# Automatic Relational Verification with *llrêve*

## KeY Symposium 2016

Moritz Kiefer

July 22, 2016

Karlsruhe Institute of Technology

# Outline

**Relational Verification**

Given inputs satisfying a relational predicate $\varphi$, do the outputs satisfy the relational predicate $\psi$ for all runs?

**Equivalence of programs $P$ and $Q$**

- $\varphi(x_P, x_Q) = x_P \equiv x_Q$
- $\psi(x'_P, x'_Q) = x'_P \equiv x'_Q$

## Usecases

- Regression verification
    - Verify that refactoring doesn’t change behavior
- Use an existing implementation as specification
    - E. g. write a libc implementation that behaves like *glibc*
- Slicing
    - Covered in Stephan’s talk

LLRÊVE

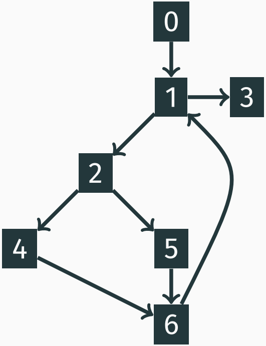
- Relational verification of C code
- Actual verification is done on *LLVM IR*
- (Almost) fully automatic
- Support for all kinds of control flow in C
    - Including arbitrary *GOTO* statements
- Unbounded integers and unbounded arrays

# LLRÊVE

## Synchronization Points

# Synchronization Points

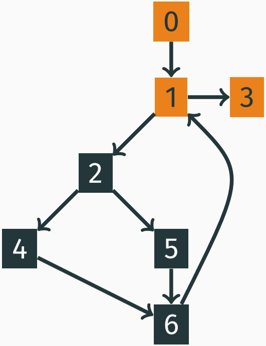
- Break cycles in CFG at *synchronization points*
- Linear paths between synchronization points




- Transition predicate $T_P^{n,m}(x_P, x'_P)$ for path from $n$ to $m$ in program $P$

# Synchronization Points

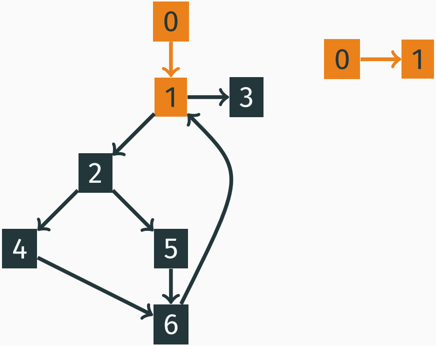
- Break cycles in CFG at *synchronization points*
- Linear paths between synchronization points




- Transition predicate $T_P^{n,m}(x_P, x'_P)$ for path from $n$ to $m$ in program $P$

# Synchronization Points

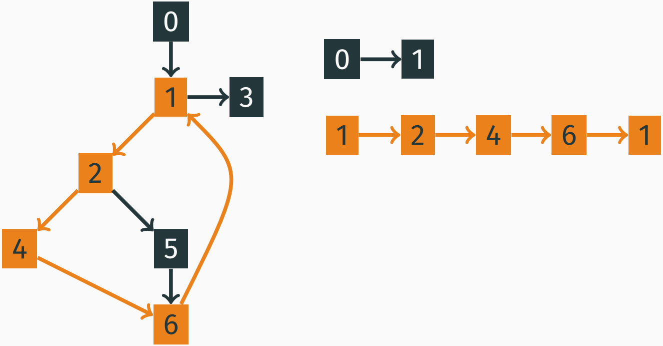
- Break cycles in CFG at *synchronization points*
- Linear paths between synchronization points




- Transition predicate $T_P^{n,m}(x_P, x'_P)$ for path from $n$ to $m$ in program $P$

# Synchronization Points

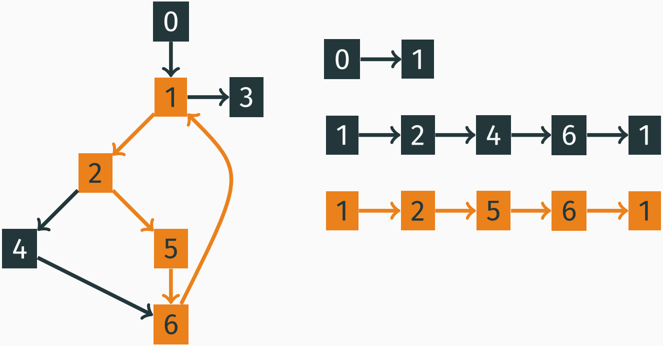
- Break cycles in CFG at *synchronization points*
- Linear paths between synchronization points




- Transition predicate $T_P^{n,m}(x_P, x'_P)$ for path from $n$ to $m$ in program $P$

# Synchronization Points

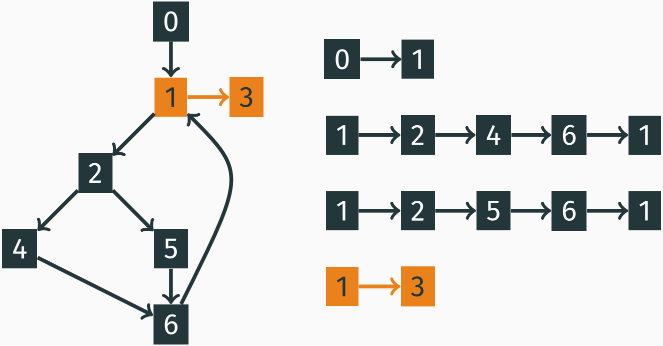
- Break cycles in CFG at *synchronization points*
- Linear paths between synchronization points




- Transition predicate $T_P^{n,m}(x_P, x'_P)$ for path from $n$ to $m$ in program $P$

# Synchronization Points

- Break cycles in CFG at *synchronization points*
- Linear paths between synchronization points




- Transition predicate $T_P^{n,m}(x_P, x'_P)$ for path from $n$ to $m$ in program $P$

# Synchronization Points

- Relational invariants for corresponding synchronization points

$$C_n(x_P, x_Q) \wedge T_P^{n,m}(x_P, x'_P) \wedge T_Q^{n,m}(x_Q, x'_Q) \\ \to C_m(x'_P, x'_Q)$$

- Enforce coupling

$$C_n(x_P, x_Q) \wedge T_P^{n,m}(x_P, x'_P) \wedge T_Q^{n,m'}(x_Q, x'_Q) \\ \to \text{false}, \quad m \neq m'$$

- Non-synchronized loops

$$C_n(x_P, x_Q) \wedge T_P^{n,n}(x_P, x'_P) \wedge (\bigwedge \neg T_Q^{n,n}(x_Q, x'_Q)) \\ \to C_n(x'_P, x_Q)$$

# Example

```c
int f(int n) {
  int r = 0;
  for (int i = 0;
       i < n; ++i) {
    ++r;
  }
  return r;
}
```

```c
int f(int n) {
  int r = 0;
  for (int i = n;
       i > 0; --i) {
    ++r;
  }
  return r;
}
```

# Example

```
int f(int n) {
  int r = 0;
  for (int i = 0;
       i < n; ++i) {
    ++r;
  }
  return r;
}
```

```
int f(int n) {
  int r = 0;
  for (int i = n;
       i > 0; --i) {
    ++r;
  }
  return r;
}
```

# Example

```
int f(int n) {
  int r = 0;
  for (int i = 0;
         i < n; ++i) {
    ++r;
  }
  return r;
}
```

```
int f(int n) {
  int r = 0;
  for (int i = n;
         i > 0; --i) {
    ++r;
  }
  return r;
}
```

$$\varphi(n_P, n_Q) \implies C_0(n_P, 0, 0, n_Q, 0, n_Q)$$

# Example

```
int f(int n) {
  int r = 0;
  for (int i = 0;
       i < n; ++i) {
    ++r;
  }
  return r;
}
```

```
int f(int n) {
  int r = 0;
  for (int i = n;
       i > 0; --i) {
    ++r;
  }
  return r;
}
```

$$C_0(n_P, r_P, i_P, n_Q, r_Q, i_Q) \wedge i_P < n_P \wedge i_Q > 0 \implies C_0(n_P, r_P + 1, i_P + 1, n_Q, r_Q + 1, i_Q - 1)$$

# Example

```
int f(int n) {
  int r = 0;
  for (int i = 0;
       i < n; ++i) {
    ++r;
  }
  return r;
}
```

```
int f(int n) {
  int r = 0;
  for (int i = n;
       i > 0; --i) {
    ++r;
  }
  return r;
}
```

$$C_0(n_P, r_P, i_P, n_Q, r_Q, i_Q) \wedge i_P < n_P \wedge \neg(i_Q > 0) \implies C_0(n_P, r_P + 1, i_P + 1, n_Q, r_Q, i_Q)$$

# Example

```
int f(int n) {
  int r = 0;
  for (int i = 0;
       i < n; ++i) {
    ++r;
  }
  return r;
}
```

```
int f(int n) {
  int r = 0;
  for (int i = n;
       i > 0; --i) {
    ++r;
  }
  return r;
}
```

$$C_0(n_P, r_P, i_P, n_Q, r_Q, i_Q) \wedge \neg(i_P < n_P) \wedge i_Q > 0 \implies C_0(n_P, r_P, i_P, n_Q, r_Q + 1, i_Q - 1)$$

# Example

```
int f(int n) {
  int r = 0;
  for (int i = 0;
       i < n; ++i) {
    ++r;
  }
  return r;
}
```

```
int f(int n) {
  int r = 0;
  for (int i = n;
       i > 0; --i) {
    ++r;
  }
  return r;
}
```

$$C_0(n_P, r_P, i_P, n_Q, r_Q, i_Q) \wedge \neg(i_P < n_P) \wedge \neg(i_Q > 0) \implies \phi(r_P, r_Q)$$

# LLRÊVE

## Mutual Function Summaries

## Mutual Function Summaries

- Match function calls on path combinations
- Abstract matched calls using *mutual function summaries*
- Use mutual function summary as output relation

```
int f(int n) {
    int r = g(1 + n);
    return 1 + r;
}
```

$$\forall n_P, n_Q.\; \varphi(n_P, n_Q) \rightarrow \forall r_P, r_Q.\; g(1 + n_P, 1 + n_Q, r_P, r_Q) \rightarrow \phi(1 + r_P, 1 + r_Q)$$

# Invariant Inference

# Invariant Inference

- *llrêve* uses SMT Horn logic (*Eldarica* or *Z3*)
- Problems
  - Fragile
  - Slow, especially when the number of loop grows
- Idea: Exploit the fact that invariants have limited forms
- Test invariants candidates on execution traces
- Check candidates using SMT solver

# Invariant Inference

## Custom Invariant Patterns

# Custom Invariant Patterns

- (restricted) FOL formula with *holes*
- Holes are instantiated using variables
- Supports arrays
- No inference of constants

# Pattern Examples

- $\_ \geq \_$
- $\_ > \_$
- $\_ < 0$
- $\forall i. heap_Q[i] = heap_P[i]$
- $heap_Q[\_] = \_$

# Invariant Inference

## Polynomial Invariants

# Polynomial Invariants

- Polynomial equation over local variables

$$\sum_{e_1,\ldots,e_n} a_{e_1,\ldots,e_n} x_1^{e_1} \cdot \ldots \cdot x_n^{e_n} = 0$$

- For pratical reasons

$$a_0 + \sum_i \sum_{1 \leq e} a_{i,e} x_i^e = 0$$

- Extract equations from execution traces

# Polynomial Invariants

$c_{i,j}$ = Value of variable i in state j

$$\begin{pmatrix} 1 & c_{1,1} & c_{1,2} & \cdots & c_{1,n} \\ 1 & c_{2,1} & c_{2,2} & \cdots & c_{2,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & c_{m,1} & c_{2,m} & \cdots & c_{m,n} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix} = 0$$

Calculate an arbitrary basis of the kernel

# Polynomial Invariants

$$\begin{pmatrix} 1 & 3 & 3 & 2 \\ 1 & 2 & 2 & 2 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = 0 \qquad \begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & 1 & 0 \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{pmatrix} = 0$$

Basis:

$$\left\{ \begin{pmatrix} 0 \\ 1 \\ -1 \\ 0 \end{pmatrix}, \begin{pmatrix} 2 \\ 0 \\ 0 \\ -1 \end{pmatrix} \right\}$$

Invariant:

$$x_1 = x_2 \wedge 2x_0 = x_3$$

# Advantages

- No need to supply constants
- No need to supply patterns
- Faster than testing equivalent patterns
- Space usage $\mathcal{O}(n^2)$

# Invariant Inference

## Using Counterexamples As New Inputs

## Using Counterexamples As New Inputs

- Invalid invariant → Path condition violated
- SMT solver returns counterexample
- Counterexample → New execution traces
- Refinement of invariants
- One counterexample can refine multiple invariants

# Example

```
int f(int n) {
  int r = 0;
  for (int i = 0;
       i < n; ++i) {
    ++r;
  }
  return r;
}
```

```
int f(int n) {
  int r = 0;
  for (int i = n;
       i > 0; --i) {
    ++r;
  }
  return r;
}
```

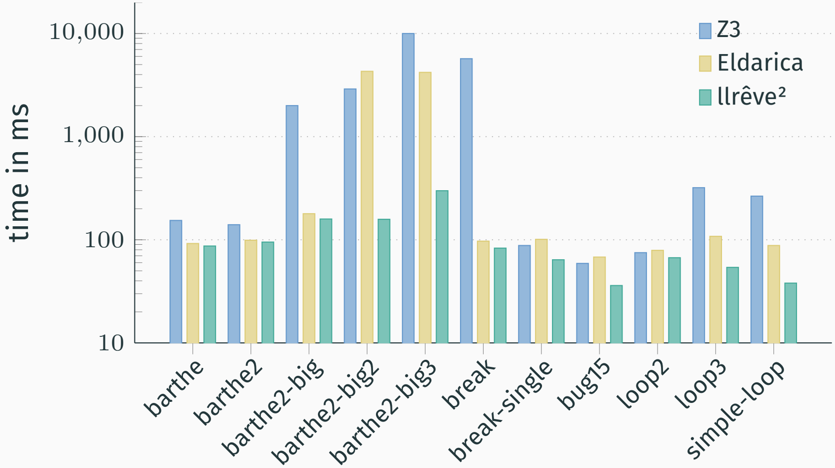
- Initial input $n = 0$

  $n_P = 0 \land n_Q = 0 \land i_P = 0 \land i_Q = 0 \land x_P = 0 \land x_Q = 0$
- Counterexample $n = 1$

  $n_P = n_Q \land i_P = x_P \land i_P + i_Q = n_P \land i_P = x_Q$

# Experiments

# Experiments

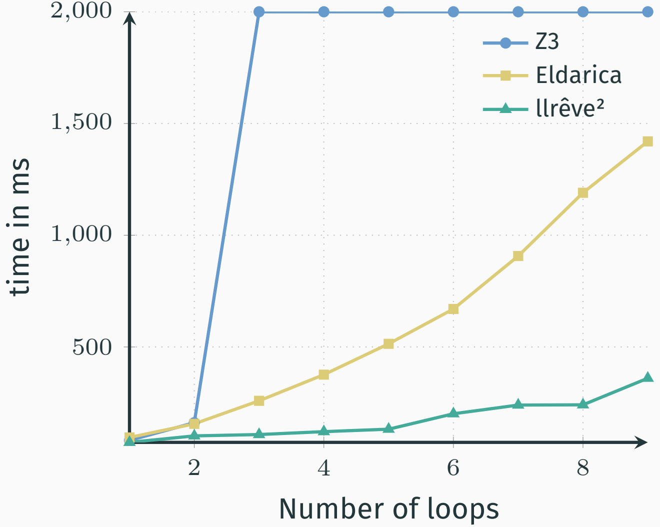


# Experiments

- Verify that program is equivalent to itself
- Increase the number of loops

```c
int f(int n) {
    int x = 0;
    for (int i = 0; i < n; ++i) {
        ++x;
    }
    return x;
}
```

# Experiments



# Conclusion

- LLVM makes verification of “real world” code possible
  - Only two kinds of control flow
    - Branches → Synchronization points
    - Function calls → Mutual function summaries
- Horn solvers too generic for our usecase

  → Exploit the limited form of relational invariants
- Combine
  - Polynomial invariants for equalities
  - Patterns for everything else

# Questions?