Proving or Disproving Properties with Constraint Reasoning

T. Denmat      M. Ducassé      A. Gotlieb
Irisa Rennes France
WLPE 05 - Sitges
Inference of Program Properties

Abstract semantic

OVER-approximation

Program

UNDER-approximation

Observations of executions

DEDUCTION

Static analysis

Sound properties

INDUCTION

Dynamic analysis

Precise properties

???
Our approach: 1-Modeling

- C program
  \[ P(X) = Y \]

- Constraint
  \[ P(X, Y) \]

- Modeling of the relational semantics
  \[ S[P] = \{(X,Y) \mid \text{there exists a trace } t \text{ with } \text{init}(t) = X \text{ and } \text{final}(t) = Y \} \]

- Correct and complete
  \[ P(X,Y) = \text{true} \iff (X,Y) \in S[P] \]

- Implemented in Inka
Our approach : 2-Inducing

- Inference of an invariant (= property)
  - Relation between the memory states $X$ and $Y$

- Could be a relation between intermediary states
Our approach: 3-Refuting

C program
\[ P(X) = Y \]

Constraint
\[ P(X, Y) \]

Invariant
\[ Inv(X, Y) \]

Executions pool

Solving of
\[ P(X, Y) \land \neg Inv(X, Y) \]

No solution

\[ Inv(X, Y) \text{ is sound} \]

(Xs, Ys) is a solution

Counterexample
Our approach: 4-Refining

- Enlarge the pool of executions with the new one
- Maybe refine directly the invariant
Expected contributions

- Obtain the correctness of dynamically inferred invariants

- Precise invariants due to the mechanism of refinement

- Potentially very large panel of invariants (all the relations !)
Outline

- Step 1: Translation of an imperative program into CLP(FD)
- Step 2: Dynamic inference of properties
  \((\text{Daïkon as a black-box})\)
- Step 3: Validation of properties
  - Motivating example
  - Problems and future work

- No step 4 until now !!!
Constraint-model of a program

- Translation of an imperative program into a constraint system
- 2 main problems
  - multiple assignments to a variable
  - conditionals and loops

- Approach of Gotlieb et al. [ISSTA 98]
  - SSA-Form
  - New constraint combinators
SSA Form

- Translation of the program into SSA-form
  - Preserves the semantics
  - Each variable is assigned only once during execution
    - Except the iteration structures
  - Data flow is preserved via phi-functions

- Direct translation into constraints
  - A variable in the SSA form -> A logic variable
  - A control-structure -> A constraint
"Ite" combinator

<table>
<thead>
<tr>
<th>C program</th>
<th>SSA - form</th>
<th>Constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>if c</td>
<td>if c</td>
<td>ite(c, v₀, v₁, v₂, C_then, C_else)</td>
</tr>
<tr>
<td>S_then</td>
<td>S'_then</td>
<td></td>
</tr>
<tr>
<td>else</td>
<td>else</td>
<td></td>
</tr>
<tr>
<td>S_else</td>
<td>S'_else</td>
<td></td>
</tr>
<tr>
<td>v₂ = Φ(v₀, v₁)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Guarded constraints**

\[
\begin{align*}
\neg (c \land C_{\text{then}} \land v₂ = v₀) & \rightarrow \neg c \land C_{\text{else}} \land v₂ = v₁ \\
\neg (\neg c \land C_{\text{then}} \land v₂ = v₁) & \rightarrow c \land C_{\text{then}} \land v₂ = v₀
\end{align*}
\]
“w” Combinator

**C program**
while c
S

**SSA - form**
\( v_2 = \Phi(v_0, v_1) \)
while c
S'

**Constraint**
\( w(c, v_0, v_1, v_2, C_{\text{body}}) \)

\[
W(c, v_0, v_1, v_2, C_{\text{body}}) : \\
\neg (c \land C_{\text{body}}) \rightarrow \neg c \land v_2 = v_0 \\
\neg (\neg c \land v_0 = v_2) \rightarrow c \land C_{\text{body}} \land w(c, v_1, v_3, v_2, C'_{\text{body}})
\]
Dynamic inference of properties

- We use Daïkon as a black box, in its by default configuration [Ernst ICSE 99]

- Generate a set of potential relationships between variables of a program
  - At “interesting” points of the program
  - For “interesting” variables

- Run a test suite

- Consider relationships that hold over every test case as a Likely Invariant
Motivating example

```c
int foo (int n, int r)
    b = 0;
    while (n > 0)
        if (b == 0)
            b = 1; r ++;
        else
            b = 0; r --;
    return r;
```

Likely invariants inferred

- orig(r) = 0 $\Rightarrow$ return = 0
- return = 0 $\Rightarrow$ orig(r) = 0
- return $\geq$ orig(r)
Problem of the Oracle:
- Difficult to know if likely invariants hold

Automatically checking these invariants is crucial

Related work
- Nimmer and Ernst 02: based on a theorem prover
  - Proving properties
- Vaziri and Jackson 00: based on constraint solving
  - Disproving properties

Our method:
- Both proving and disproving invariants
Declarative semantics of invariant validation

- Gopal Gupta [the LP paradigm 99]
  - Pre(X) : pre-condition on input vector X
  - P(X,Y) : denotation of an imperative program
    - Relation between input vector X and output vector Y
  - Post(X,Y) : post-condition

- Post condition is proved to hold if the following goal has no solution
  - Pre(X), P(X,Y), not Post(X,Y)
State space reduction with CLP

- Using pure horn logic:
  - Generate and Test
  - Try all values of X such that Pre(X)

- Using a CLP denotation:
  - Constrain – generate and Test
  - Asserting not Post(X,Y) reduces the search space

- Conjecture:
  - The reduction makes the approach more tractable
Running example - invariant 1

- Refutation of $\text{orig}(r) = 0 \implies \text{return} = 0$

- $\text{foo}(N,R,\text{Ret}) \land R = 0 \land \text{Ret} \neq 0$

  Input domains reduction:
  
  $N \in [1,\sup], \ R = 0$

  Labeling step:
  
  find a solution: $N = 1, \ R = 0, \ \text{Ret} = 1$

- Invariant 1 is disproved
Refutation of \( return = 0 \Rightarrow \text{orig}(r) = 0 \)

\[ \text{foo}(N,R,\text{Ret}) \land \text{Ret} = 0 \land R \neq 0 \]

Input domains reduction:

\[ N \in [1,\text{sup}], \ R \in [\text{inf},-1] \cup [1,\text{sup}] \]

Labeling step:

find a solution: \( N = 1, \ R = -1, \ \text{Ret} = 0 \)

Invariant 2 is disproved
The labeling step is crucial to find counter examples.

In our two examples the default labeling procedure is “magically” efficient enough:
- For example, beginning to label variable R would have been terrible.

Future work:
- Design specialized heuristics for finding counter examples.
Refutation of \( return \geq orig(r) \)

\[ \text{foo}(N,R,\text{Ret}) \land \text{Ret} < R \]

Input domains reduction:

\[ N \in \emptyset, R \in \emptyset \]

No labeling step

Invariant 3 is proved
Details of the refutation 3

Initial state

<table>
<thead>
<tr>
<th>Constraint store</th>
<th>Variables domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>B = 0,</td>
<td>B in [0,0]</td>
</tr>
<tr>
<td>w(...)</td>
<td>N in [-100,100]</td>
</tr>
<tr>
<td>RET &lt; R</td>
<td>R in [-99,100]</td>
</tr>
<tr>
<td></td>
<td>RET in [-100,99]</td>
</tr>
</tbody>
</table>

int foo (int n, int r)

    b = 0;
    while (n > 0)
        if (b == 0)
            b = 1;
            r ++;
        else
            b = 0;
            r --;
        return r;
Propagation in the w combinator:

entailment checking of the 2nd guard

\[ \neg (\neg c \land v0 = v2) \rightarrow c \land Cbody \land w(c,v1,v3,v2,C'body) \]

**Constraint store**
- B = 0,
- w(...)
- RET < R
- N =< 0,
- RET = R

**Variables domains**
- B in [0,0]
- N in [-100,0]
- R in Ø
- RET in Ø

Failure ➔ the guard is entailed

```c
int foo (int n, int r)
{
    b = 0;
    while (n > 0)
    {
        if (b == 0)
        {
            b = 1;
            r ++;
        }
        else
        {
            b = 0;
            r --;
        }
    return r;
```

```c
}
Propagation of the w combinator:

setting the tail of the constraint

\[ \neg (\neg c \land v_0 = v_2) \rightarrow c \land \text{Cbody} \land w(c,v_1,v_3,v_2,C'body) \]

**Constraint store**

- B = 0,
- w(...)
- RET < R
- N > 0
- N1 = N - 1
- B1 = 1
- R1 = R + 1

**Variables domains**

- \( B \) in \([0,0]\)
- \( N \) in \([1,100]\)
- \( R \) in \([-99,99]\)
- \( \text{RET} \) in \([-100,98]\)
- \( N1 \) in \([0,99]\)
- \( B1 \) in \([1,1]\)
- \( R1 \) in \([-98,100]\)

int foo (int n, int r)

\[ b = 0; \]
while (n > 0)

\[ \text{if} \ (b == 0) \]
\[ \quad b = 1; \]
\[ \quad r ++; \]
\[ \text{else} \]
\[ \quad b = 0; \]
\[ \quad r --; \]
\[ \text{return} \ r; \]
Details of the refutation 3

Propagation in the w combinator:

entailment checking of the 2nd guard again

<table>
<thead>
<tr>
<th>Constraint store</th>
<th>Variables domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>B = 0, w(...) RET &lt; R N &gt; 0 N1 = N -1 B1 = 1 R1 = R + 1 N1 =&lt; 0 RET = R1</td>
<td>B in [0,0] N in [1,100] R in [-99,99] RET in [-100,98] N1 in [0,99] B1 in [1,1] R1 in [-98,100]</td>
</tr>
</tbody>
</table>

```c
int foo (int n, int r)
{
    b = 0;
    while (n > 0)
    {
        if (b == 0)
            b = 1; r ++;
        else
            b = 0; r --;
    return r;
```
Propagation in the w combinator:

entailment checking of the 2nd guard again

**Constraint store**

- B = 0,
- w(...)
- RET < R
- N > 0
- N1 = N -1
- B1 = 1
- R1 = R + 1
- N1 =< 0
- RET = R1

**Variables domains**

- B in [0,0]
- N in [1,1]
- R in [-99,99]
- RET in [-100,98]
- N1 in [0,0]
- B1 in [1,1]
- R1 in [-98,100]

```c
int foo (int n, int r)
{
    b = 0;
    while (n > 0)
    {
        if (b == 0)
        {
            b = 1;
            r ++;
        } else
        {
            b = 0;
            r --;
        }
    return r;
} 
```
Details of the refutation 3

Propagation in the w combinator:
entailment checking of the 2nd guard again

**Constraint store**
- B = 0,
- w(...)
- RET < R
- N > 0
- N1 = N - 1
- B1 = 1
- R1 = R + 1
- N1 <= 0
- RET = R1

**Variables domains**
- B in [0,0]
- N in [1,1]
- R in [-99,99]
- RET in [-98,98]
- N1 in [0,0]
- B1 in [1,1]
- R1 in [-98,98]

```
int foo (int n, int r)
    b = 0;
    while (n > 0)
        if (b == 0)
            b = 1;
            r ++;
        else
            b = 0;
            r --;
    return r;
```
Details of the refutation 3

Propagation in the w combinator:

entailment checking of the 2nd guard again

<table>
<thead>
<tr>
<th>Constraint store</th>
</tr>
</thead>
<tbody>
<tr>
<td>B = 0,</td>
</tr>
<tr>
<td>w(...)</td>
</tr>
<tr>
<td>RET &lt; R</td>
</tr>
<tr>
<td>N &gt; 0</td>
</tr>
<tr>
<td>N1 = N - 1</td>
</tr>
<tr>
<td>B1 = 1</td>
</tr>
<tr>
<td>R1 = R + 1</td>
</tr>
<tr>
<td>N1 =&lt; 0</td>
</tr>
<tr>
<td>RET = R1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variables domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>B in [0,0]</td>
</tr>
<tr>
<td>N in [1,1]</td>
</tr>
<tr>
<td>R in [-97,99]</td>
</tr>
<tr>
<td>RET in [-98,98]</td>
</tr>
<tr>
<td>N1 in [0,0]</td>
</tr>
<tr>
<td>B1 in [1,1]</td>
</tr>
<tr>
<td>R1 in [-98,98]</td>
</tr>
</tbody>
</table>

int foo (int n, int r)
  b = 0;
  while (n > 0)
    if (b == 0)
      b = 1;
      r ++;
    else
      b = 0;
      r --;
  return r;
Details of the refutation 3

Propagation in the w combinator:
entailment checking of the 2nd guard again

```
int foo (int n, int r)
    b = 0;
    while (n > 0)
        if (b == 0)
            b = 1; r ++;
        else
            b = 0;
            r --;
    return r;
```

<table>
<thead>
<tr>
<th>Constraint store</th>
<th>Variables domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>B = 0,</td>
<td>B in [0,0]</td>
</tr>
<tr>
<td>w(...)</td>
<td>N in [1,1]</td>
</tr>
<tr>
<td>RET &lt; R</td>
<td>R in [-97,97]</td>
</tr>
<tr>
<td>N &gt; 0</td>
<td>RET in [-98,98]</td>
</tr>
<tr>
<td>N1 = N -1</td>
<td>N1 in [0,0]</td>
</tr>
<tr>
<td>B1 = 1</td>
<td>B1 in [1,1]</td>
</tr>
<tr>
<td>R1 = R + 1</td>
<td>R1 in [-96,98]</td>
</tr>
<tr>
<td>N1 &lt;= 0</td>
<td></td>
</tr>
<tr>
<td>RET = R1</td>
<td></td>
</tr>
</tbody>
</table>
Details of the refutation 3

Propagation in the w combinator:

entailment checking of the 2nd guard again

\[
\begin{align*}
\text{Constraint store} & : & B = 0, \\
& & \text{w(…)} \\
& & \text{RET} < \text{R} \\
& & N > 0 \\
& & N1 = N - 1 \\
& & B1 = 1 \\
& & R1 = R + 1 \\
& & N1 =< 0 \\
& & \text{RET} = \text{R1} \\
\text{Variables domains} & : & B \text{ in } [0,0] \\
& & N \text{ in } [1,1] \\
& & R \text{ in } \emptyset \\
& & \text{RET} \text{ in } \emptyset \\
& & N1 \text{ in } [0,0] \\
& & B1 \text{ in } [1,1] \\
& & R1 \text{ in } \emptyset \\
\end{align*}
\]

\[
\begin{align*}
\text{int foo (int n, int r)} & : \\
& b = 0; \\
& \text{while } (n > 0) \\
& \quad \text{if } (b == 0) \\
& \quad \quad b = 1; \\
& \quad \quad r ++; \\
& \quad \text{else} \\
& \quad \quad b = 0; \\
& \quad \quad r --; \\
& \quad \text{return } r; \\
\end{align*}
\]
Propagation of the \( w \) combinator: setting the tail of the constraint

<table>
<thead>
<tr>
<th>Constraint store</th>
<th>Variables domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B = 0, )</td>
<td>( B ) in [0,0]</td>
</tr>
<tr>
<td>( w(...) )</td>
<td>( N ) in [2,100]</td>
</tr>
<tr>
<td>( RET &lt; R )</td>
<td>( R ) in [-99,99]</td>
</tr>
<tr>
<td>( N &gt; 0 )</td>
<td>( RET ) in [-100,98]</td>
</tr>
<tr>
<td>( N1 = N - 1 )</td>
<td>( N1 ) in [1,99]</td>
</tr>
<tr>
<td>( B1 = 1 )</td>
<td>( B1 ) in [1,1]</td>
</tr>
<tr>
<td>( R1 = R + 1 )</td>
<td>( R1 ) in [-98,100]</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>( N1 &gt; 0 )</td>
<td></td>
</tr>
<tr>
<td>( N2 = N1 - 1 )</td>
<td>( N2 ) in [0,98]</td>
</tr>
</tbody>
</table>

```c
int foo (int n, int r)
{
    b = 0;
    while (n > 0)
    {
        if (b == 0)
        {
            b = 1;
            r ++;
        }
        else
        {
            b = 0;
            r --;
        }
    }
    return r;
}
```
Details of the refutation 3

<table>
<thead>
<tr>
<th>Constraint store</th>
<th>Variables domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>B = 0, w(...) RET &lt; R N &gt; 0 N1 = N - 1 B1 = 1 R1 = R + 1</td>
<td>B in [0,0] N in [100,100] R in [-99,99] RET in [-100,98] N1 in [99,99] B1 in [1,1] R1 in [-98,100]</td>
</tr>
<tr>
<td>... N1 &gt; 0 N2 = N1 – 1 N100 = N99 - 1</td>
<td>... N2 in [98,98] N100 in [0,0]</td>
</tr>
</tbody>
</table>

We have a failure as it is impossible to unfold the loop and to exit the loop.
The propagation is very long
- We need to show inconsistencies at each loop unfolding
- Each inconsistency is long to demonstrate
  - Bound consistency $\Rightarrow$ slow convergence

Future work
- Use information about the loops such as loop invariants to add redundant constraint
- Mix CLP(FD) with other types of constraint solver
Conclusion

- An approach to both prove and disprove invariants based on constraints
  - No approximation
  - Based on clp(fd)

- Need to specialize constraint techniques to this particular problem
  - Propagation step
  - Labeling step