# Lockstep Composition for Unbalanced Loops

Ameer Hamza, Grigory Fedyukovich
Florida State University, USA

April 26, Paris, France

# Motivation

- Optimizations (compiler/hand) need formal guarantees
- Checking equivalence of a program (source) and an optimized version (target) is required
- Checking equivalence is difficult, especially for programs with different structures
- Formally verify structure-altering optimizations

# Motivating Example

**Source program**

```
int M = nondet(), K = nondet(),
a = 0, N = 2*M+1+K, b = 2*M+1;
assume(M >= 0 && K >= 0);
while (a != N) {
    if (a >= b) b++;
    a++;
}
```

**Target program**

```
int X = nondet(), Y = nondet(),
c = 1, d = 2*X+1;
assume(X >= 0 && Y >= 0);
while (c < 2*X+1) {
    c += 2;
}
while (c != 2*X+1+Y) {
    d++;
    c++;
}
```

# Motivating Example

## Source program

```
int M = nondet(), K = nondet(),
a = 0, N = 2*M+1+K, b = 2*M+1;
assume(M >= 0 && K >= 0);
while (a != N) {
    if (a >= b) b++;
    a++;
}
```

## Target program

```
int X = nondet(), Y = nondet(),
c = 1, d = 2*X+1;
assume(X >= 0 && Y >= 0);
while (c < 2*X+1) {
    c += 2;
}
while (c != 2*X+1+Y) {
    d++;
    c++;
}
```

# Motivating Example

**Source program**

```
int M = nondet(), K = nondet(),
a = 0, N = 2*M+1+K, b = 2*M+1;
assume(M >= 0 && K >= 0);
while (a != N) {
    if (a >= b) b++;
    a++;
}
```

**Target program**

```
int X = nondet(), Y = nondet(),
c = 1, d = 2*X+1;
assume(X >= 0 && Y >= 0);
while (c < 2*X+1) {
    c += 2;
}
while (c != 2*X+1+Y) {
    d++;
    c++;
}
```

# Motivating Example

**Source program**

```
int M = nondet(), K = nondet(),
a = 0, N = 2*M+1+K, b = 2*M+1;
assume(M >= 0 && K >= 0);
while (a != N) {
    if (a >= b) b++;
    a++;
}
```

**Target program**

```
int X = nondet(), Y = nondet(),
c = 1, d = 2*X+1;
assume(X >= 0 && Y >= 0);
while (c < 2*X+1) {
    c += 2;
}
while (c != 2*X+1+Y) {
    d++;
    c++;
}
```

# Motivating Example

## Source program

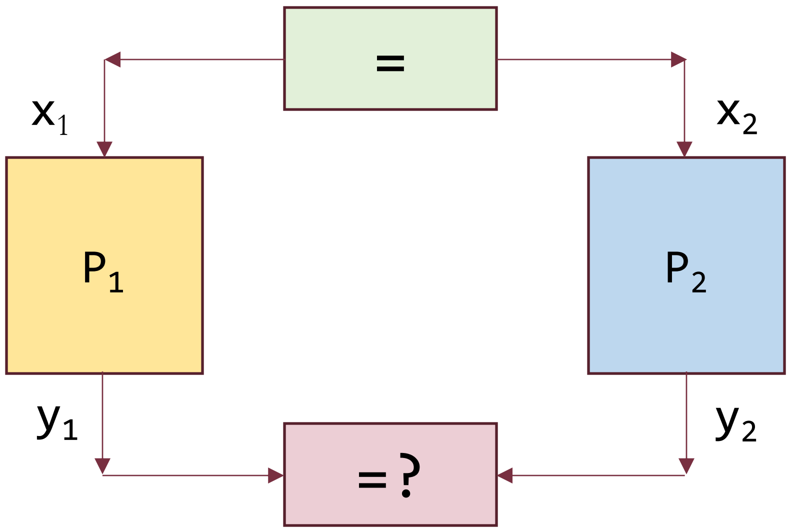
```
int M = nondet(), K = nondet(),
a = 0, N = 2*M+1+K, b = 2*M+1;
assume(M >= 0 && K >= 0);
while (a != N) {
    if (a >= b) b++;
    a++;
}
```

## Target program

```
int X = nondet(), Y = nondet(),
c = 1, d = 2*X+1;
assume(X >= 0 && Y >= 0);
while (c < 2*X+1) {
    c += 2;
}
while (c != 2*X+1+Y) {
    d++;
    c++;
}
```

# Equivalence Checking

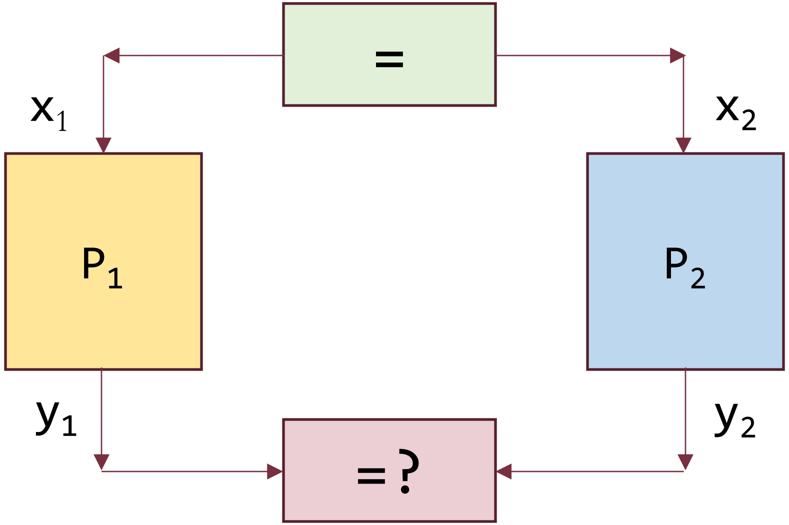
For equivalence, **pre**=**post**=pairwise equality



# Equivalence Checking

For equivalence, **pre**=**post**=pairwise equality

check $\mathbf{x}_1 = \mathbf{x}_2 \Rightarrow \mathbf{y}_1 = \mathbf{y}_2$



# Motivating Example

**Source program**

**pre**: M=X ∧ K=Y

```
int M = nondet(), K = nondet(),
a = 0, N = 2*M+1+K, b = 2*M+1;
assume(M >= 0 && K >= 0);
while (a != N) {
    if (a >= b) b++;
    a++;
}
```

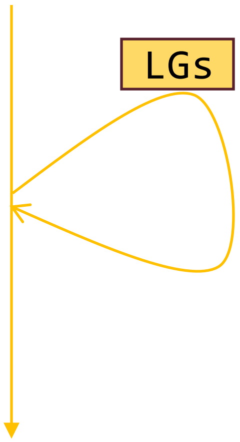
**Target program**

```
int X = nondet(), Y = nondet(),
c = 1, d = 2*X+1;
assume(X >= 0 && Y >= 0);
while (c < 2*X+1) {
    c += 2;
}
while (c != 2*X+1+Y) {
    d++;
    c++;
}
```

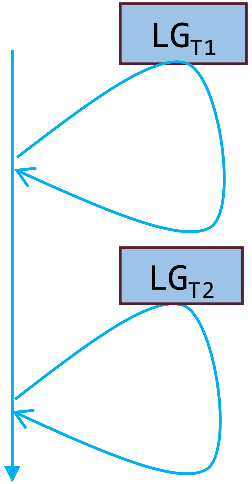
**post**: M=X ∧ K=Y ∧ a=c ∧ b=d

# (Simplified) Control Flow

## Source program

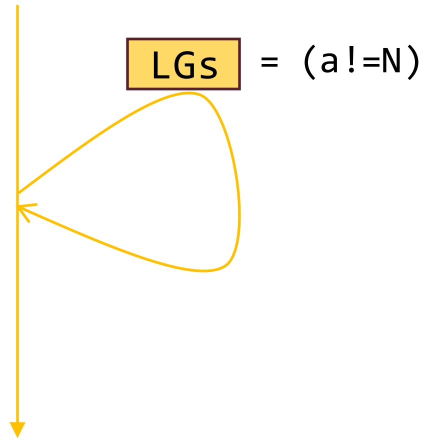



## Target program

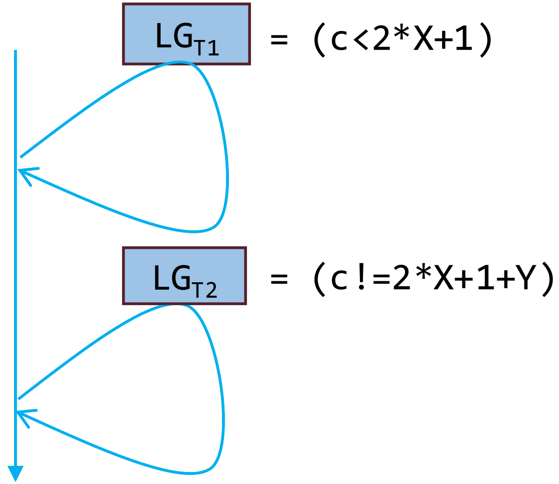


# (Simplified) Control Flow

## Source program

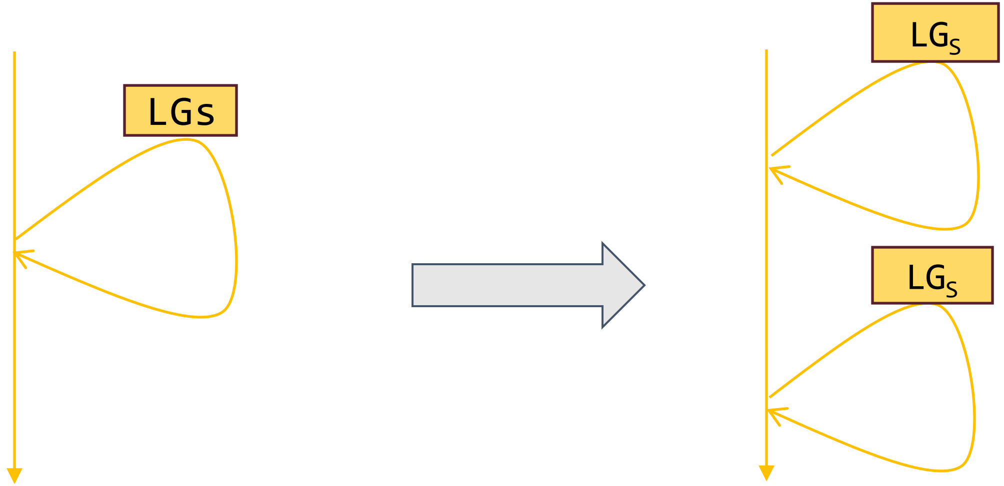



## Target program



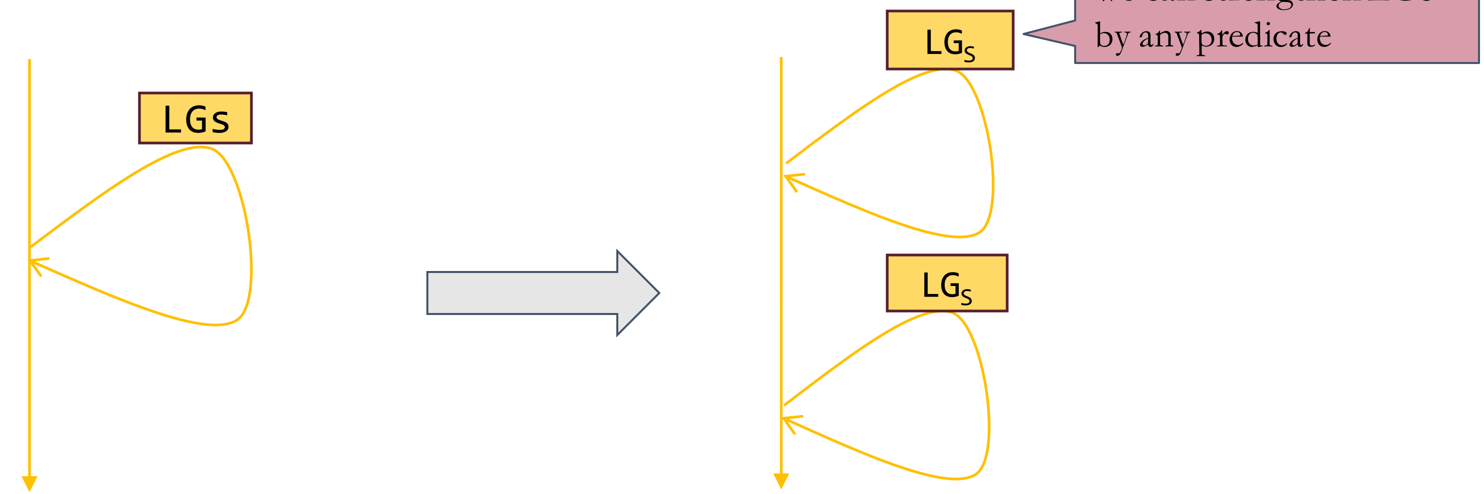
# Decomposition of Source

**Step 1**: create two copies of the source loop



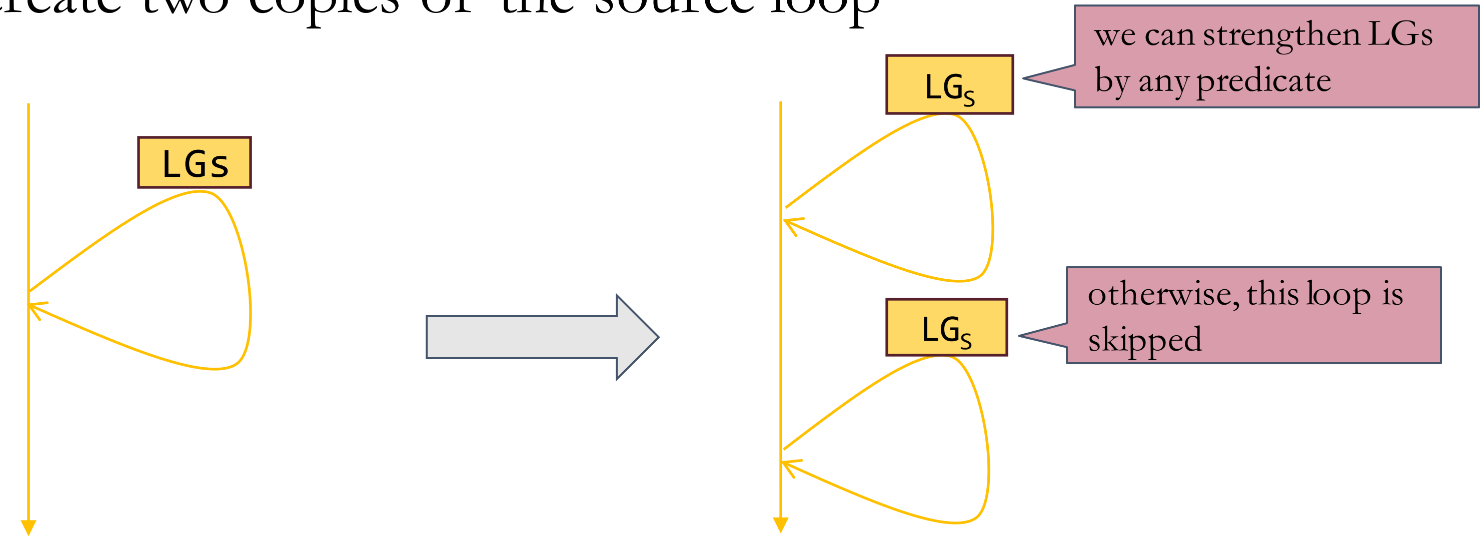
# Decomposition of Source

**Step 1**: create two copies of the source loop



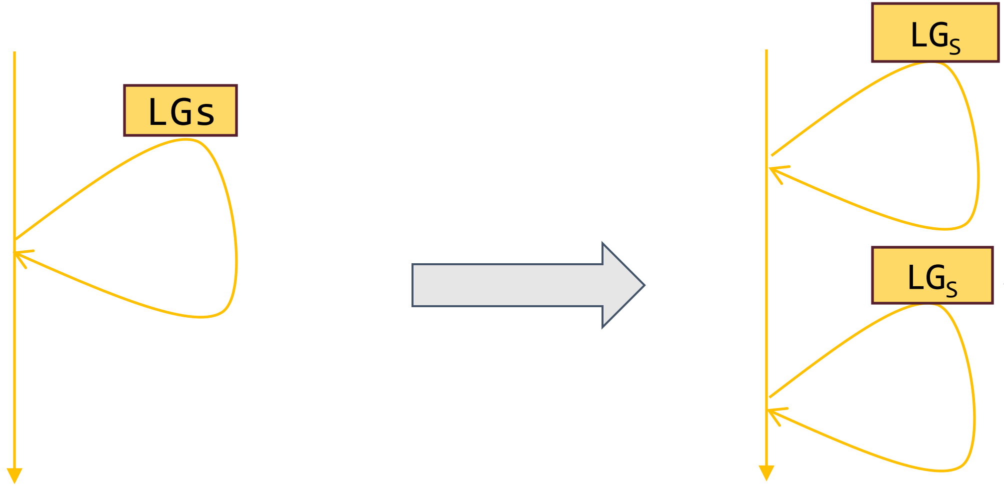
# Decomposition of Source

**Step 1**: create two copies of the source loop



# Decomposition of Source

**Step 1**: create two copies of the source loop




we can strengthen LGs by any predicate

otherwise, this loop is skipped

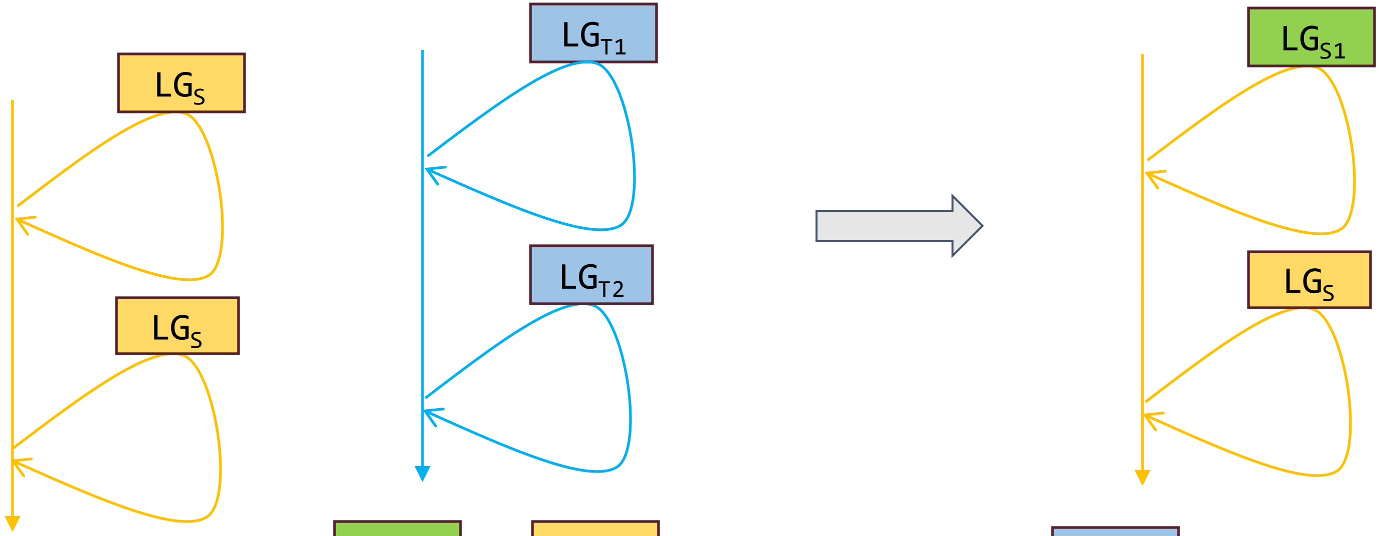
programs are equivalent by construction

* assuming the programs are deterministic

# Decomposition of Source

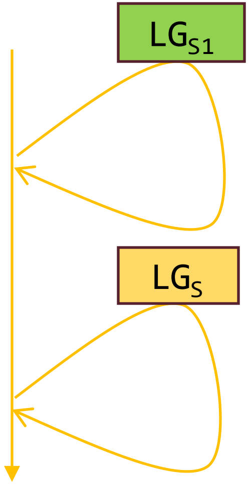
**Step 2**: split iterations among two loops

`mapping: M=X ∧ K=Y ∧ a=c ∧ b=d`



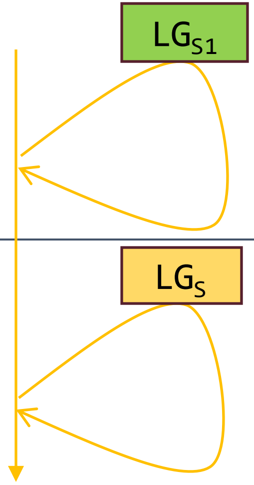

$LG_{S1}$ = $LG_S$ ∧ renameVars( $LG_{T1}$ )

# Decomposed Source



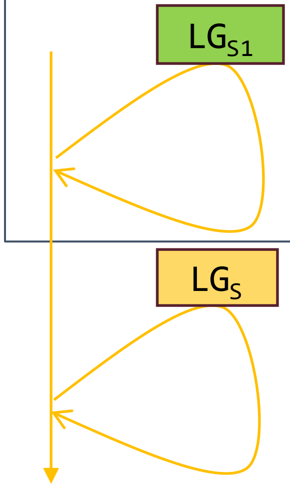

```
int M = nondet(), K = nondet(),
a = 0, N = 2*M+1+K, b = 2*M+1;
assume(M >= 0 && K >= 0);
while (a != N && a < 2*M+1) {
    if (a >= b) b++;
    a++;
}
while (a != N) {
    if (a >= b) b++;
    a++;
}
```

# Decomposed Source




```
int M = nondet(), K = nondet(),
a = 0, N = 2*M+1+K, b = 2*M+1;
assume(M >= 0 && K >= 0);
while (a != N && a < 2*M+1) {
    if (a >= b) b++;
    a++;
}
while (a != N) {
    if (a >= b) b++;
    a++;
}
```

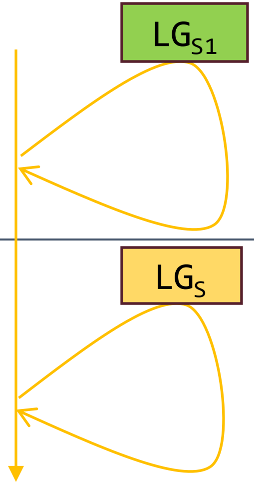
# Decomposed Source




```
int M = nondet(), K = nondet(),
a = 0, N = 2*M+1+K, b = 2*M+1;
assume(M >= 0 && K >= 0);
while (a != N && a < 2*M+1) {
      if (a >= b) b++;
      a++;
}
while (a != N) {
      if (a >= b) b++;
      a++;
}
```

added predicate

# Decomposed Source

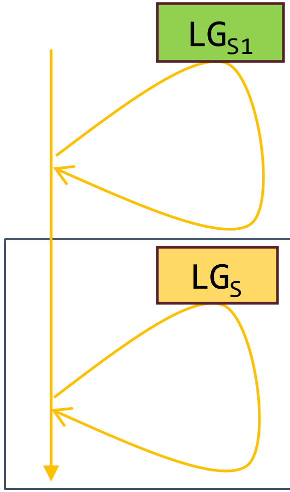



```
int M = nondet(), K = nondet(),
a = 0, N = 2*M+1+K, b = 2*M+1;
assume(M >= 0 && K >= 0);
while (a != N && a < 2*M+1) {
     if (a >= b) b++;
     a++;
}
while (a != N) {
     if (a >= b) b++;
     a++;
}
```

added predicate

```
renameVars(c<2*X+1)
         = a<2*M+1
```

# Decomposed Source

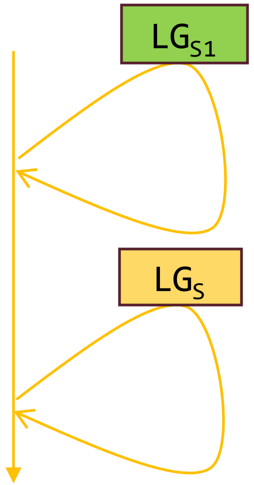


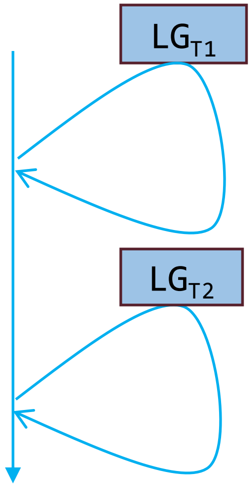

```
int M = nondet(), K = nondet(),
a = 0, N = 2*M+1+K, b = 2*M+1;
assume(M >= 0 && K >= 0);
while (a != N && a < 2*M+1) {
     if (a >= b) b++;
     a++;
}
while (a != N) {
     if (a >= b) b++;
     a++;
}
```

added predicate

```
renameVars(c<2*X+1)
         = a<2*M+1
```

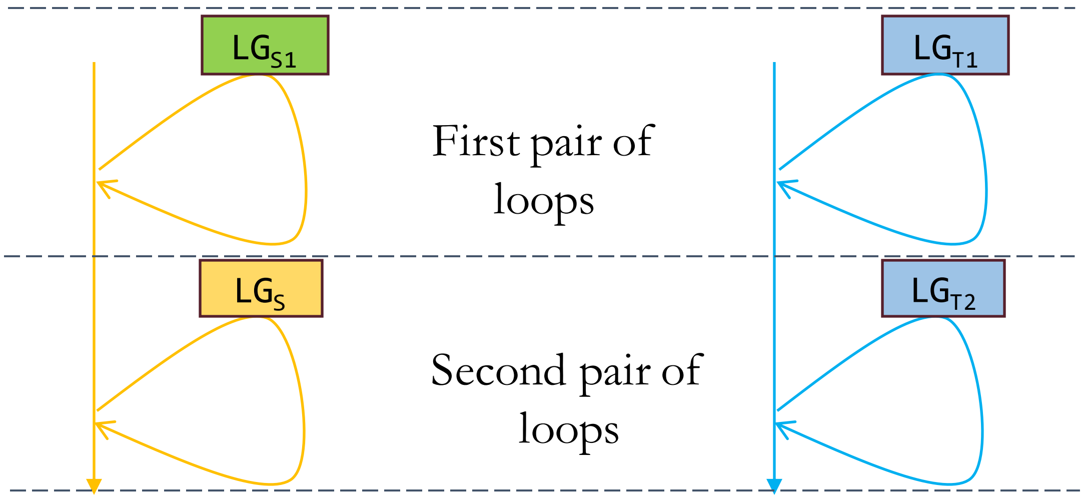
# Comparing Decomposed Source and Target



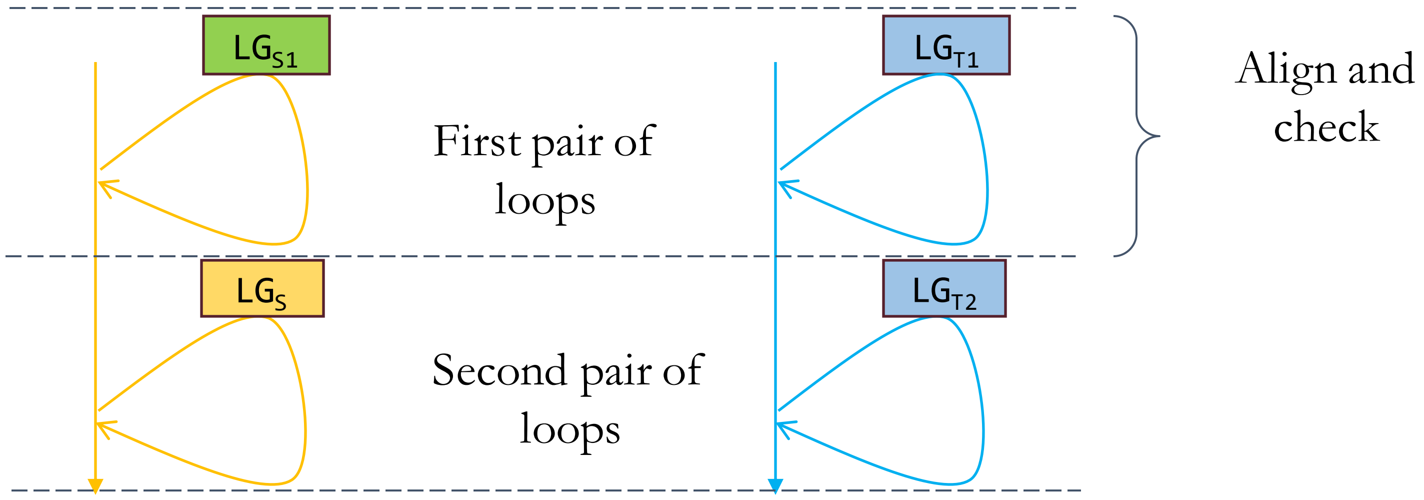

Equivalence:

# Comparing Decomposed Source and Target



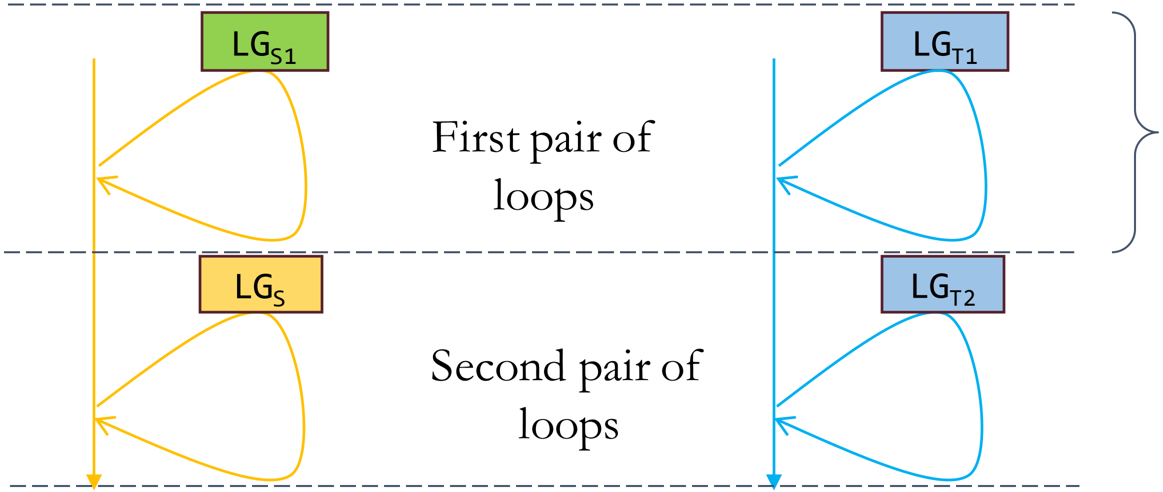

Equivalence:

# Comparing Decomposed Source and Target




Equivalence:

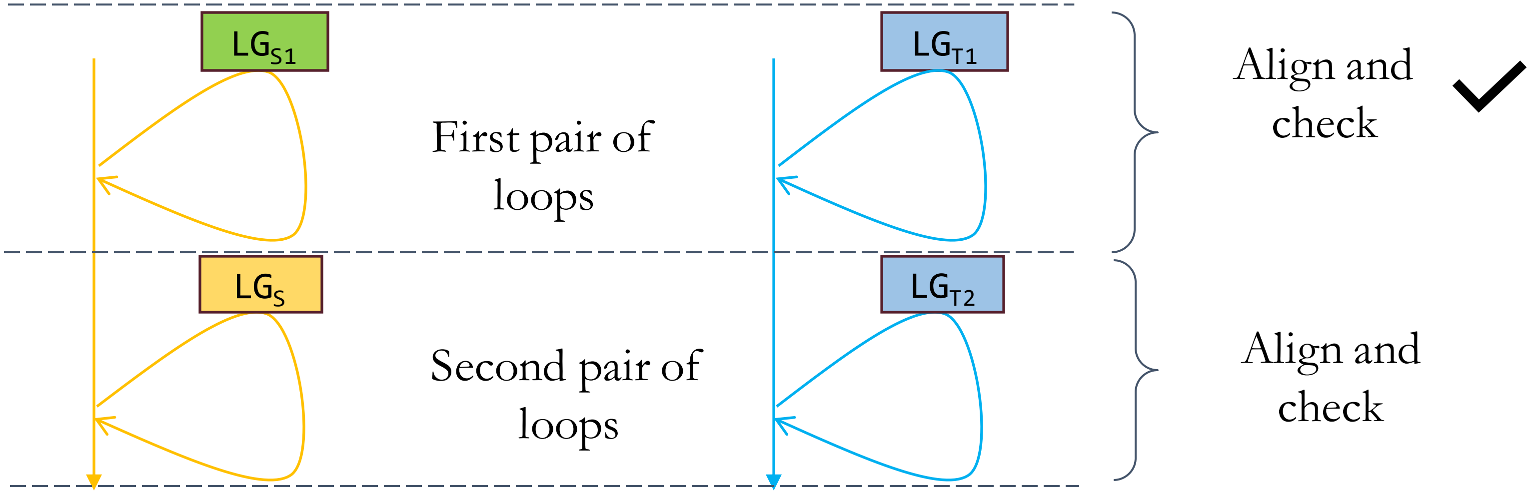
# Comparing Decomposed Source and Target




Align and check ✓

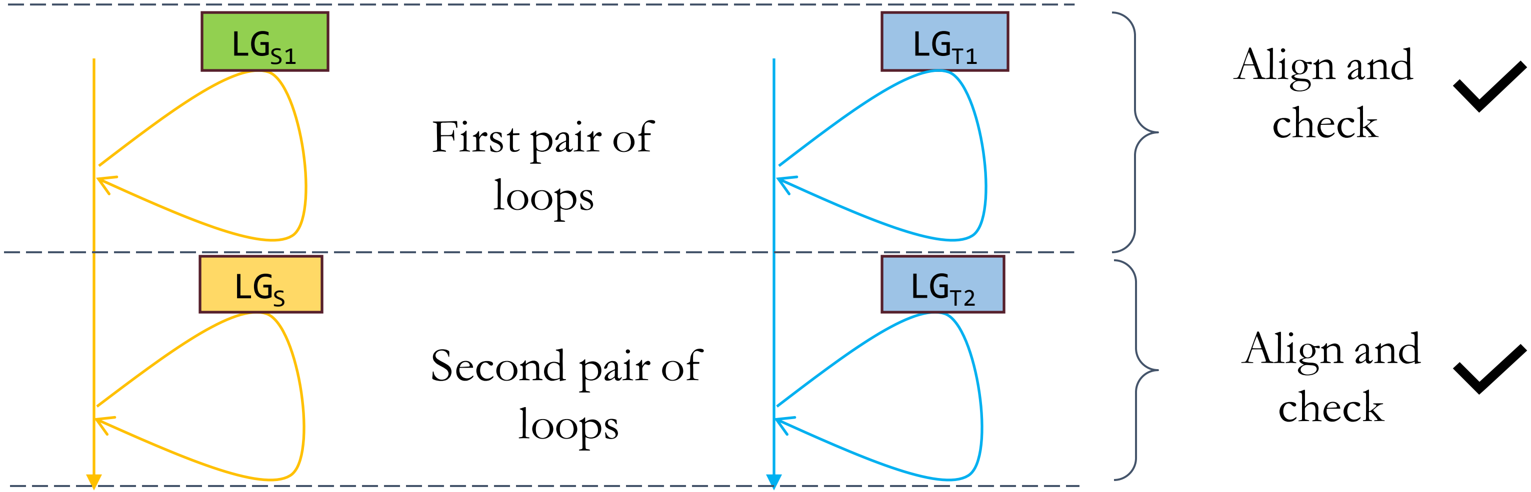
Equivalence:

# Comparing Decomposed Source and Target



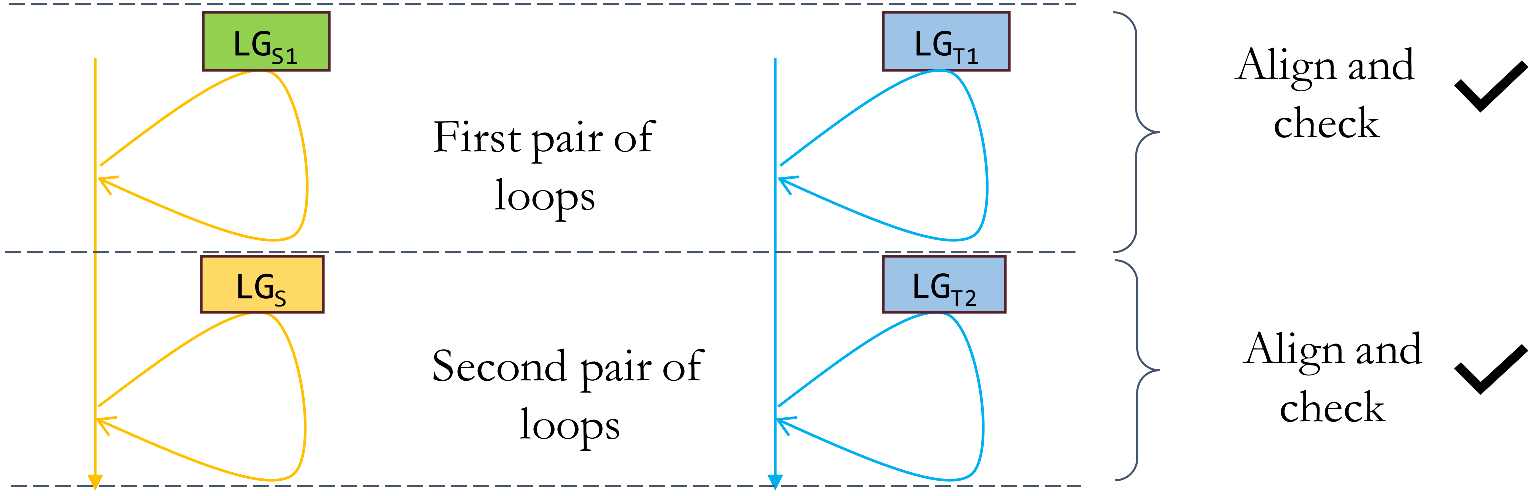

Equivalence:

# Comparing Decomposed Source and Target



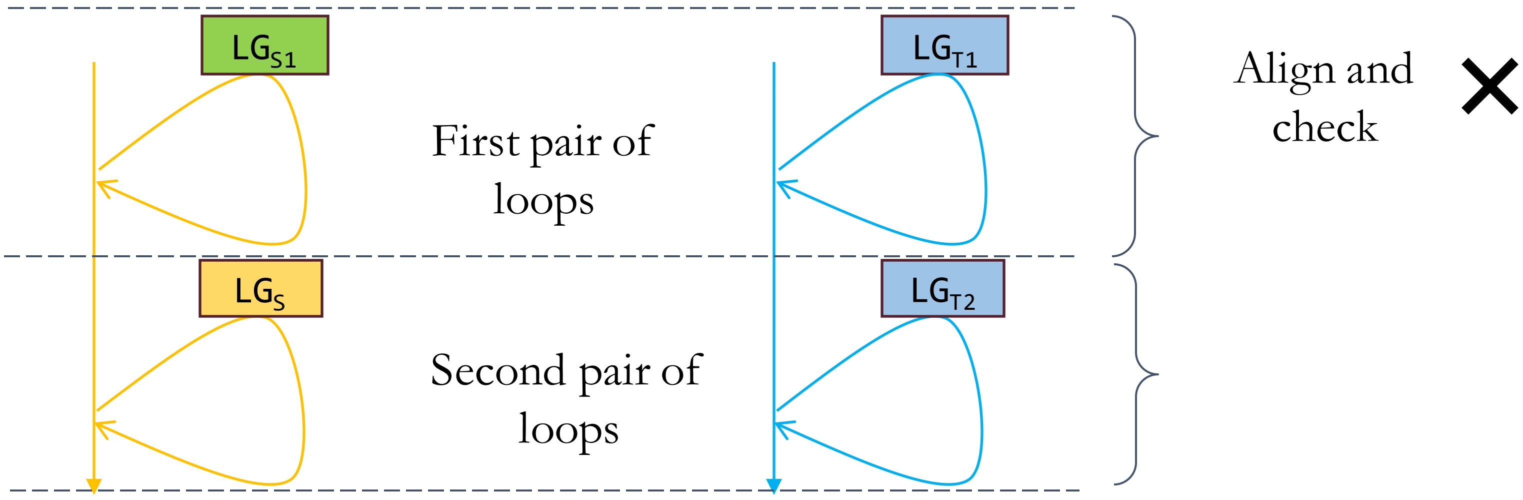

Equivalence:

# Comparing Decomposed Source and Target



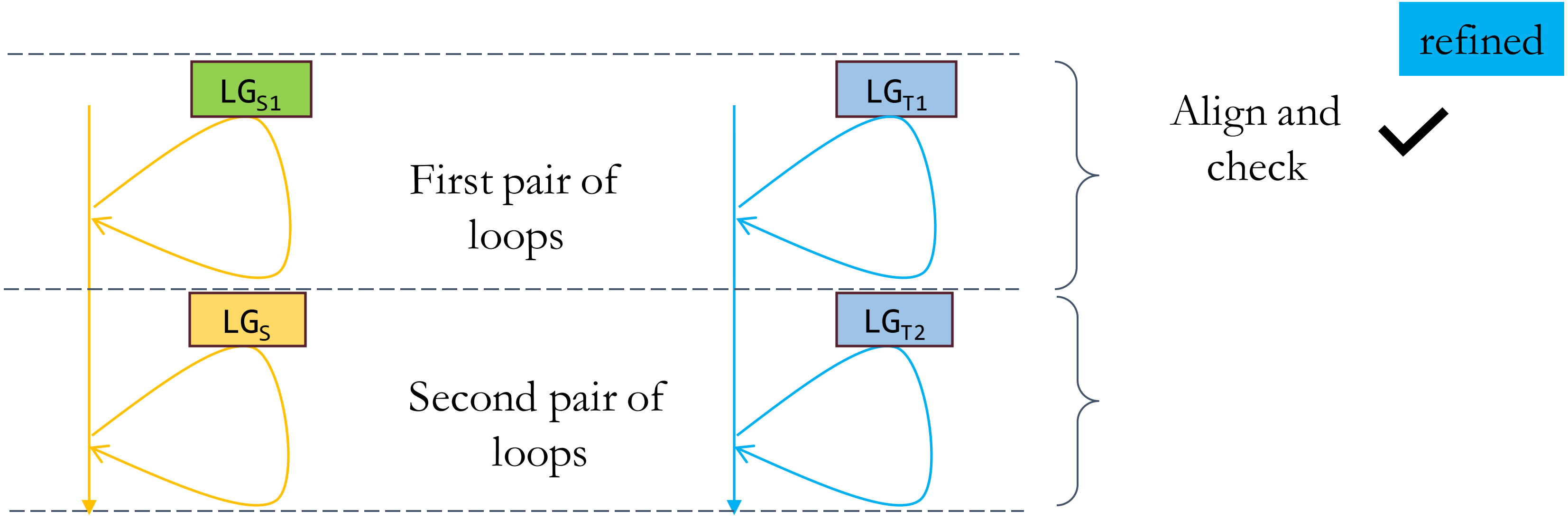

Equivalence: ✓

# Comparing Decomposed Source and Target




Equivalence:

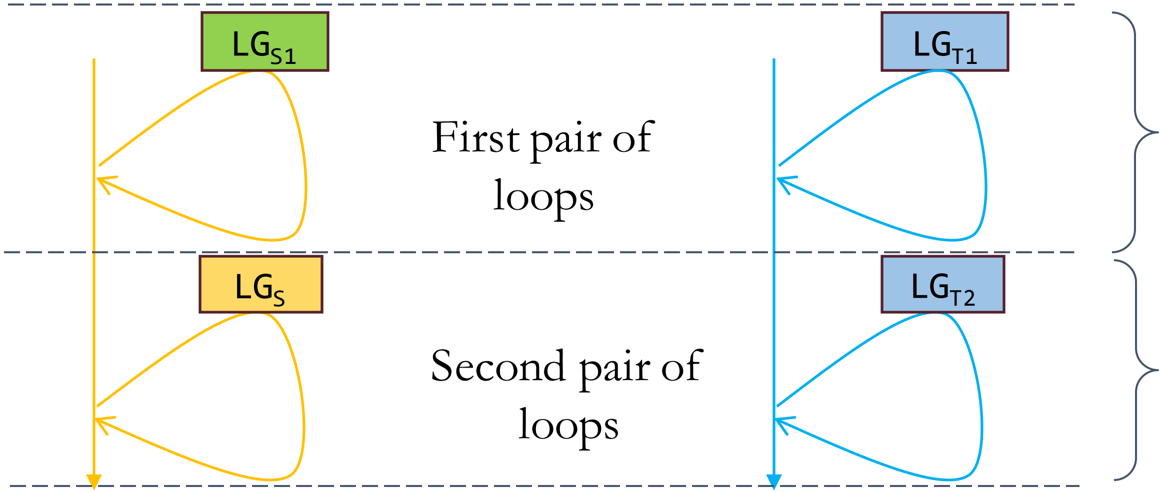
# Comparing Decomposed Source and Target




Equivalence:

# Comparing Decomposed Source and Target

refined



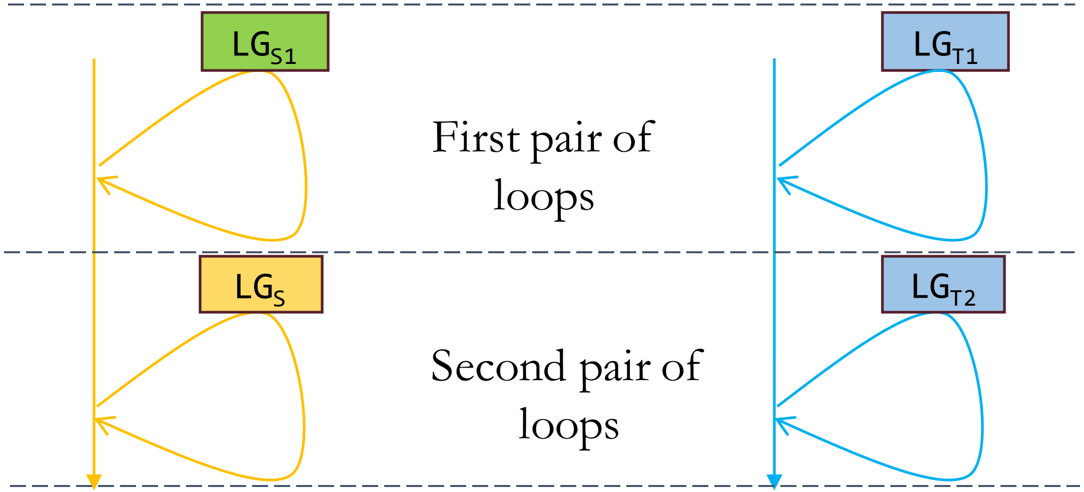

Align and check ✓

Align and check ✓

Equivalence:

# Comparing Decomposed Source and Target

refined

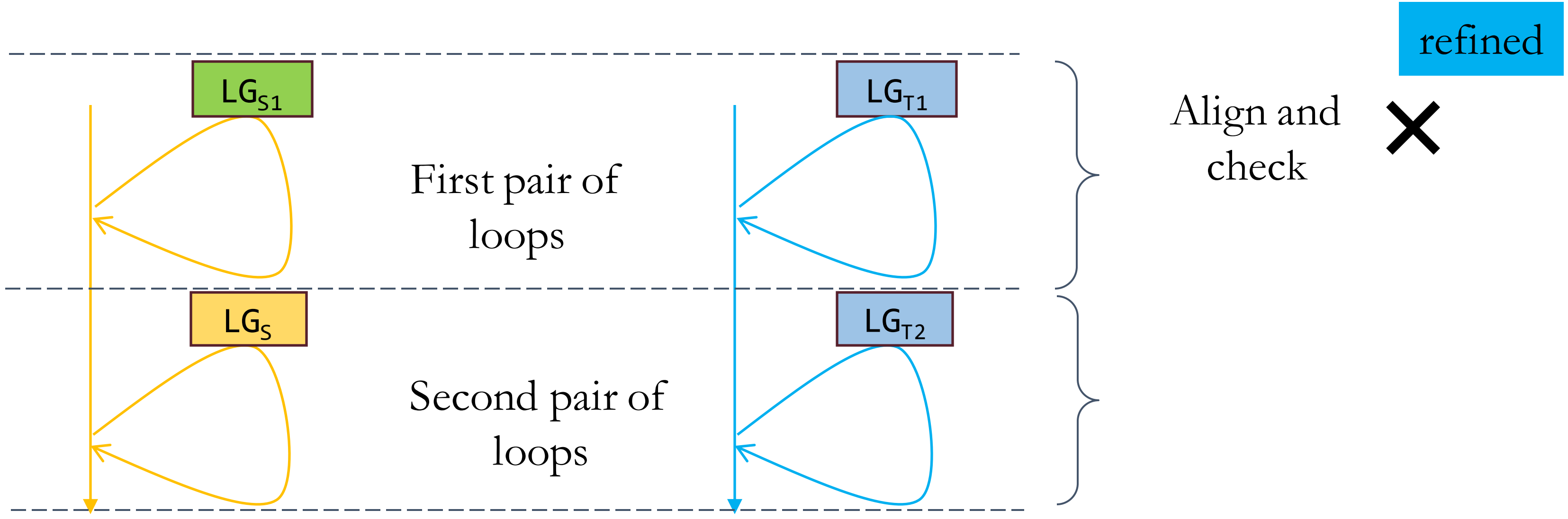



Align and check ✓

Align and check ✓

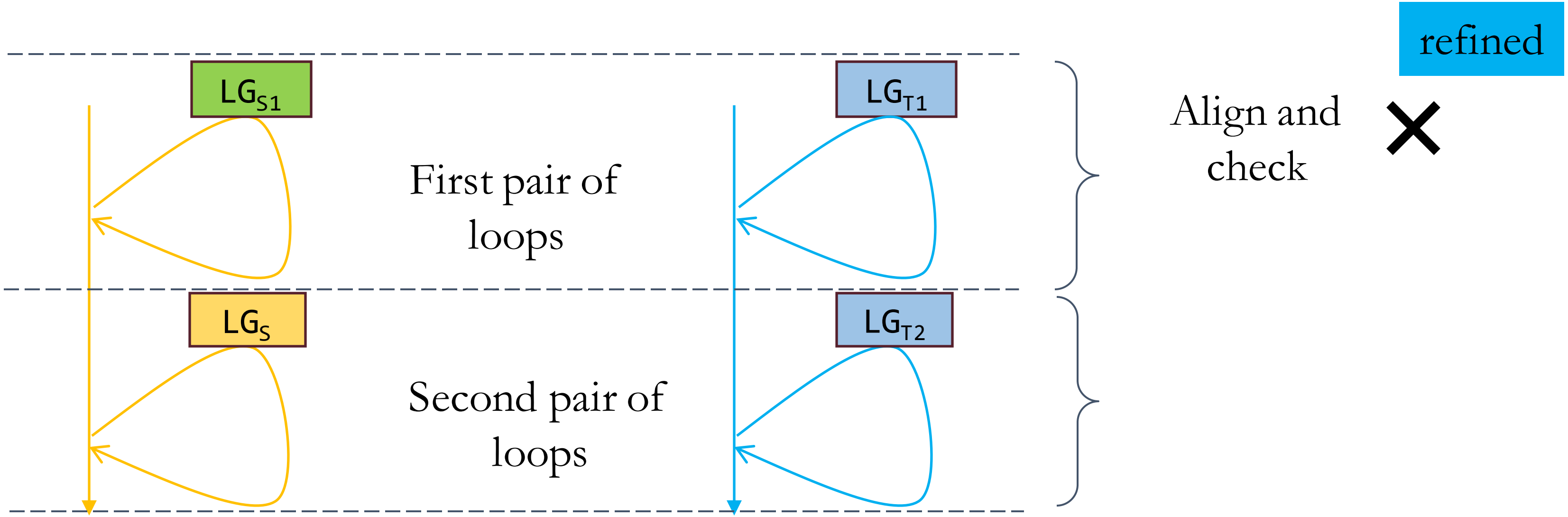
Equivalence: ✓

# Comparing Decomposed Source and Target



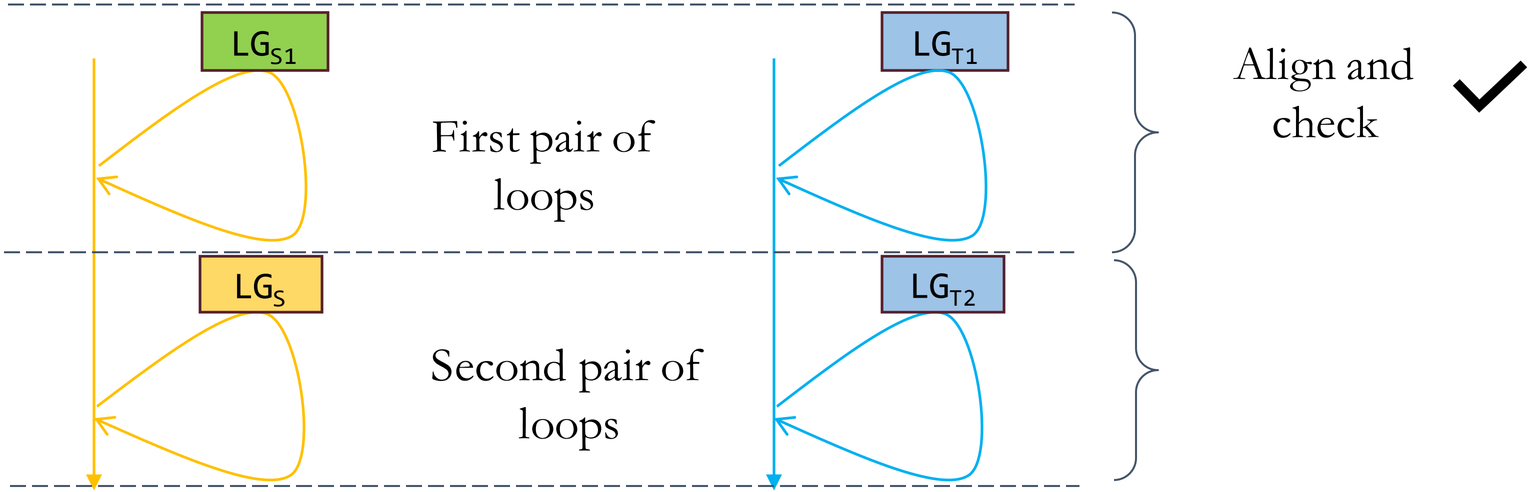

Equivalence:

# Comparing Decomposed Source and Target



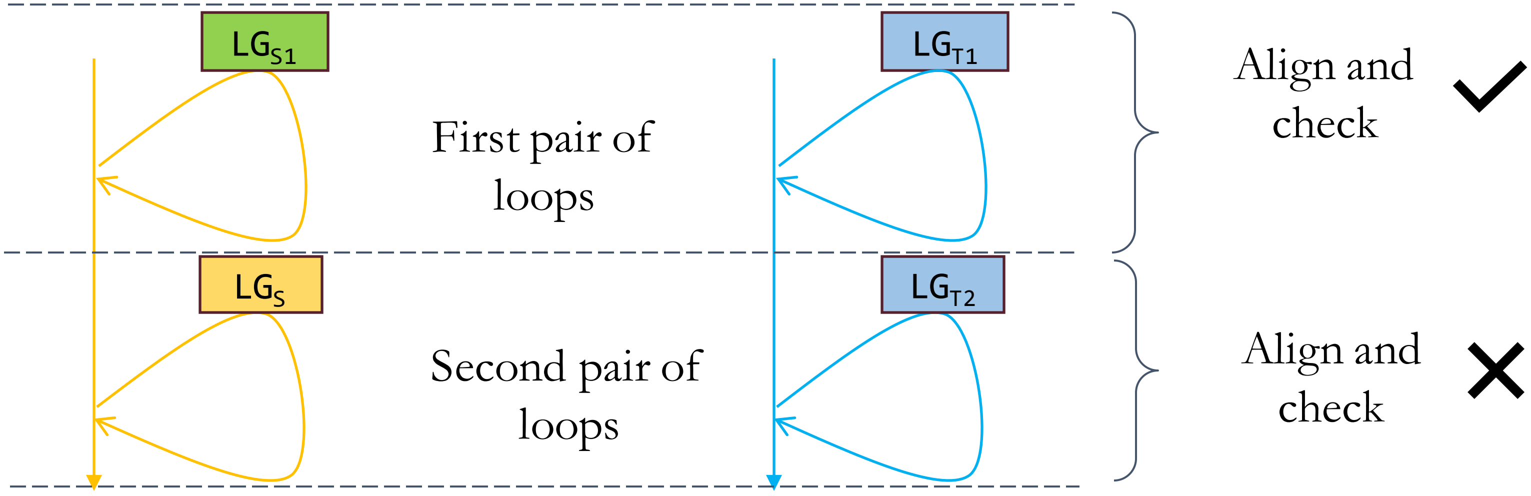

Equivalence: unknown

# Comparing Decomposed Source and Target



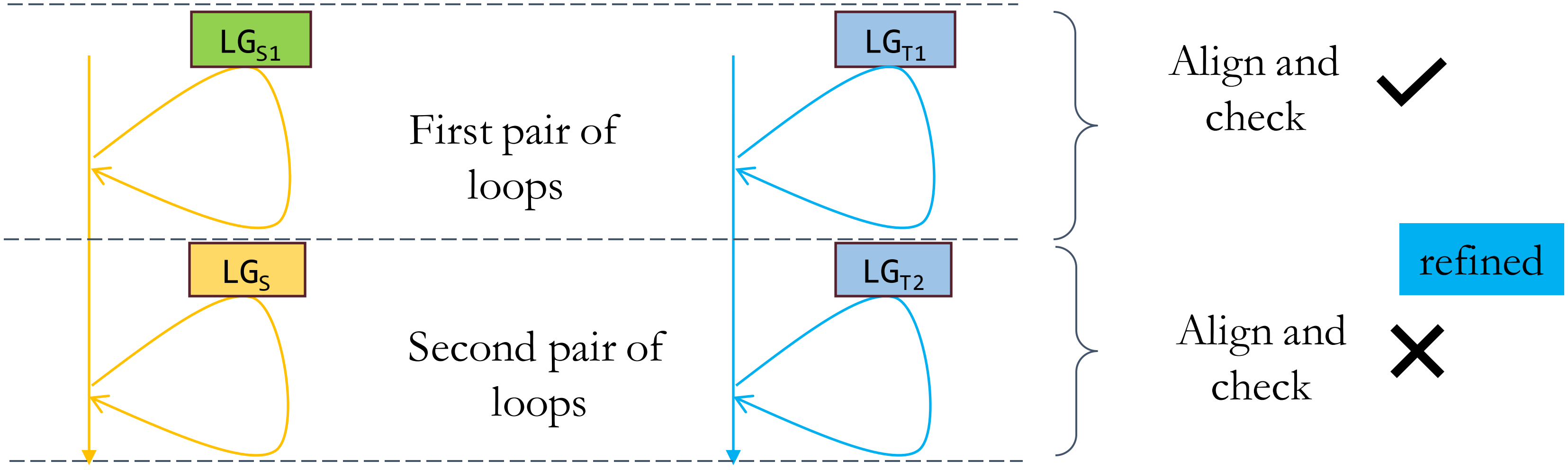

Equivalence:

# Comparing Decomposed Source and Target



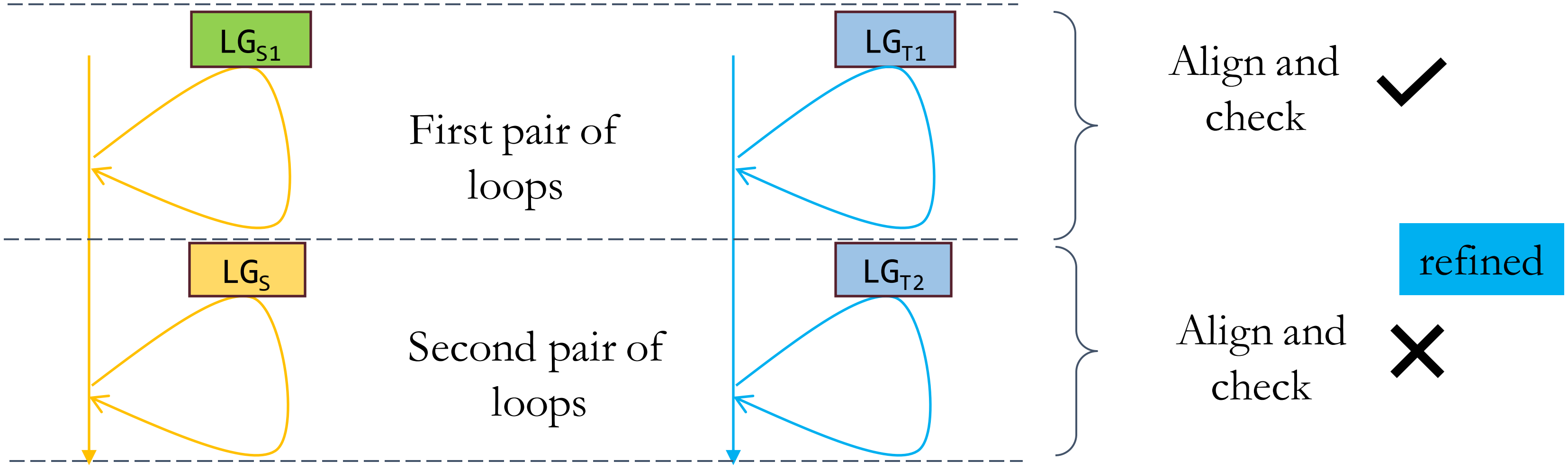

Equivalence:

# Comparing Decomposed Source and Target




Equivalence:

# Comparing Decomposed Source and Target



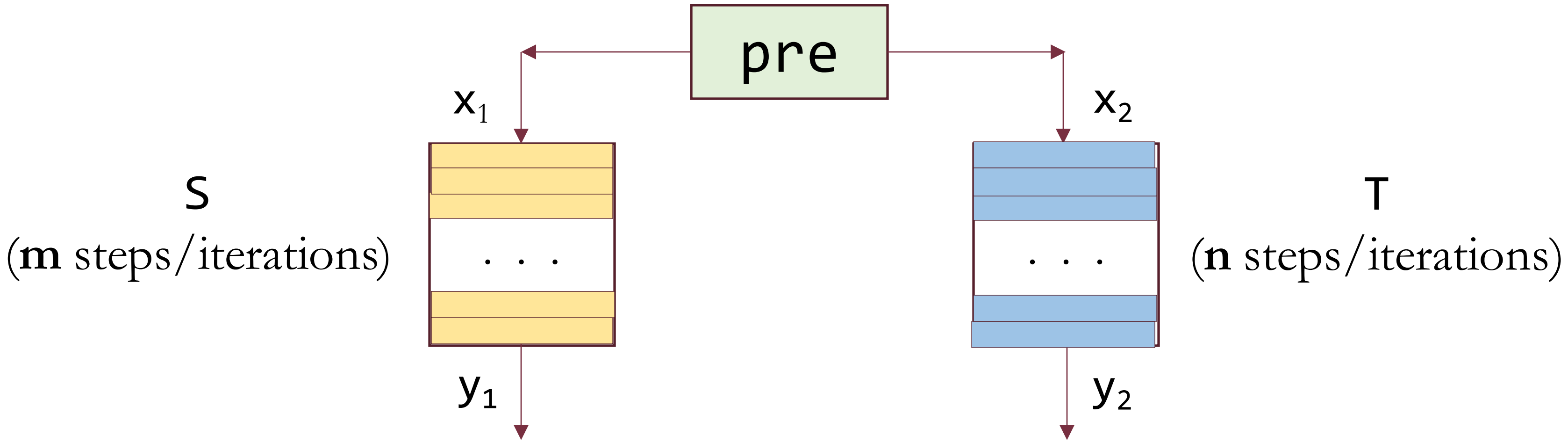

Equivalence: unknown

# Equivalence Checking

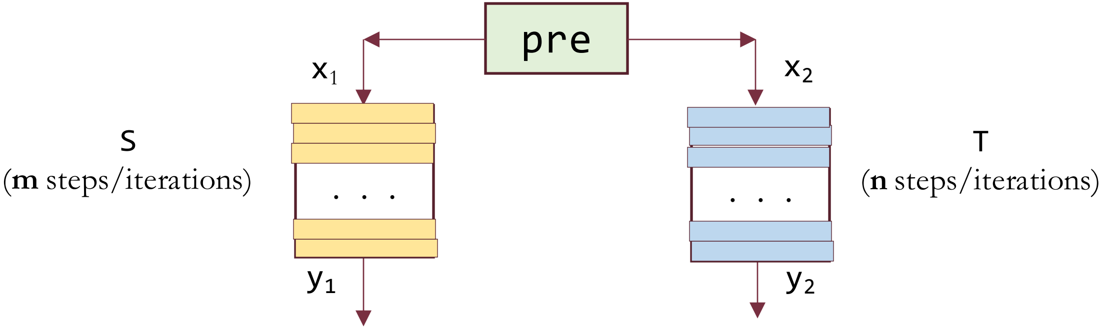
- Equivalence checking of (single loop) programs S and T can be reduced to safety verification of a **product program P**
  - P computes exactly what S and T compute [Barthe et al., FM'11]
  - P begins in a state satisfying a relational *pre*-condition
  - At the end of P, a relational *post*-condition should hold
- **Lockstep composition** of programs facilitates an automated construction of a product program

# Lockstep Composition

Both programs have **same** number of steps (**n** = **m**)



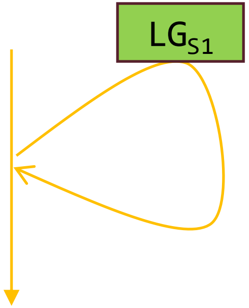
# Product Program

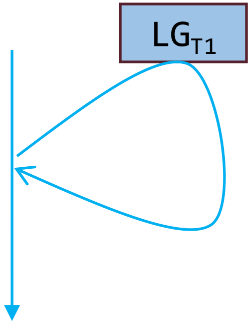



# Example (cont.) – First Pair of Loops

Check **lockstep composability**

**pre**: $M=X \land K=Y \land a=c \land b=d$



# Example (cont.) – First Pair of Loops

Check **lockstep composability**

**pre**: M=X ∧ K=Y ∧ a=c ∧ b=d

```
int M = nondet(), K = nondet(),
a = 0, N = 2*M+1+K, b = 2*M+1;
assume(M >= 0 && K >= 0);
while (a != N && a < 2*M+1) {
    if (a >= b) b++;
    a++;
}
```

```
int X = nondet(), Y = nondet(),
c = 1, d = 2*X+1;
assume(X >= 0 && Y >= 0);
while (c < 2*X+1) {
    c += 2;
}
```

# Example (cont.) – First Pair of Loops

Check **lockstep composability**

**pre**: M=X ∧ K=Y ∧ a=c ∧ b=d

**pre** is inconsistent with **init**s

```
int M = nondet(), K = nondet(),
a = 0, N = 2*M+1+K, b = 2*M+1;
assume(M >= 0 && K >= 0);
while (a != N && a < 2*M+1) {
    if (a >= b) b++;
    a++;
}
```

```
int X = nondet(), Y = nondet(),
c = 1, d = 2*X+1;
assume(X >= 0 && Y >= 0);
while (c < 2*X+1) {
    c += 2;
}
```

# Example (cont.) – First Pair of Loops

Check **lockstep composability**

**pre**: M=X ∧ K=Y ∧ a=c ∧ b=d

**pre** is inconsistent with **init**s

```
int M = nondet(), K = nondet(),
a = 0, N = 2*M+1+K, b = 2*M+1;
assume(M >= 0 && K >= 0);
while (a != N && a < 2*M+1) {
    if (a >= b) b++;
    a++;
}
```

increments a by 1

```
int X = nondet(), Y = nondet(),
c = 1, d = 2*X+1;
assume(X >= 0 && Y >= 0);
while (c < 2*X+1) {
    c += 2;
}
```

increments c by 2

# Example (cont.) – First Pair of Loops

Check **lockstep composability**

**pre**: M=X ∧ K=Y ∧ a=c ∧ b=d

**pre** is inconsistent with **init**s

```
int M = nondet(), K = nondet(),
a = 0, N = 2*M+1+K, b = 2*M+1;
assume(M >= 0 && K >= 0);
while (a != N && a < 2*M+1) {
     if (a >= b) b++;
     a++;
}
```

increments a by 1

```
int X = nondet(), Y = nondet(),
c = 1, d = 2*X+1;
assume(X >= 0 && Y >= 0);
while (c < 2*X+1) {
     c += 2;
}
```

increments c by 2

Lockstep composability check **fails**

# Example (cont.) – First Pair of Loops

Check **lockstep composability**

**pre**: M=X ∧ K=Y ∧ a=c ∧ b=d

**pre** is inconsistent with **init**s

```
int M = nondet(), K = nondet(),
a = 0, N = 2*M+1+K, b = 2*M+1;
assume(M >= 0 && K >= 0);
while (a != N && a < 2*M+1) {
     if (a >= b) b++;
     a++;
}
```

increments a by 1

```
int X = nondet(), Y = nondet(),
c = 1, d = 2*X+1;
assume(X >= 0 && Y >= 0);
while (c < 2*X+1) {
     c += 2;
}
```

increments c by 2

need alignment of the source and target loop

# Automated Finding of Alignment of Loops

- Find exact **number of iterations** as a function of input variables
  - A hard problem, but easier for loops with **induction variables**
  - Induction variables have: 1) static lower and upper bounds,
    2) iterator increases (or decreases) monotonically by constant value
- **Rearrange** source to match number of iterations in target loop

# Automated Finding of Alignment of Loops

- Find exact **number of iterations** as a function of input variables
  - A hard problem, but easier for loops with **induction variables**
  - Induction variables have: 1) static lower and upper bounds,
    2) iterator increases (or decreases) monotonically by constant value
- **Rearrange** source to match number of iterations in target loop

For first pair of loops

- # iterations: source loop -- 2*M+1, target loop -- X
- Rearrangement:
  - move 1 iteration in source before the loop
  - for each target loop iteration, perform 2 source loop iterations

# First Pair of Loops

Loops are **lockstep composable**

**pre:** $M=X \land K=Y \land a=c \land b=d$

```
int M = nondet(), K = nondet(),
a = 0, N = 2*M+1+K, b = 2*M+1;
assume(M >= 0 && K >= 0);
if (a >= b) b++;   a++;
while (a != N && a < 2*M+1) {
    if (a >= b) b++;   a++;
    if (a >= b) b++;   a++;
}
```

```
int X = nondet(), Y = nondet(),
c = 1, d = 2*X+1;
assume(X >= 0 && Y >= 0);
while (c < 2*X+1) {
    c += 2;
}
```

# First Pair of Loops

Loops are **lockstep composable**

**pre:** M=X ∧ K=Y ∧ a=c ∧ b=d

```
int M = nondet(), K = nondet(),
a = 0, N = 2*M+1+K, b = 2
assume(M >= 0 && K >= 0);
if (a >= b) b++;   a++;
while (a != N && a < 2*M+1) {
    if (a >= b) b++;   a++;
    if (a >= b) b++;   a++;
}
```

moved 1 iteration before loop

```
ondet(), Y = nondet(),
= 2*X+1;
assume(X >= 0 && Y >= 0);
while (c < 2*X+1) {
    c += 2;
}
```

# First Pair of Loops

Loops are **lockstep composable**

**pre:** M=X ∧ K=Y ∧ a=c ∧ b=d

```
int M = nondet(), K = nondet(),
a = 0, N = 2*M+1+K, b = 2*M+1;
assume(M >= 0 && K >= 0);
if (a >= b) b++;   a++;
while (a != N && a < 2*M+1) {
    if (a >= b) b++;   a++;
    if (a >= b) b++;   a++;
}
```

created a group of 2 iterations

```
int X = nondet(), Y = nondet(),
c = 1, d = 2*X+1;
assume(X >= 0 && Y >= 0);
while (c < 2*X+1) {
    c += 2;
}
```

# First Pair of Loops

Check **equivalence**

```
int M = nondet(), K = nondet(),
a = 0, N = 2*M+1+K, b = 2*M+1;
assume(M >= 0 && K >= 0);
if (a >= b) b++;   a++;
while (a != N && a < 2*M+1) {
    if (a >= b) b++;   a++;
    if (a >= b) b++;   a++;
}
```

```
int X = nondet(), Y = nondet(),
c = 1, d = 2*X+1;
assume(X >= 0 && Y >= 0);
while (c < 2*X+1) {
    c += 2;
}
```

# First Pair of Loops

Check **equivalence**

**pre:** $M=X \land K=Y \land a=c \land b=d$

```
int M = nondet(), K = nondet(),
a = 0, N = 2*M+1+K, b = 2*M+1;
assume(M >= 0 && K >= 0);
if (a >= b) b++;   a++;
while (a != N && a < 2*M+1) {
     if (a >= b) b++;   a++;
     if (a >= b) b++;   a++;
}
```

```
int X = nondet(), Y = nondet(),
c = 1, d = 2*X+1;
assume(X >= 0 && Y >= 0);
while (c < 2*X+1) {
     c += 2;
}
```

**post:** $M=X \land K=Y \land a=c \land b=d$

# First Pair of Loops

Loops are **equivalent**

**pre:** M=X ∧ K=Y ∧ a=c ∧ b=d

```
int M = nondet(), K = nondet(),
a = 0, N = 2*M+1+K, b = 2*M+1;
assume(M >= 0 && K >= 0);
if (a >= b) b++;   a++;
while (a != N && a < 2*M+1) {
     if (a >= b) b++;   a++;
     if (a >= b) b++;   a++;
}
```

```
int X = nondet(), Y = nondet(),
c = 1, d = 2*X+1;
assume(X >= 0 && Y >= 0);
while (c < 2*X+1) {
     c += 2;
}
```

**post:** M=X ∧ K=Y ∧ a=c ∧ b=d ✓

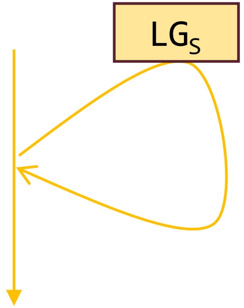
# Equivalence Checking of Single Loops

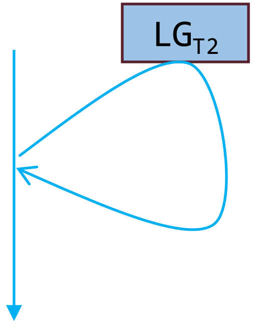
- Automated construction of product program
- Safety verification of the product program
  - Program is safe if there is a safe inductive invariant (INV)
  - INV translates to a relational invariant over two programs
  - Relational invariant => programs are **equivalent**
- Finding inductive invariants is challenging
  - We rely on external SMT-based tools (a.k.a. CHC solvers, e.g., ELDARICA, Spacer, FreqHorn).

# Second Pair of Loops

Check **lockstep composability**

**pre:** M=X ∧ K=Y ∧ a=c ∧ b=d



# Second Pair of Loops

Check **lockstep composability**

**pre:** M=X ∧ K=Y ∧ a=c ∧ b=d

```
while (a != N) {
    if (a >= b) b++;
    a++;
}
```

```
while (c != 2*X+1+Y) {
    d++;
    c++;
}
```

# Second Pair of Loops

Check **lockstep composability**

**pre:** M=X ∧ K=Y ∧ a=c ∧ b=d

what is N?

```
1.  while (a != N) {
2.      if (a >= b) b++;
3.      a++;
4.  }
```

```
1.  while (c != 2*X+1+Y) {
2.      d++;
3.      c++;
4.  }
```

# Second Pair of Loops

Check **lockstep composability**

**pre:** M=X ∧ K=Y ∧ a=c ∧ b=d

what is N?

```
while (a != N) {
    if (a >= b) b++;
    a++;
}
```

```
while (c != 2*X+1+Y) {
    d++;
    c++;
}
```

Lockstep composability **fails** because **N** is not known

# Second Pair of Loops

Check **lockstep composability**

**pre:** M=X ∧ K=Y ∧ a=c ∧ b=d

what is N?

```
while (a != N) {
    if (a >= b) b++;
    a++;
}
```

```
while (c != 2*X+1+Y) {
    d++;
    c++;
}
```

Lockstep composability **fails** because **N** is not known

We receive a counterexample cex

# Second Pair of Loops

Check **lockstep composability**

**pre:** M=X ∧ K=Y ∧ a=c ∧ b=d

what is N?

```
1.  while (a != N) {
2.      if (a >= b) b++;
3.      a++;
4.  }
```

```
1.  while (c != 2*X+1+Y) {
2.      d++;
3.      c++;
4.  }
```

Using **cex**, we want to refine source with value of **N**

We receive a counterexample cex

# Refinement

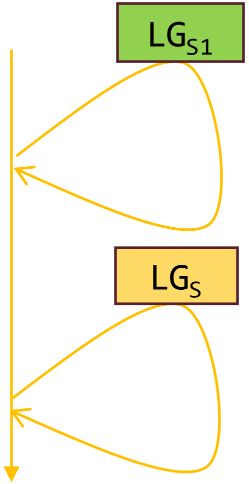
- **Saturate** the verification conditions in a program by useful program properties
  - Driven by counterexamples
- Refinement is needed when:
  - our model loses information due to decomposition
  - constant propagation has been applied in target
- We propagate properties available earlier in the program, to strengthen source and target in later parts (being analyzed)

# Example (cont.) – Second Pair of Loops

**Refinement** of source

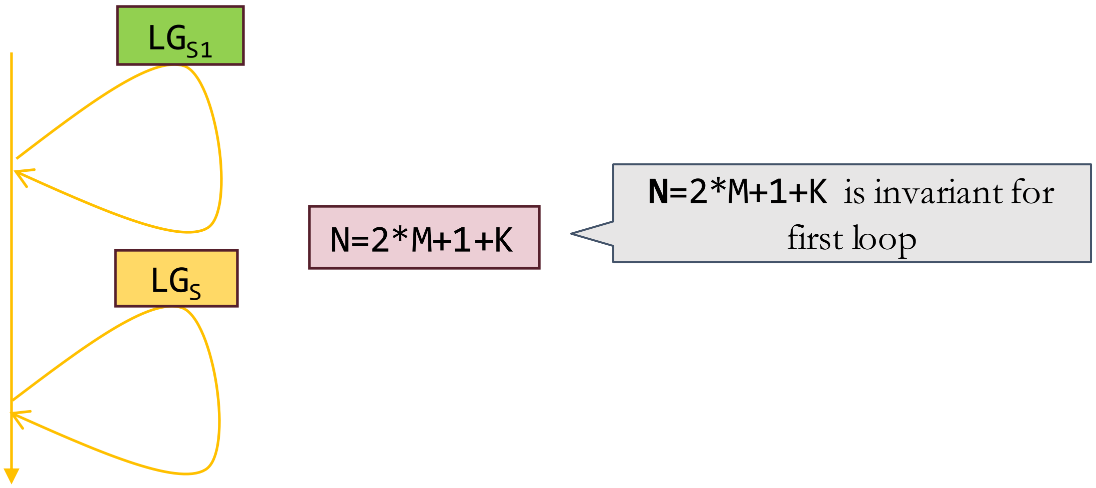
N=2*M+1+K

**N** is known initially



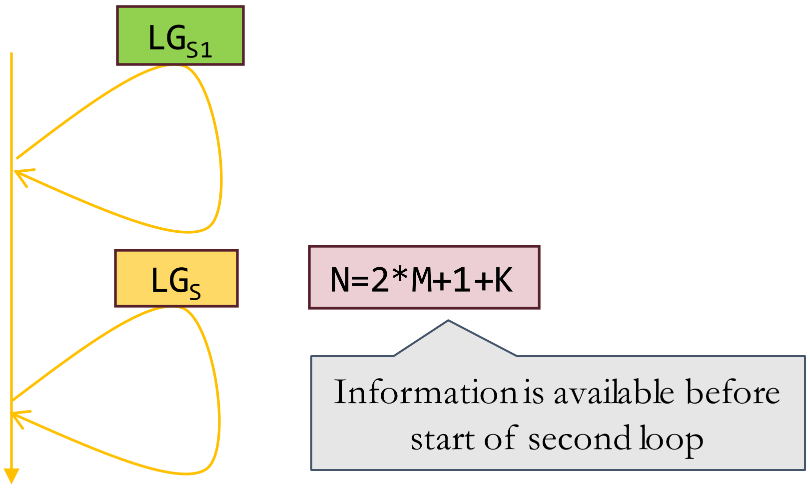
# Example (cont.) – Second Pair of Loops

**Refinement** of source



# Example (cont.) – Second Pair of Loops

**Refinement** of source



# Second Pair of Loops

Loops are **lockstep composable**

**pre:** M=X ∧ K=Y ∧ a=c ∧ b=d

```
assume(N == 2*M+1+K);
while (a != N) {
    if (a >= b) b++;
    a++;
}
```

```
while (c != 2*X+1+Y) {
    d++;
    c++;
}
```

# Second Pair of Loops

Loops are **lockstep composable**

**pre:** $M=X \wedge K=Y \wedge a=c \wedge b=d$

Refinement added

```
assume(N == 2*M+1+K);
while (a != N) {
    if (a >= b) b++;
    a++;
}
```

```
while (c != 2*X+1+Y) {
    d++;
    c++;
}
```

# Second Pair of Loops

Check **equivalence**

**pre:** $M=X \land K=Y \land a=c \land b=d$

```
assume(N == 2*M+1+K);
while (a != N) {
     if (a >= b) b++;
     a++;
}
```

```
while (c != 2*X+1+Y) {
     d++;
     c++;
}
```

**post:** $M=X \land K=Y \land a=c \land b=d$

# Second Pair of Loops

## Equivalence check **fails**

**pre:** $M=X \land K=Y \land a=c \land b=d$

```
assume(N == 2*M+1+K);
while (a != N) {
    if (a >= b) b++;
    a++;
}
```

```
while (c != 2*X+1+Y) {
    d++;
    c++;
}
```

**post:** $M=X \land K=Y \land a=c \land b=d$ ✗

# Second Pair of Loops

**Equivalence** check **fails**

**pre:** M=X ∧ K=Y ∧ a=c ∧ b=d

```
assume(N == 2*M+1+K);
while (a != N) {
      if (a >= b) b++;
      a++;
}
```

we do not know if `a >= b`

```
while (c != 2*X+1+Y) {
      d++;
      c++;
}
```

**post:** M=X ∧ K=Y ∧ a=c ∧ b=d ✕

# Second Pair of Loops

**Equivalence** check **fails**

**pre:** M=X ∧ K=Y ∧ a=c ∧ b=d

```
assume(N == 2*M+1+K);
while (a != N) {
    if (a >= b) b++;
    a++;
}
```

we do not know if `a >= b`

```
while (c != 2*X+1+Y) {
    d++;
    c++;
}
```

**post:** M=X ∧ K=Y ∧ a=c ∧ b=d ✗

We need another **refinement** using **cex** received

# Second Pair of Loops

Loops are **equivalent**

**pre:** M=X ∧ K=Y ∧ a=c ∧ b=d

```
assume(N == 2*M+1+K);
assume(b == 2*M+1);
while (a != N) {
    if (a >= b) b++;
    a++;
}
```

```
while (c != 2*X+1+Y) {
    d++;
    c++;
}
```

**post:** M=X ∧ K=Y ∧ a=c ∧ b=d ✓

# Second Pair of Loops

Loops are **equivalent**

**pre:** M=X ∧ K=Y ∧ a=c ∧ b=d

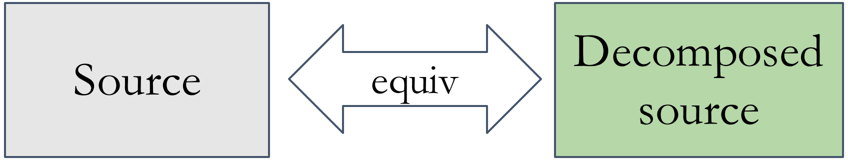
Refinement added

```
assume(N == 2*M+1+K);
assume(b == 2*M+1);
while (a != N) {
     if (a >= b) b++;
     a++;
}
```

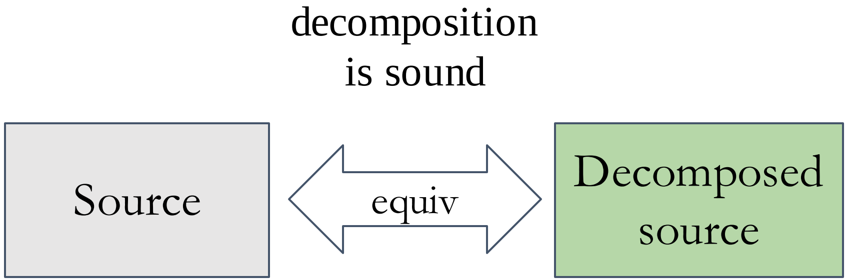
```
while (c != 2*X+1+Y) {
     d++;
     c++;
}
```

**post:** M=X ∧ K=Y ∧ a=c ∧ b=d ✓

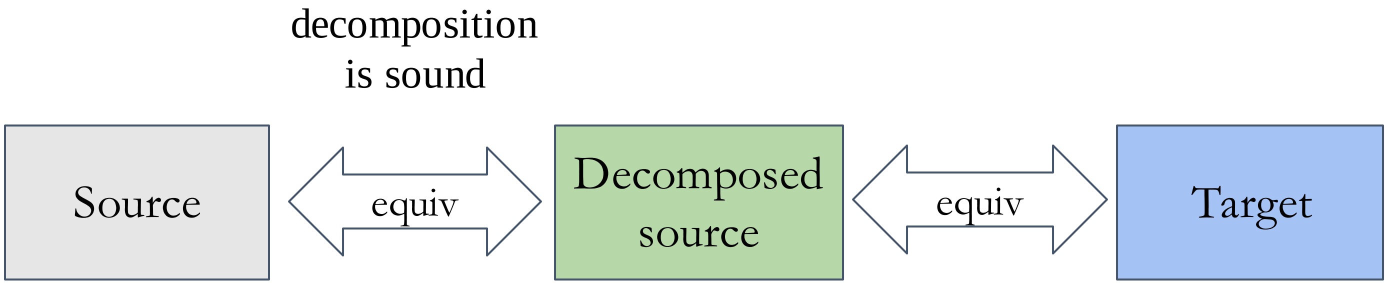
# Equivalence Checking



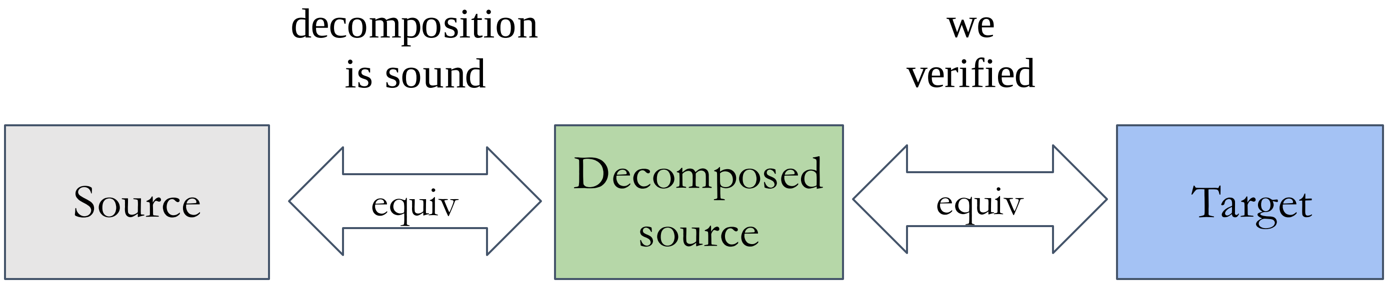
# Equivalence Checking



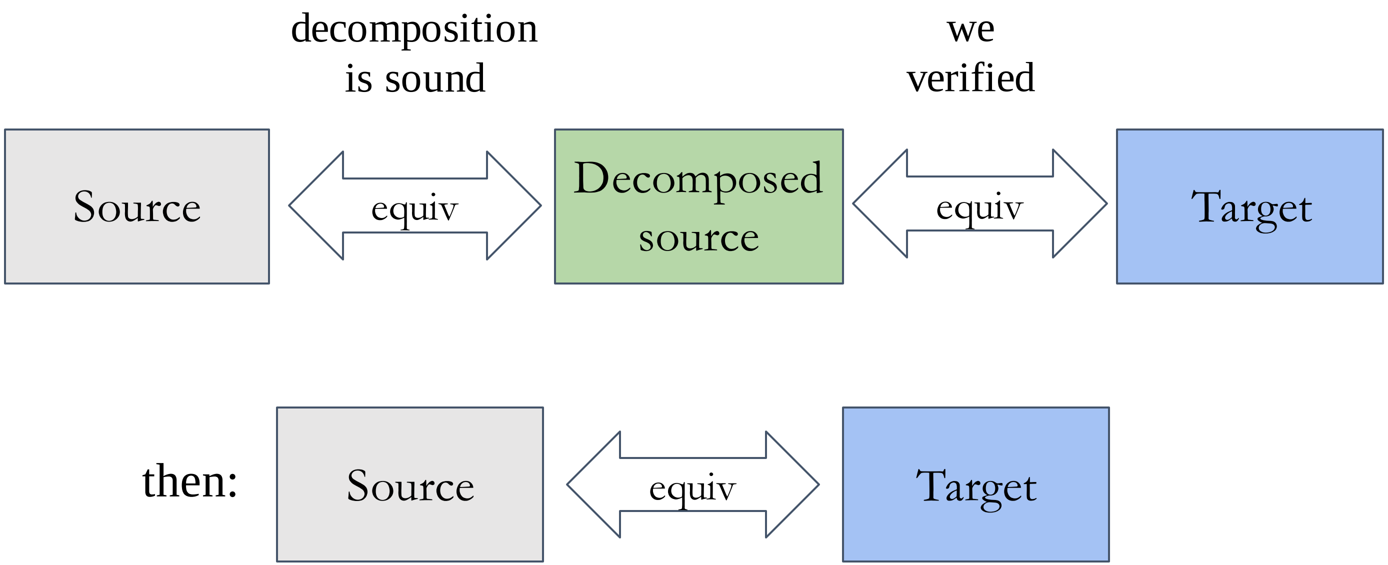
# Equivalence Checking



# Equivalence Checking



# Equivalence Checking



# Implementation

- Implemented in ALIEN tool
- Programs are represented using Constrained Horn Clauses (CHCs) – all operations done on CHCs
- Implemented on top of the FreqHorn CHC solver

[G. Fedyukovich, et al, FMCAD'17]

- ALIEN uses Z3 as SMT solver [L. de Moura, N. Bjørner, TACAS'08]

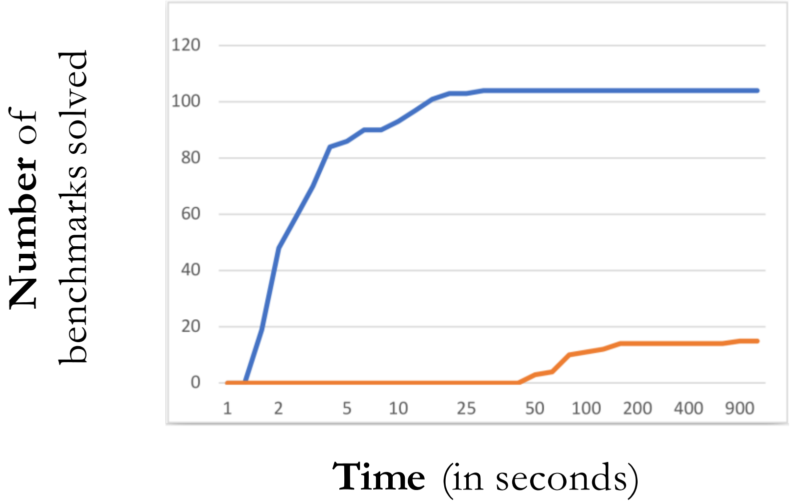
# Evaluation

- We check the equivalence of source/target programs from:
  - Test Suite of Vectorizing Compilers (TSVC) [S. Maleki et al., PACT'11]
    - 104 benchmarks
    - All have a single loop, unrolling+peeling
  - Multi-phase benchmarks [D. Riley, G. Fedyukovich, FSE'22]
    - 24 benchmarks
    - 2-3 loops, loop unswitching transformation
- Compared to COUNTER [S. Gupta et al., OOPSLA'20]
  - CounterExample-Guided Translation Validation tool that computes bisimulations between intermediate points of two programs and generates invariants

# Evaluation

ALIEN
COUNTER

- ALIEN solved **103**
- COUNTER solved **15**

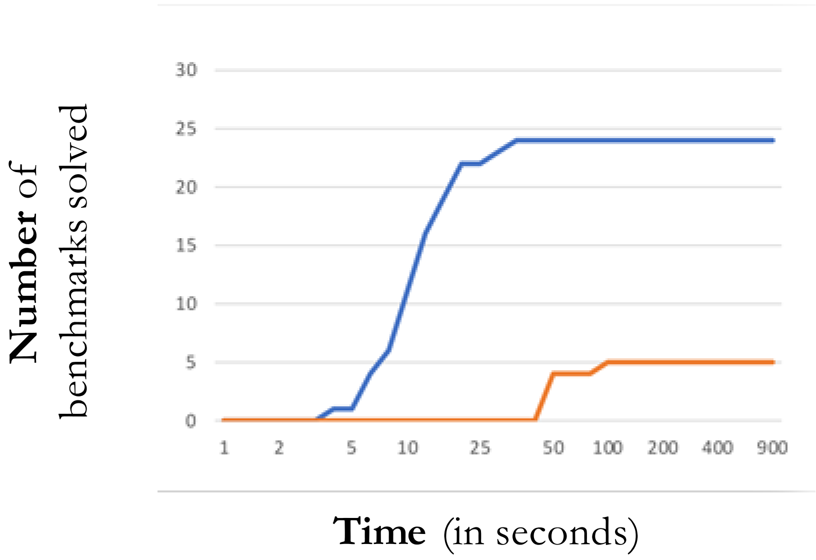



TSVC benchmarks (104 benchmarks)

# Evaluation

ALIEN
COUNTER

- ALIEN solved **24**
- COUNTER solved **5**




Multi-phase benchmarks

# Conclusion. Thank you!

- We present an automated technique for Equivalence Checking of programs with unbalanced loops based on Decomposition, Refinement, and Alignment techniques
- ALIEN performs order of magnitudes faster than COUNTER
- In future,
  - multiple loops in source as well
  - support nested loops
  - support for benchmarks that require universally quantified invariants