_k \), such that

\[
\bigcup_{1 \leq i \leq k} \gamma(\mu X. \ \hat{F}_i(X)) \supseteq \mu X. \ F(X)
\]

This can be done by clipping the analysis by an abstract element:

\[
\hat{F}_i = \hat{F} \cap a_i
\]
Value-based Trace Partitionings

- If the abstract transformer $\hat{F}$ is too imprecise, find a set of transformers $\hat{F}_1, \ldots, \hat{F}_k$, such that

$$\bigcup_{1 \leq i \leq k} \gamma(\mu X. \hat{F}_i(X)) \supseteq \mu X. F(X)$$

- This can be done by clipping the analysis by an abstract element:

$$\hat{F}_i = \hat{F} \cap a_i$$
Value-based Trace Partitionings

- If the abstract transformer $\hat{F}$ is too imprecise, find a set of transformers $\hat{F}_1, \ldots, \hat{F}_k$, such that
  \[ \bigcup_{1 \leq i \leq k} \gamma(\mu X. \hat{F}_i(X)) \supseteq \mu X. F(X) \]

- This can be done by clipping the analysis by an abstract element:
  \[ \hat{F}_i = \hat{F} \cap a_i \]
Value-based Trace Partitionings

New question:
Value-based Trace Partitionings

New question:

How can we find such a set of elements $a_1, \ldots, a_k$?
Value-based Trace Partitionings

New question:

How can we find such a set of elements $a_1, \ldots, a_k$?

Use the search architecture of a SAT solver!
DPLL framework

DPLL procedure:
- **decide**: Assume a value for an undetermined variable
- **propagate**: Deduce implied variable values
- **Conflict**: Learn reason for conflict and backtrack
- **learn**: Add learned conflict to knowledge base
- **backtrack**: Return to previous variable for decision
DPLL framework

Main phases of the DPLL procedure:
DPLL framework

- Main phases of the DPLL procedure:
  - **Decision**: Assume a value for an undetermined variable
DPLL framework

- Main phases of the DPLL procedure:
  - **Decision**: Assume a value for an undetermined variable
  - **Propagation**: Deduce implied variable values
DPLL framework

Main phases of the DPLL procedure:

- **Decision**: Assume a value for an undetermined variable
- **Propagation**: Deduce implied variable values
- **Learning**: Learn reason for conflict and backtrack
DPLL framework

Main phases of the DPLL procedure:

- **Decision**: Assume a value for an undetermined variable
- **Propagation**: Deduce implied variable values
- **Learning**: Learn reason for conflict and backtrack
DPLL Procedure

Is $\phi(x, y, z)$ satisfiable?
Is $\phi(x, y, z)$ satisfiable?

$x = 1$
DPLL Procedure

Is \( \phi(x, y, z) \) satisfiable?

\[ x = 1 \]
\[ z = 0 \]
DPLL Procedure

Is $\phi(x, y, z)$ satisfiable?

$x = 1$
$y = 1$
$z = 0$

Decision
DPLL Procedure

Is $\phi(x, y, z)$ satisfiable?

- $x = 1$
- $y = 1$
- $z = 1$
- $z = 0$

Propagation
Is $\phi(x, y, z)$ satisfiable?

- $x = 1$
- $y = 1$
- $z = 0$
- $z = 1$

Conflict
DPLL Procedure

Is $\phi(x, y, z)$ satisfiable?

$x = 1$

$y = 1$

$z = 0$

$z = 1$

Learning
DPLL Procedure

Is $\phi(x, y, z)$ satisfiable?

$x = 0$

Learning
SAT-Style Program Analysis

SAT-Style Program Analysis

- Decide
- Cliped fixpoint
- Safety proven
- Generalize
- Backtrack

Decision Refine current element

\[ a \xrightarrow{\text{by}} a' \]

Propagation Compute clipped fixpoint

\[ \mu X. \hat{T}(X) \cap a' \]

Learning Find

\[ a'' \sqsubseteq a', \text{such that} \]

\[ \mu X. \hat{F}(X) \cap a'' \text{is safe.} \]
SAT-Style Program Analysis

Decision  Refine current element $a$ by $a' \sqsubseteq a$
SAT-Style Program Analysis

Decision  Refine current element $a$ by $a' \sqsubset a$

Propagation  Compute clipped fixpoint $\mu X. \hat{T}(X) \sqcap a'$
SAT-Style Program Analysis

**Decision**  Refine current element $a$ by $a' \sqsubset a$

**Propagation**  Compute clipped fixpoint $\mu X. \hat{T}(X) \sqcap a'$

**Learning**  Find $a'' \sqsubseteq a'$, such that $\mu X. \hat{F}(X) \sqcap a''$ is safe.
Initially, $a = \top$. 

Decision: 

- For $A_1$, decision is refined $\hat{F}(X) \land A_1$ is not safe. 
- For $B_2$, decision is refined $\hat{F}(X) \land B_2$ is safe. 

Backtrack and continue...
SAT-Style Program Analysis

Initially, $a = \top$

Propagation

Initially, $a = \top$

$\mu X. \hat{F}(X)$ not safe

Decision: refine $a = X$.

$\hat{F}(X) \oplus a_1$ not safe

Decision: refine $a = X$.

$\hat{F}(X) \oplus a_2$ safe

Backtrack and continue
Initially, \( \mu X. \hat{F}(X) \) not safe. Decision: refine \( \mu X. \hat{F}(X) \).

\( A_1 \) is safe. Decision: refine \( \mu X. \hat{F}(X) \).

\( A_2 \) \( \land \) \( A_1 \) is not safe. Decision: refine \( \mu X. \hat{F}(X) \).

\( A_3 \) \( \land \) \( A_1 \) is safe. Decision: refine \( \mu X. \hat{F}(X) \).

\( A_4 \) \( \land \) \( A_1 \) is not safe. Backtrack and continue.
Initially, $\mu X.(\hat{F}(X) \land A_1)$ not safe

Decision: refine $\mu X.$

$\hat{F}(X) \land A_1$ not safe

Backtrack and continue
SAT-Style Program Analysis

Initially, $a = \top$

$\hat{F}(X)$ not safe

Decision: refine $a$

$\hat{F}(X) \cap A_1$ not safe

$A_1 \cup B_2$ safe

$A_2 \cup B_3$ safe

$A_3 \cup B_4$ safe

$A_4 \cup B_5$ safe

Backtrack and continue
Initially, $a = \top$. $
abla X. (\hat{F}(X) \sqcap B_2)$ safe.

Decision: refine $a$. $\hat{F}(X) \sqcap A_1$ not safe.

Propagation: $\bot$.

Decision: refine $a\mu X. X. (\hat{F}(X) \sqcap A_2)$. $\bot$.

Decision: refine $a\mu X. X. (\hat{F}(X) \sqcap A_3)$. $\bot$.

Decision: refine $a\mu X. X. (\hat{F}(X) \sqcap A_4)$. $\bot$.

Decision: refine $a\mu X. X. (\hat{F}(X) \sqcap B_2)$. $\bot$. 

Backtrack and continue.
Initially, $\mu(X) = \top$. 

$\hat{F}(X)$ not safe

Decision: refine $\mu(X)$. 

$\hat{F}(X) \land A_1$ not safe

$A_2 \mu(X)$. 

$\hat{F}(X) \land A_2$ safe

$B_2 \mu(X)$.

$B_3 \land B_4 \land B_5$ safe

Backtrack and continue
SAT-Style Program Analysis

Decision Propagation

Generalization

Initially, \( a = \top \).

\[ \hat{F}(\mu X.\hat{F}(X) \sqcap A_2) \text{ safe} \]
Initially, $a = \top$.\n
$\mu X. \hat{F}(X) \sqcap A_2$ safe

Backtrack and continue.
Initially, $a = \top$. $\hat{F}(X)$ not safe

Decision: refine $a$.

$\hat{F}(X) \land A_1$ not safe

Decision: refine $a$.

$A_2 \land \hat{F}(X)$ safe

$B_2$ safe

Backtrack and continue

$C_1$ safe

$C_2$ safe

$C_3$ safe

$C_4$ safe

$\bot$
Comments on Analysis

- When can we efficiently prove safety with this?
Comments on Analysis

- When can we efficiently prove safety with this?
  - When there is a small and finite number of elements $a_1, \ldots, a_k$ such that the fixpoints $\mu X. (\hat{F}(X) \sqcap a_i)$ can be put together to form a concrete postfixpoint.
Comments on Analysis

> When can we efficiently prove safety with this?

  > When there is a small and finite number of elements \( a_1, \ldots, a_k \) such that the fixpoints \( \mu X. (\hat{F}(X) \cap a_i) \) can be put together to form a concrete postfixpoint.

> Specific implementation issues:

  > Generalization step
  > Decision heuristic
Value-based Refinement for Intervals

We have created a preliminary instantiation of this framework for the domain of intervals.
Value-based Refinement for Intervals

We have created a preliminary instantiation of this framework for the domain of intervals.

**Decision:**
Choose an initial assignment for all variables
Value-based Refinement for Intervals

We have created a preliminary instantiation of this framework for the domain of intervals.

**Decision:**
Choose an initial assignment for all variables

**Propagation:**
Compute forward interpretation for this initial value
Value-based Refinement for Intervals

We have created a preliminary instantiation of this framework for the domain of intervals.

**Decision:**
Choose an initial assignment for all variables

**Propagation:**
Compute forward interpretation for this initial value

**Generalization and Learning:**
Generalize the result by locally generalizing intervals. Remove generalized initial values from selection pool
Example 1

Choose initial: \( x = 0, y = 0 \)

\[ [x > 5] \]
\[ [x \leq 5] \]

\( y := -1 \)
\( y := 1 \)

assert(\( y \neq 0 \));
Example 1

Choose initial: \( x = 0, y = 0 \)

- \( x > 5 \)
  - \( x \leq 5 \)
    - \( y = -1 \)
    - \( y = 1 \)

assert(\( y \neq 0 \));

- \( \bot \)
- \( \top \)
Example 1

Choose initial: $x = 0, y = 0$

- $x > 5$
  - $y := -1$
  - $y := 1$
  - $\neg x \leq 5$

assert($y \neq 0$);

Generalization
Example 1

Choose initial: \( x = 0, y = 0 \)

\([x > 5]\) \( \Rightarrow y := -1 \)

\([x \leq 5]\) \( \Rightarrow y := 1 \)

assert\((y \neq 0)\);

\( \top \)
Example 1

Choose initial: $x = 0, y = 0$

- $x > 5$
- $x \leq 5$

- $y := -1$
- $y := 1$

- $y > 0$
- $y \not= 0$

Generalized init:

- $x \leq 5$
- $x > 5$

Choose initial:

- $x = 0, y = 0$

Generalization
Example 1

Choose initial: $x = 0, y = 0$

$[x > 5]$ $[x \leq 5]$

$y := -1$ $y := 1$

$y > 0$

assert($y! = 0$);

$\top$
Example 1

Generalization

Generalized init: \( x \leq 5 \)

\[
\begin{align*}
[x > 5] & \quad [x \leq 5] \\
y := -1 & \quad y := 1 \\
\bot & \quad \top \\
\text{assert}(y \neq 0); & \\
\top &
\end{align*}
\]
Example 1

\[\neg x \leq 5\]

Choose initial: \(x = 8, y = 0\)

\(\neg x \leq 5\)

\[\neg x \leq 5\]
Example 1

Propagation

Choose initial: $x = 0, y = 0$

Choose initial: $x = 0, y = 1$

Generalized init: $x \leq 5$

Choose initial: $x = 8, y = 0$

Generalized init: $x > 5$

assert($y \neq 0$);
Example 1

\[ \begin{align*}
\neg x \leq 5 \\
\neg x > 5
\end{align*} \]

\[ y := -1 \]

\[ y := 1 \]

\[ \text{assert}(y \neq 0); \]

\[ y < 0 \]

\[ \bot \]

\[ \top \]
Example 1

SMT-Style Program Analysis
Value-based Refinement for Intervals

[x > 5] [x <= 5]

y := -1 y := 1

assert(y != 0);

Generalization

⊥ ¬ x ≤ 5 ¬ x > 5

⊥ ⊤ y < 0 ⊤ ¬ x > 5

⊥
Example 2

Decision

[decision]

x := y

[x < 5]

assert (y < 5)

[x >= 5]

x := 0, y := 1
Example 2

Propagation

- - - - - $x = 0, y = 1$
  $x := y$

- - - - - $x = 1, y = 1$
  $[x < 5]$

- - - - - $x = 1, y = 1$
  $[x \geq 5]$

- - - - - $x = 1, y = 1$
  assert ($y < 5$)

- - - - - $x = 1, y = 1$
Example 2

Generalization

Generalized init: $y < 5$

$x := y$

$[x < 5]$

$\neg y < 5$

$[x \geq 5]$

$\neg y < 5$

$\top$

assert($y < 5$)
Example 2

\[ x := y \]

\[ [x < 5] \]

assert \( y < 5 \)

\[ [x \geq 5] \]

\[ \neg y < 5 \]
Example 2

SMT-Style Program Analysis

Value-based Refinement for Intervals

Decision

\[ \neg y < 5 \]

\[ \begin{align*}
\text{x := y} & \\
\text{[x<5]} & \\
\text{[x>=5]} & \\
\text{assert(y<5)} & \\
\text{x = 0, y = 6} & \\
\end{align*} \]
Example 2

\[
x := y \\
\begin{array}{ll}
[x < 5] & \Rightarrow y < 5 \\
[x \geq 5] & \Rightarrow \neg y < 5
\end{array}
\]
Example 2

Generalization

Generalized init: \( y \geq 5 \)

\[ x := y \]

\( [x \geq 5] \)

\( [x < 5] \)

assert(\( y < 5 \))

Generalized init: \( \neg y < 5 \)

\( x = 0, y = 1 \)

\( x = 1, y = 1 \)

\( x = 1, y = 1 \)

\( x = 1, y = 1 \)

\( x = 6, y = 6 \)

\( x = 6, y = 6 \)

\( x = 6, y = 6 \)

\( x = 6, y = 6 \)
Example 2

\begin{align*}
x &:= y \\
\text{assert}(y < 5) \\
\text{assert}(y \geq 5)
\end{align*}
Notes on Implementation

- Initial values chosen by call to a SAT solver.
Notes on Implementation

- Initial values chosen by call to a SAT solver.
- Generalization uses local repair (SMPP):
  - Set every location to $\top$
  - For each invalid triple $\{\text{pre} \}$ stmt $\{\text{post}\}$
    - repair with $\{\text{pre}\}$ from forward analysis.
    - generalize using search on bounds.
Notes on Implementation

- Initial values chosen by call to a SAT solver.
- Generalization uses local repair (SMPP):
  - Set every location to $\top$.
  - For each invalid triple $\{pre\}$ stmt $\{post\}$
    - repair with $\{pre\}$ from forward analysis.
    - generalize using search on bounds.
- Generalization step:

  $$0 \leq a \leq 5, \ b > 5, \ c < 10$$  
  Repair using SAT solver

  $$\text{assert}(a \leq 10 \lor a \geq -10)$$

  $$b > 5$$
Notes on Implementation

- Initial values chosen by call to a SAT solver.
- Generalization uses local repair (SMPP):
  - Set every location to $\top$
  - For each invalid triple $\{pre\} \texttt{stmt} \{post\}$
    - repair with $\{pre\}$ from forward analysis.
    - generalize using search on bounds.
- Generalization step:

\[
\begin{align*}
0 &\leq a \leq 5, \quad b > 5, \quad c < 10 \\
\text{assert(a} &\leq 10 \quad \text{|| a} \geq -10) \\
\text{Repairs using SAT solver} \\
\text{Increase bounds by search} \\
\end{align*}
\]
Notes on Implementation

- Initial values chosen by call to a SAT solver.
- Generalization uses local repair (SMPP):
  - Set every location to $\top$
  - For each invalid triple $\{pre\} \texttt{stmt} \{post\}$
    - repair with $\{pre\}$ from forward analysis.
    - generalize using search on bounds.
- Generalization step:

\[
0 \leq a \leq \infty, \ b > 5, \ c < 10
\]
\[
\text{assert}(a \leq 10 \ || \ a \geq -10)
\]
\[
b > 5
\]

Repair using SAT solver
Increase bounds by search
Notes on Implementation

- Initial values chosen by call to a SAT solver.
- Generalization uses local repair (SMPP):
  - Set every location to $\top$
  - For each invalid triple $\{pre\}$ stmt $\{post\}$
    - repair with $\{pre\}$ from forward analysis.
    - generalize using search on bounds.
- Generalization step:

$$0 \leq a \leq \infty, b > 5, c < 10$$

Repair using SAT solver

assert($a \leq 10$ $\lor$ $a \geq -10$)

Increase bounds by search

$$b > 5$$
Notes on Implementation

- Initial values chosen by call to a SAT solver.
- Generalization uses local repair (SMPP):
  - Set every location to $\top$
  - For each invalid triple $\{\text{pre}\} \text{ stmt } \{\text{post}\}$
    - repair with $\{\text{pre}\}$ from forward analysis.
    - generalize using search on bounds.
- Generalization step:

  $-10 \leq a \leq \infty, b > 5, c < 10$
  
  Repair using SAT solver

  assert(a <= 10 || a >= -10)
  
  Increase bounds by search
  
  $b > 5$
Preliminary benchmarks

- Selection of *NEC Small Static Analysis Benchmarks* (slightly modified)
- Interval analysis too imprecise in all cases
Preliminary benchmarks

- Selection of *NEC Small Static Analysis Benchmarks* (slightly modified)
- Interval analysis too imprecise in all cases

<table>
<thead>
<tr>
<th>Inst.</th>
<th># paths (SCC-decomp.)</th>
<th>runtime (s)</th>
<th>iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>inf1.c</td>
<td>36</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>inf2.c</td>
<td>12</td>
<td>0.7</td>
<td>5</td>
</tr>
<tr>
<td>inf3.c</td>
<td>16</td>
<td>0.9</td>
<td>4</td>
</tr>
<tr>
<td>inf4.c</td>
<td>1080</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>inf5.c</td>
<td>28</td>
<td>2.1</td>
<td>19</td>
</tr>
<tr>
<td>inf6.c</td>
<td>32</td>
<td>0.9</td>
<td>4</td>
</tr>
<tr>
<td>inf7.c</td>
<td>27</td>
<td>1.7</td>
<td>7</td>
</tr>
<tr>
<td>inf8.c</td>
<td>40</td>
<td>3.3</td>
<td>9</td>
</tr>
</tbody>
</table>
Preliminary benchmarks

- Selection of *NEC Small Static Analysis Benchmarks* (slightly modified)
- Interval analysis too imprecise in all cases

<table>
<thead>
<tr>
<th>Inst.</th>
<th># paths (SCC-decomp.)</th>
<th>runtime (s)</th>
<th>iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>inf1.c</td>
<td>36</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>inf2.c</td>
<td>12</td>
<td>0.7</td>
<td>5</td>
</tr>
<tr>
<td>inf3.c</td>
<td>16</td>
<td>0.9</td>
<td>4</td>
</tr>
<tr>
<td>inf4.c</td>
<td>1080</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>inf5.c</td>
<td>28</td>
<td>2.1</td>
<td>19</td>
</tr>
<tr>
<td>inf6.c</td>
<td>32</td>
<td>0.9</td>
<td>4</td>
</tr>
<tr>
<td>inf7.c</td>
<td>27</td>
<td>1.7</td>
<td>7</td>
</tr>
<tr>
<td>inf8.c</td>
<td>40</td>
<td>3.3</td>
<td>9</td>
</tr>
</tbody>
</table>

- Does not work if fully relational information is required (inf1.c, inf4.c)

```plaintext
assume(x > y);
assert(x > y);
```
Current Work

- Extending the prototype into a tool
Current Work

- Extending the prototype into a tool
- Move towards a fully SAT-style analyzer
Current Work

- Extending the prototype into a tool
- Move towards a fully SAT-style analyzer
- Handling of floating-point numbers
Current Work

- Extending the prototype into a tool
- Move towards a fully SAT-style analyzer
- Handling of floating-point numbers
- Move to more powerful domains
Current Work

- Extending the prototype into a tool
- Move towards a fully SAT-style analyzer
- Handling of floating-point numbers
- Move to more powerful domains
- Use trace partitioning and SMT/SAT-style analysis as “glue“ to combine a static analyzer with a bounded model checker.