Effective Abstractions for Verification under Relaxed Memory Models

Andrei Dan
ETH Zurich

Yuri Meshman
Technion

Martin Vechev
ETH Zurich

Eran Yahav
Technion

Dekker's Algorithm

initial: flag[0] = false, flag[1] = false, turn = 0

Thread 0:

```
flag[0] := true
while  (flag[1] = true)
   if  (turn ≠ 0)
      flag[0] := false
      while  (turn ≠ 0) { }
      flag[0] := true

/* Critical Section */
```

Thread 1:

```
flag[1] := true
while  (flag[0] = true)
   if  (turn ≠ 1)
      flag[1] := false
      while  (turn ≠ 1) { }
      flag[1] := true

/* Critical Section */
```

Spec: mutual exclusion over Critical Section

Dekker's Algorithm

Sequential Consistency

initial: flag[0] = false, flag[1] = false, turn = 0

Thread 0:

```
flag[0] := true
while  (flag[1] = true)
   if  (turn ≠ 0)
      flag[0] := false
      while  (turn ≠ 0) { }
      flag[0] := true

/* Critical Section */
```

Thread 1:

```
flag[1] := true
while  (flag[0] = true)
   if  (turn ≠ 1)
      flag[1] := false
      while  (turn ≠ 1) { }
      flag[1] := true

/* Critical Section */
```

Spec: mutual exclusion over Critical Section

Dekker's Algorithm

Sequential Consistency

initial: flag[0] = false, flag[1] = false, turn = 0

Thread 0:

```
flag[0] := true
while  (flag[1] = true)
    if  (turn ≠ 0)
        flag[0] := false
        while  (turn ≠ 0) { }
        flag[0] := true

/* Critical Section */
```

Thread 1:

```
flag[1] := true
while  (flag[0] = true)
    if  (turn ≠ 1)
        flag[1] := false
        while  (turn ≠ 1) { }
        flag[1] := true

/* Critical Section */
```

Spec: mutual exclusion over Critical Section

Dekker's Algorithm

Sequential Consistency 

Relaxed Model x86 TSO

initial: flag[0] = false, flag[1] = false, turn = 0

Thread 0:

```
flag[0] := true
while  (flag[1] = true)
   if  (turn ≠ 0)
      flag[0] := false
      while  (turn ≠ 0) { }
      flag[0] := true

/* Critical Section */
```

Thread 1:

```
flag[1] := true
while  (flag[0] = true)
   if  (turn ≠ 1)
      flag[1] := false
      while  (turn ≠ 1) { }
      flag[1] := true

/* Critical Section */
```

Spec: mutual exclusion over Critical Section

Dekker's Algorithm

Sequential Consistency OKAY

Relaxed Model x86 TSO

initial: flag[0] = false, flag[1] = false, turn = 0

Thread 0:

```
flag[0] := true
while (flag[1] = true)
    if (turn ≠ 0)
        flag[0] := false
        while (turn ≠ 0) { }
        flag[0] := true

/* Critical Section */
```

Thread 1:

```
flag[1] := true
while (flag[0] = true)
    if (turn ≠ 1)
        flag[1] := false
        while (turn ≠ 1) { }
        flag[1] := true

/* Critical Section */
```

Spec: mutual exclusion over Critical Section

Dekker's Algorithm

Sequential Consistency OKAY

Relaxed Model x86 TSO NOT OKAY

initial: flag[0] = false, flag[1] = false, turn = 0

Thread 0:

```
flag[0] := true
while  (flag[1] = true)
   if  (turn ≠ 0)
      flag[0] := false
      while  (turn ≠ 0) { }
      flag[0] := true

/* Critical Section */
```

Thread 1:

```
flag[1] := true
while  (flag[0] = true)
   if  (turn ≠ 1)
      flag[1] := false
      while  (turn ≠ 1) { }
      flag[1] := true

/* Critical Section */
```

Spec: mutual exclusion over Critical Section

Correct Dekker's Algorithm

Relaxed Model x86 TSO

initial: flag[0] = false, flag[1] = false, turn = 0

Thread 0:

```
flag[0] := true
fence
while (flag[1] = true)
   if (turn ≠ 0)
      flag[0] := false
      while (turn ≠ 0) { }
      flag[0] := true
      fence
/* Critical Section */
```

Thread 1:

```
flag[1] := true
fence
while (flag[0] = true)
   if (turn ≠ 1)
      flag[1] := false
      while (turn ≠ 1) { }
      flag[1] := true
      fence
/* Critical Section */
```

Spec: mutual exclusion over Critical Section

Correct Dekker's Algorithm

Relaxed Model x86 TSO

initial: flag[0] = false, flag[1] = false, turn = 0

Thread 0:

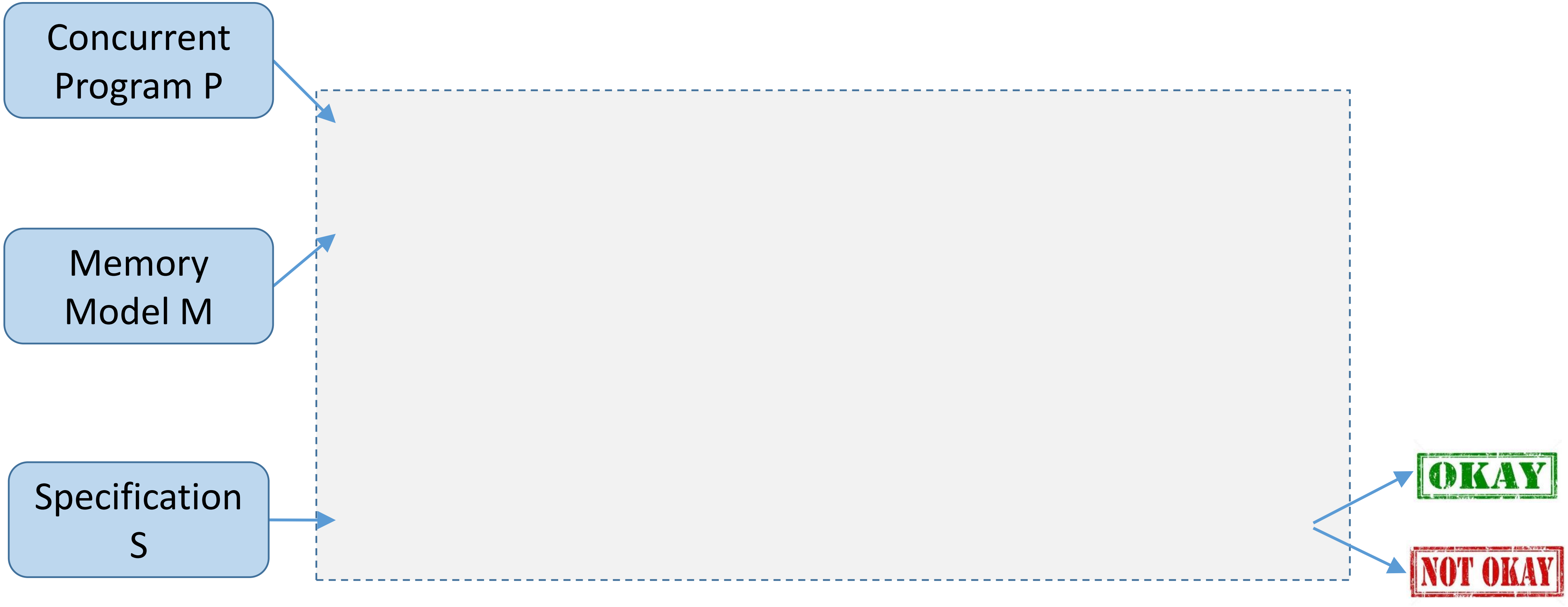
```
flag[0] := true
fence
while (flag[1] = true)
   if (turn ≠ 0)
      flag[0] := false
      while (turn ≠ 0) { }
      flag[0] := true
      fence
/* Critical Section */
```

Thread 1:

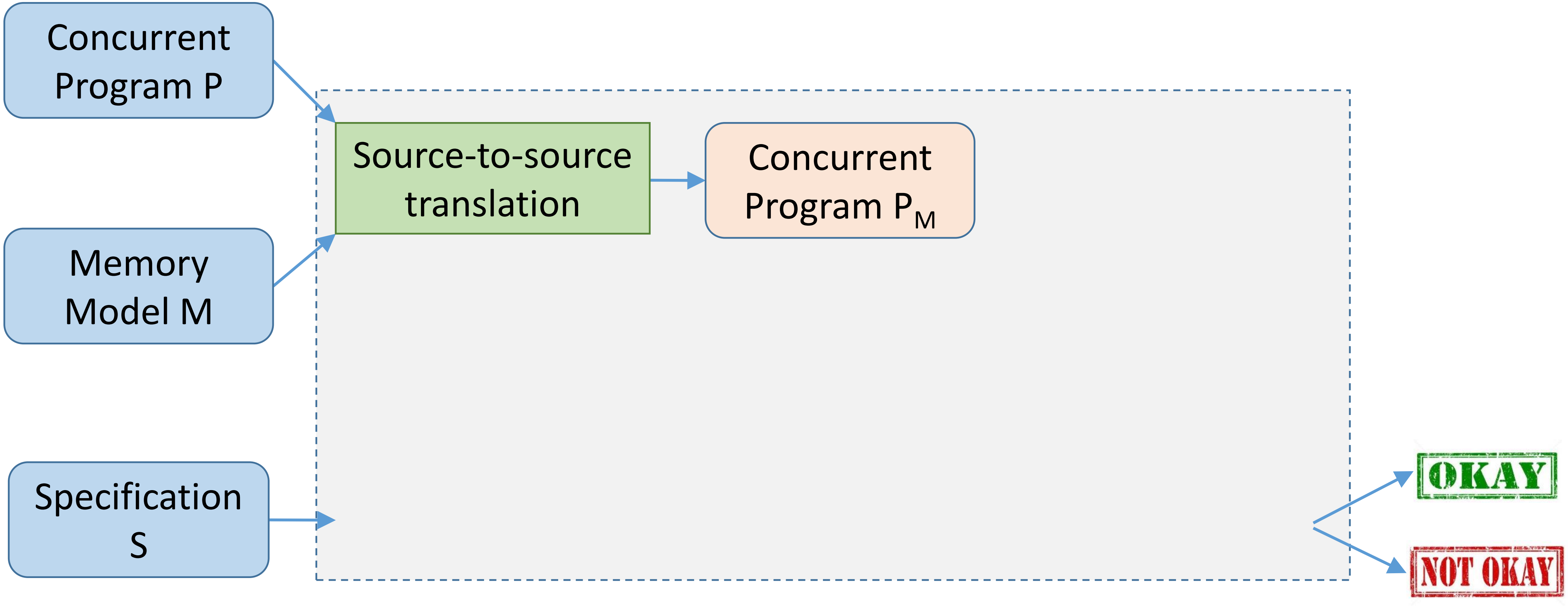
```
flag[1] := true
fence
while (flag[0] = true)
   if (turn ≠ 1)
      flag[1] := false
      while (turn ≠ 1) { }
      flag[1] := true
      fence
/* Critical Section */
```

Spec: mutual exclusion over Critical Section

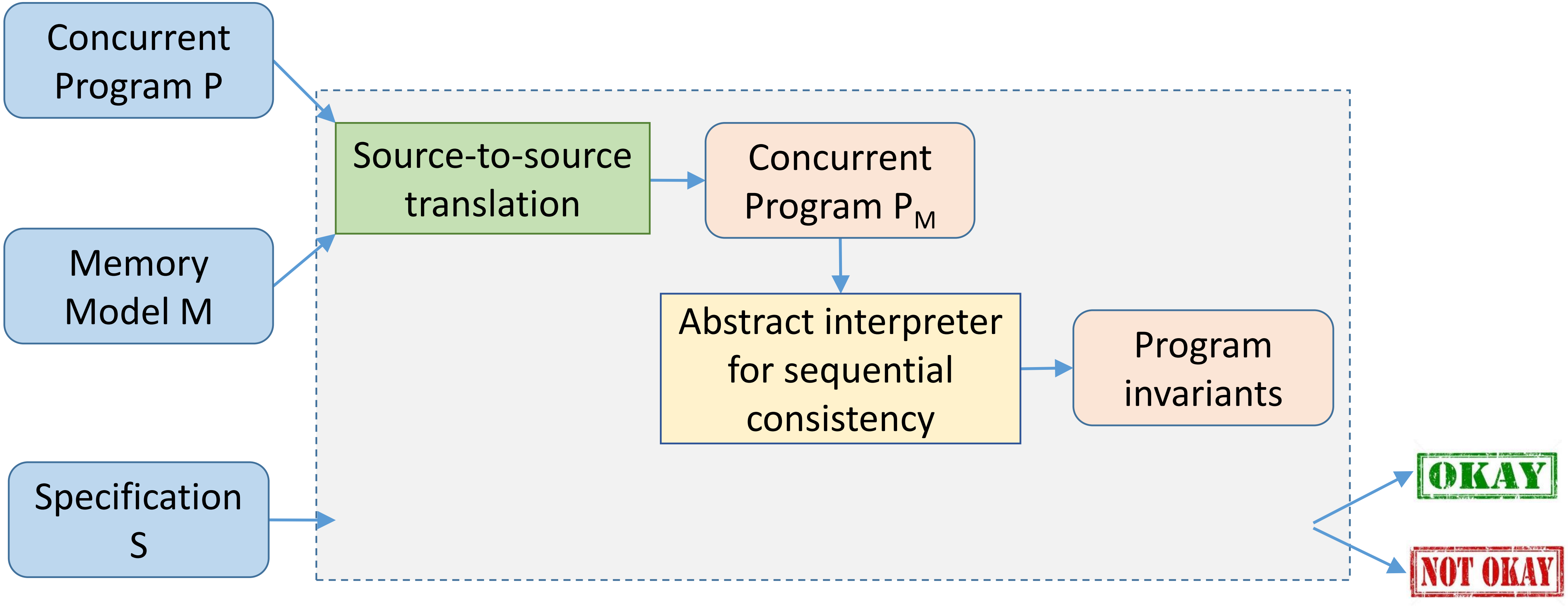
This work


Concurrent Program P
Memory Model M
Specification S
OKAY
NOT OKAY

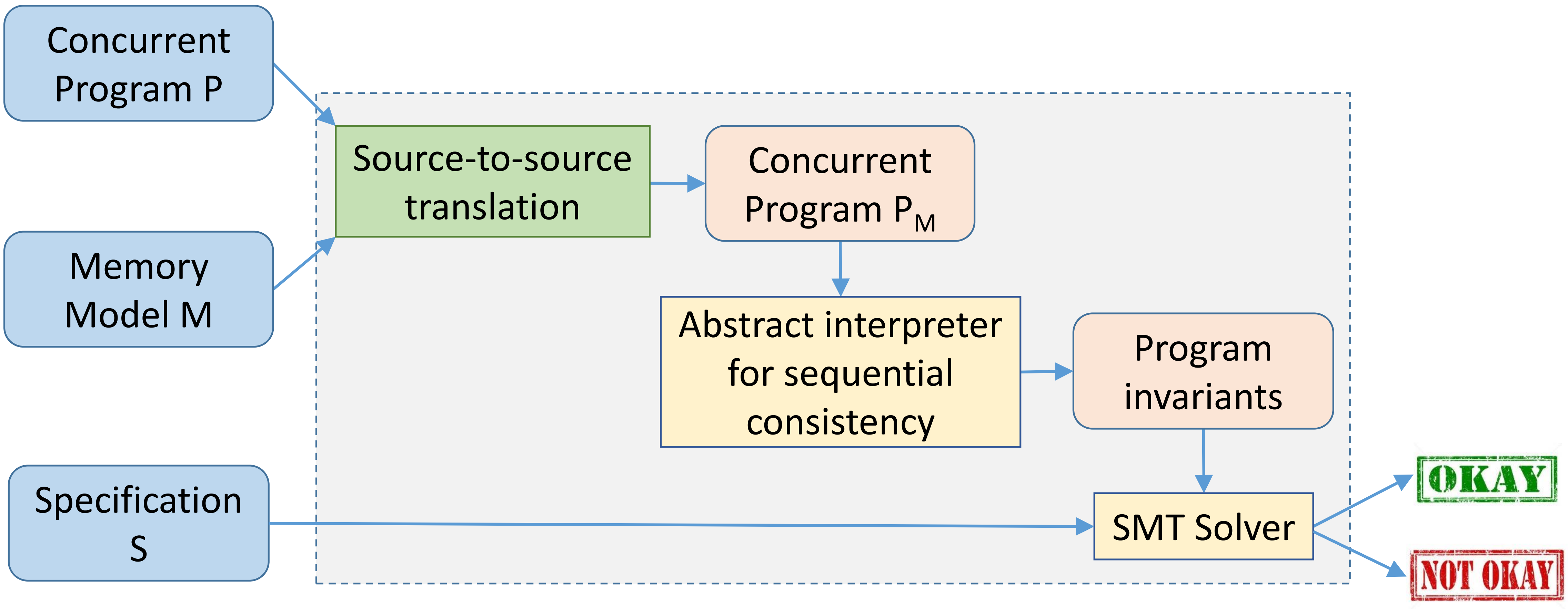
This work


Concurrent Program P
Memory Model M
Specification S
Source-to-source translation
Concurrent Program P_M
OKAY
NOT OKAY

This work

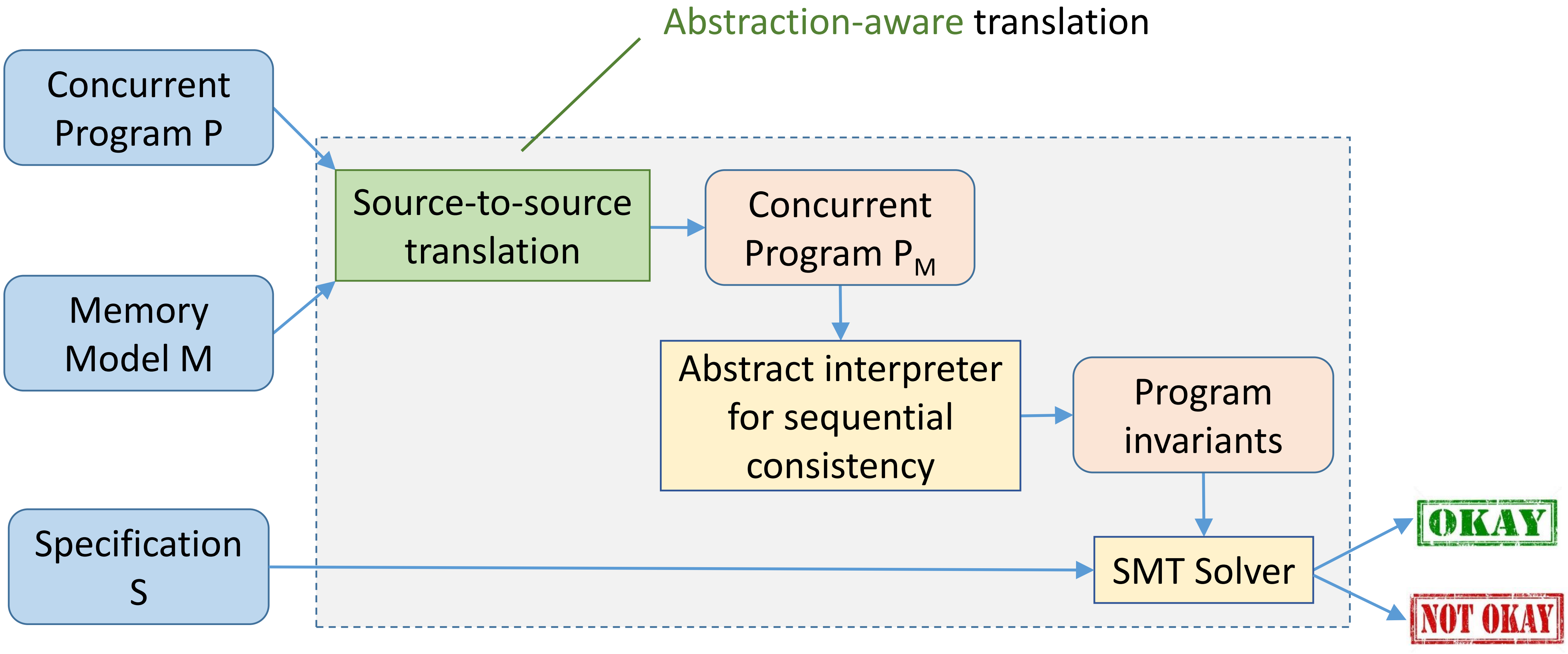

Concurrent Program P
Memory Model M
Specification S
Source-to-source translation
Concurrent Program P_M
Abstract interpreter for sequential consistency
Program invariants
OKAY
NOT OKAY

This work



Concurrent Program P
Memory Model M
Specification S
Source-to-source translation
Concurrent Program P_M
Abstract interpreter for sequential consistency
Program invariants
SMT Solver
OKAY
NOT OKAY

This work


Abstraction-aware translation
Concurrent Program P
Memory Model M
Specification S
Source-to-source translation
Concurrent Program P_M
Abstract interpreter for sequential consistency
Program invariants
SMT Solver
OKAY
NOT OKAY

Talk outline

Direct translation [SAS ‘14]

Abstraction-aware translation:

1. Leverage more refined abstract domain
2. Buffer semantics without shifting [Abstraction]

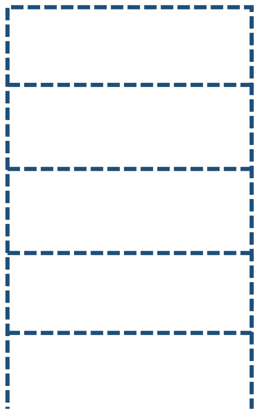
Evaluation

Direct translation for x86 TSO [SAS '14]

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
fence

Write Buffer 0:



Shared Memory:

X = 0
Y = 0

Direct translation for x86 TSO [SAS '14]

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
fence

Write Buffer 0:

X := 1

Shared Memory:

X = 0
Y = 0

Direct translation for x86 TSO [SAS '14]

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
fence

Write Buffer 0:

X := 1

Shared Memory:

X = 0
Y = 0

X := 1 —translated to→ { lhs_1 := 'X'; rhs_1 := 1; cnt := cnt + 1 }

Introduce 2 local variables in **Thread 0** to encode each location of the finite buffer.

Introduce a variable cnt. It represents the number of elements in the buffer: {0 .. k}.

Direct translation for x86 TSO [SAS '14]

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
fence

Write Buffer 0:

X := 1

Shared Memory:

X = 0
Y = 0

Direct translation for x86 TSO [SAS ‘14]

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
fence

Write Buffer 0:

X := 1

Shared Memory:

X = 0
Y = 0

Establish a limit k for the size of the buffers for each thread. For example k = 3.

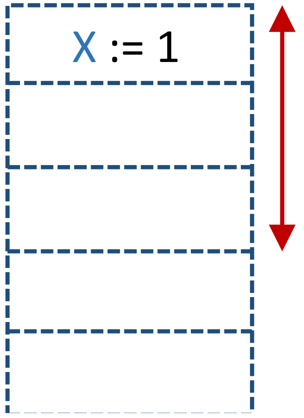
Sound abstraction.

Direct translation for x86 TSO [SAS '14]

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
fence

Write Buffer 0:



Shared Memory:

X = 0
Y = 0

Direct translation for x86 TSO [SAS ‘14]

Thread 0:

X := 1
Flush
a := X
Flush
Y := a + 1
Flush
X := a – 1
Flush
fence

Write Buffer 0:

X := 1

Shared Memory:

X = 0
Y = 0

Direct translation for x86 TSO [SAS '14]

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
fence

Write Buffer 0:

X := 1

Shared Memory:

X = 0
Y = 0

Direct translation for x86 TSO [SAS '14]

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
fence

Write Buffer 0:

X := 1

Shared Memory:

X = 0
Y = 0

Flush $\xrightarrow{\text{translated to}}$

while (cnt > 0 ∧ random) **do**
  if (lhs_1 = 'X') **then** X := rhs_1;
  if (lhs_1 = 'Y') **then** Y := rhs_1;
  cnt := cnt – 1

Direct translation for x86 TSO [SAS '14]

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
fence

Write Buffer 0:

X := 1

Shared Memory:

X = 0
Y = 0

Direct translation for x86 TSO [SAS '14]

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
fence

Write Buffer 0:

X := 1

Shared Memory:

X = 0
Y = 0

a := X $\xrightarrow{\text{translated to}}$

if (cnt ≥ 1 ∧ lhs_1 = 'X') **then** a := rhs_1;
else a := X;

Analysis with the direct translation

Original program:

Direct Translation:

Numerical abstract interpretation:

X := 1 —translated to→

lhs$_1$:= 'X'; rhs$_1$:= 1;

cnt := cnt + 1

Flush —translated to→

while (cnt > 0 ∧ random) **do**
  if (lhs$_1$ = 'X') **then** X := rhs$_1$;
  if (lhs$_1$ = 'Y') **then** Y := rhs$_1$;
  cnt := cnt – 1

a := X —translated to→

if (cnt ≥ 1 ∧ lhs$_1$ = 'X') **then** a := rhs$_1$;
else a := X;

Analysis with the direct translation

Original program:

Direct Translation:

Numerical abstract interpretation:

$X := 1$ $\xrightarrow{\text{translated to}}$

$lhs_1 :=$ ‘X’; $rhs_1 := 1$;

$cnt := cnt + 1$

$lhs_1 =$ ‘X’ $\wedge$ $rhs_1 = 1$ $\wedge$ $cnt = 1$ $\wedge$ $X = 0$

Flush $\xrightarrow{\text{translated to}}$

while ($cnt > 0 \wedge$ random) **do**
  if ($lhs_1 =$ ‘X’) **then** $X := rhs_1$;
  if ($lhs_1 =$ ‘Y’) **then** $Y := rhs_1$;
  $cnt := cnt - 1$

$a := X$ $\xrightarrow{\text{translated to}}$

if ($cnt \geq 1 \wedge lhs_1 =$ ‘X’) **then** $a := rhs_1$;
else $a := X$;

Analysis with the direct translation

Original program:

Direct Translation:

Numerical abstract interpretation:

$X := 1$ $\xrightarrow{\text{translated to}}$

$lhs_1 := \text{'X'};\quad rhs_1 := 1;$

$cnt := cnt + 1$

$lhs_1 = \text{'X'} \ \wedge\ rhs_1 = 1 \ \wedge\ cnt = 1 \ \wedge\ X = 0$

Flush $\xrightarrow{\text{translated to}}$

while $(cnt > 0 \wedge \text{random})$ **do**
  if $(lhs_1 = \text{'X'})$ **then** $X := rhs_1;$
  if $(lhs_1 = \text{'Y'})$ **then** $Y := rhs_1;$
  $cnt := cnt - 1$

$lhs_1 = \text{'X'} \ \wedge\ rhs_1 = 1 \ \wedge\ cnt = [0, 1] \ \wedge\ X = [0, 1]$

$a := X$ $\xrightarrow{\text{translated to}}$

if $(cnt \geq 1 \wedge lhs_1 = \text{'X'})$ **then** $a := rhs_1;$
else $a := X;$

Analysis with the direct translation

Original program:

$X := 1$ —translated to→

$Flush$ —translated to→

$a := X$ —translated to→

Direct Translation:

$lhs_1 := \text{'X'};\quad rhs_1 := 1;$

$cnt := cnt + 1$

while $(cnt > 0 \wedge random)$ **do**
  if $(lhs_1 = \text{'X'})$ **then** $X := rhs_1;$
  if $(lhs_1 = \text{'Y'})$ **then** $Y := rhs_1;$
  $cnt := cnt - 1$

if $(cnt \geq 1 \wedge lhs_1 = \text{'X'})$ **then** $a := rhs_1;$
else $a := X;$

Numerical abstract interpretation:

$lhs_1 = \text{'X'} \wedge rhs_1 = 1 \wedge cnt = 1 \wedge X = 0$

$lhs_1 = \text{'X'} \wedge rhs_1 = 1 \wedge cnt = [0, 1] \wedge X = [0, 1]$

$lhs_1 = \text{'X'} \wedge rhs_1 = 1 \wedge cnt = [0, 1] \wedge X = [0, 1] \wedge \mathbf{a = [0, 1]}$

Analysis with the direct translation

Original program:

$X := 1$ translated to

Flush translated to

$a := X$ translated to

Direct Translation:

$lhs_1 :=$ 'X'; $rhs_1 := 1$;

$cnt := cnt + 1$

while ($cnt > 0 \land$ random) **do**
  if ($lhs_1 =$ 'X') **then** $X := rhs_1$;
  if ($lhs_1 =$ 'Y') **then** $Y := rhs_1$;
  $cnt := cnt - 1$

if ($cnt \geq 1 \land lhs_1 =$ 'X') **then** $a := rhs_1$;
else $a := X$;

Numerical abstract interpretation:

$lhs_1 =$ 'X' $\land\ rhs_1 = 1 \ \land\ cnt = 1 \ \land\ X = 0$

$lhs_1 =$ 'X' $\land\ rhs_1 = 1 \ \land\ cnt = [0, 1] \ \land\ X = [0, 1]$

$lhs_1 =$ 'X' $\land\ rhs_1 = 1 \ \land\ cnt = [0, 1] \ \land\ X = [0, 1] \ \land$ **$a = [0, 1]$**

Problem: The analysis loses precision due to joins in the non-deterministic Flush.

Talk outline

Direct translation [SAS '14]

Looses precision with flushes,
cannot verify interesting concurrent algorithms.

Abstraction-aware translation:

1. Leverage more refined abstract domain ←
2. Buffer semantics without shifting [Abstraction]

Evaluation

More refined Abstract Domain

Logico-numerical abstract domain

- Concrete value is kept for the boolean variables
- Abstract value is kept for the numerical variables
- It allows disjunctions in the abstract states

Example:

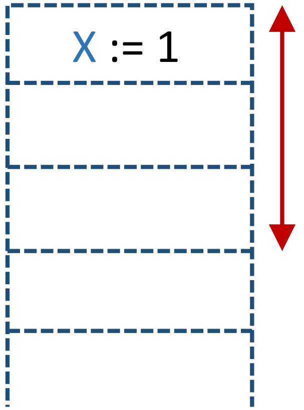
$$(b = \text{true} \land 2x + y \geq 4) \lor (b = \text{false} \land 3x - 2y \geq 7)$$

Abstraction-aware translation

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
fence

Write Buffer 0:



Shared Memory:

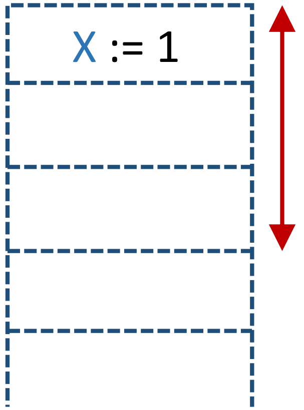
X = 0
Y = 0

Abstraction-aware translation

Thread 0:

$X := 1$
$a := X$
$Y := a + 1$
$X := a - 1$
fence

Write Buffer 0:



Shared Memory:

$X = 0$
$Y = 0$

Abstraction-aware Translation:

$X := 1$ —translated to→ $rhs_1 := 1;$ $bX_1 := true;$

Direct Translation:

$lhs_1 := \text{'X'};$ $rhs_1 := 1;$
$cnt := cnt + 1$

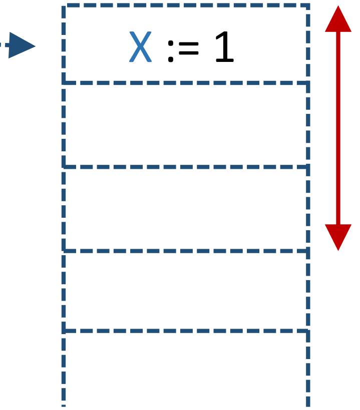
Eliminate the cnt counter variable and the lhs_1, lhs_2, lhs_3 variables.
Introduce boolean variables to replace cnt: bX_1, bX_2, bX_3, bY_1, bY_2, bY_3.

Abstraction-aware translation

Thread 0:

```
X := 1
a := X
Y := a + 1
X := a – 1
fence
```

Write Buffer 0:



Shared Memory:

X = 0
Y = 0

Abstraction-aware translation

Thread 0:

$X := 1$
$a := X$
$Y := a + 1$
$X := a - 1$
fence

Write Buffer 0:

$X := 1$

Shared Memory:

$X = 0$
$Y = 0$

Abstraction-aware Translation:

Flush $\xrightarrow{\text{translated to}}$

$$\begin{array}{l} \textbf{while}\ \ ((\ bX_1 \vee bY_1\ ) \wedge \text{random})\ \ \textbf{do} \\ \quad \textbf{if}\ \ (bX_1)\ \textbf{then}\ \ X := rhs_1;\ \ \ bX_1 := \text{false}; \\ \quad \textbf{if}\ \ (bY_1)\ \textbf{then}\ \ Y := rhs_1;\ \ \ bY_1 := \text{false}; \end{array}$$

Direct Translation:

$$\begin{array}{l} \textbf{while}\ \ (cnt > 0 \wedge \text{random})\ \ \textbf{do} \\ \quad \textbf{if}\ \ (lhs_1 = \text{'X'})\ \ \textbf{then}\ \ X := rhs_1; \\ \quad \textbf{if}\ \ (lhs_1 = \text{'Y'})\ \ \textbf{then}\ \ Y := rhs_1; \\ \quad cnt := cnt - 1 \end{array}$$

Abstraction-aware translation

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
fence

Write Buffer 0:

X := 1

Shared Memory:

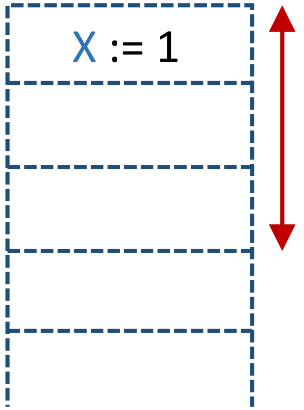
X = 0
Y = 0

Abstraction-aware translation

Thread 0:

$X := 1$
$a := X$
$Y := a + 1$
$X := a - 1$
fence

Write Buffer 0:



Shared Memory:

$X = 0$
$Y = 0$

Abstraction-aware Translation:

$a := X$ translated to

if (bX_1) **then** $a := rhs_1$;
else $a := X$;

Direct Translation:

if ($cnt \geq 1 \wedge lhs_1 = \text{'X'}$) **then** $a := rhs_1$;
else $a := X$;

Analysis with the abstraction-aware translation

Original program:

Abstraction-aware Translation:

Numerical abstract interpretation:

X := 1 —translated to→

rhs_1 := 1;

bX_1 := true;

Flush —translated to→

while ((bX_1 ∨ bY_1) ∧ random) **do**
if (bX_1) **then** X := rhs_1; bX_1 := false;
if (bY_1) **then** Y := rhs_1; bY_1 := false;

a := X —translated to→

if (bX_1) **then** a := rhs_1;
else a := X;

Analysis with the abstraction-aware translation

Original program:

Abstraction-aware Translation:

Numerical abstract interpretation:

$X := 1$ translated to

$rhs_1 := 1;$

$bX_1 := \text{true};$

$bX_1 = \text{true} \land rhs_1 = 1 \land X = 0$

Flush translated to

while $((bX_1 \lor bY_1) \land \text{random})$ **do**

  if (bX_1) **then** $X := rhs_1;\ \ bX_1 := \text{false};$

  if (bY_1) **then** $Y := rhs_1;\ \ bY_1 := \text{false};$

$a := X$ translated to

if (bX_1) **then** $a := rhs_1;$

else $a := X;$

Analysis with the abstraction-aware translation

Original program:

Abstraction-aware Translation:

Numerical abstract interpretation:

$X := 1$ —translated to→

$rhs_1 := 1;$

$bX_1 := true;$

$bX_1 = true \wedge rhs_1 = 1 \wedge X = 0$

Flush —translated to→

while (($bX_1 \vee bY_1$) $\wedge$ random) **do**
  if (bX_1) **then** $X := rhs_1;$ $bX_1 := false;$
  if (bY_1) **then** $Y := rhs_1;$ $bY_1 := false;$

$(bX_1 = true \wedge rhs_1 = 1 \wedge X = 0) \vee$
$(bX_1 = false \wedge rhs_1 = 1 \wedge X = 1)$

$a := X$ —translated to→

if (bX_1) **then** $a := rhs_1;$
else $a := X;$

Analysis with the abstraction-aware translation

Original program:

Abstraction-aware Translation:

Numerical abstract interpretation:

$X := 1$ — translated to →

$rhs_1 := 1;$

$bX_1 := true;$

$bX_1 = true \wedge rhs_1 = 1 \wedge X = 0$

Flush — translated to →

while (($bX_1 \vee bY_1$) $\wedge$ random) **do**
  if (bX_1) **then** $X := rhs_1$; $bX_1 :=$ false;
  if (bY_1) **then** $Y := rhs_1$; $bY_1 :=$ false;

$(bX_1 = true \wedge rhs_1 = 1 \wedge X = 0) \vee$
$(bX_1 = false \wedge rhs_1 = 1 \wedge X = 1)$

$a := X$ — translated to →

if (bX_1) **then** $a := rhs_1$;
else $a := X$;

$(bX_1 = true \wedge rhs_1 = 1 \wedge X = 0 \wedge a = 1) \vee$
$(bX_1 = false \wedge rhs_1 = 1 \wedge X = 1 \wedge a = 1)$

Analysis with the abstraction-aware translation

Original program:

Abstraction-aware Translation:

Numerical abstract interpretation:

$X := 1$ translated to

$rhs_1 := 1;$

$bX_1 := true;$

$bX_1 = true \wedge rhs_1 = 1 \wedge X = 0$

Flush translated to

while $((bX_1 \vee bY_1) \wedge random)$ **do**
if (bX_1) **then** $X := rhs_1;\ bX_1 := false;$
if (bY_1) **then** $Y := rhs_1;\ bY_1 := false;$

$(bX_1 = true \wedge rhs_1 = 1 \wedge X = 0) \vee$
$(bX_1 = false \wedge rhs_1 = 1 \wedge X = 1)$

$a := X$ translated to

if (bX_1) **then** $a := rhs_1;$
else $a := X;$

$(bX_1 = true \wedge rhs_1 = 1 \wedge X = 0 \wedge a = 1) \vee$
$(bX_1 = false \wedge rhs_1 = 1 \wedge X = 1 \wedge a = 1)$

Invariant from Direct Translation:

$\ldots \wedge rhs_1 = 1 \wedge X = [0, 1] \wedge a = [0, 1]$

Analysis with the abstraction-aware translation

Original program:

Abstraction-aware Translation:

Numerical abstract interpretation:

$X := 1$ translated to

$rhs_1 := 1;$

$bX_1 := true;$

$bX_1 = true \land rhs_1 = 1 \land X = 0$

Flush translated to

while (($bX_1 \lor bY_1$) $\land$ random) **do**

  if (bX_1) **then** $X := rhs_1$; $bX_1 := false$;

  if (bY_1) **then** $Y := rhs_1$; $bY_1 := false$;

$(bX_1 = true \land rhs_1 = 1 \land X = 0) \lor$
$(bX_1 = false \land rhs_1 = 1 \land X = 1)$

$a := X$ translated to

if (bX_1) **then** $a := rhs_1$;
else $a := X$;

$(bX_1 = true \land rhs_1 = 1 \land X = 0 \land a = 1) \lor$
$(bX_1 = false \land rhs_1 = 1 \land X = 1 \land a = 1)$

Invariant from Direct Translation:

$\ldots \land rhs_1 = 1 \land X = [0, 1] \land a = [0, 1]$

Flush with shifting

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
fence

Write Buffer 0:

X := 1
Y := 2
X := 0

Shared Memory:

X = 0
Y = 0

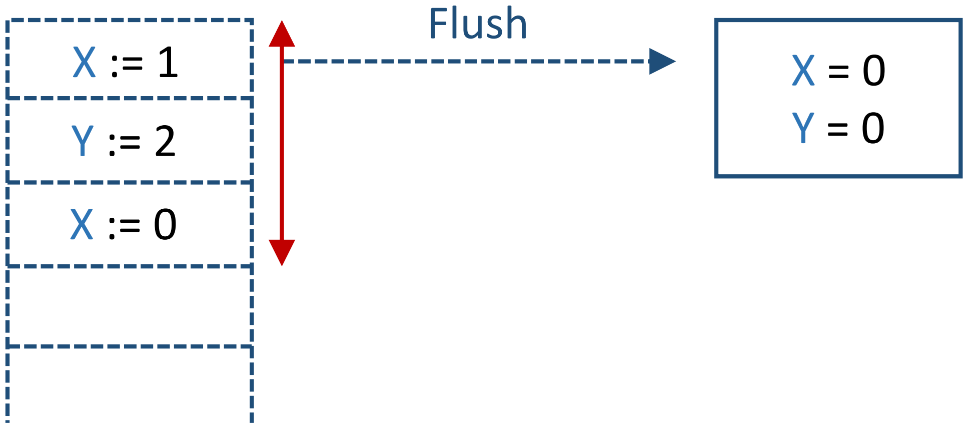
Flush with shifting

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
fence

Write Buffer 0:

Shared Memory:

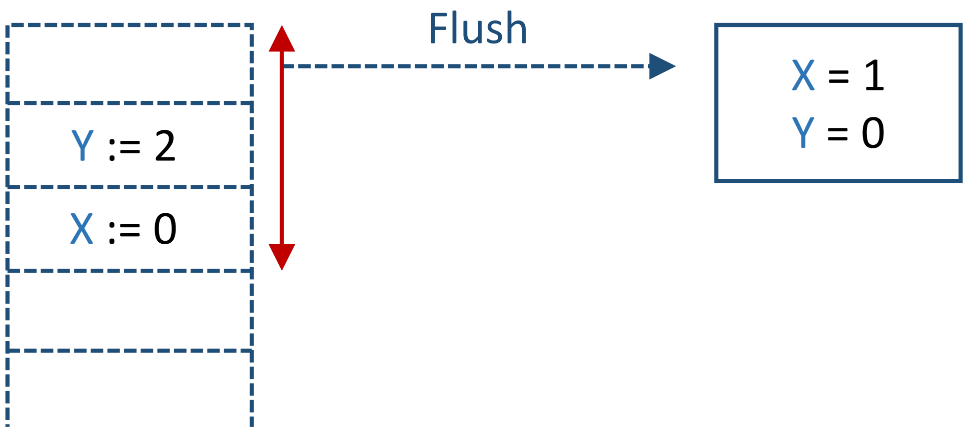


Flush with shifting

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
→ **fence**

Write Buffer 0: **Shared Memory:**

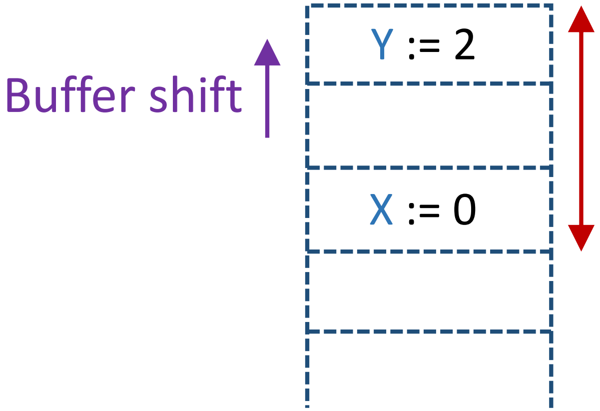


Flush with shifting

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
fence

Write Buffer 0:



Shared Memory:

X = 1
Y = 0

Flush with shifting

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
fence

Write Buffer 0:

Buffer shift

Y := 2
X := 0

Shared Memory:

X = 1
Y = 0

Flush with shifting

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
fence

Write Buffer 0:

Y := 2
X := 0

Buffer shift

Shared Memory:

X = 1
Y = 0

Flush $\xrightarrow{\text{translated to}}$

while $((bX_1 \lor bY_1) \land \text{random})$ **do**
  if (bX_1) **then** $X := rhs_1$; $bX_1 := \text{false}$;
  if (bY_1) **then** $Y := rhs_1$; $bY_1 := \text{false}$;
  if (bX_2) **then** $rhs_1 := rhs_2$; $bX_1 := \text{true}$; $bX_2 := \text{false}$;
  if (bY_2) **then** $rhs_1 := rhs_2$; $bY_1 := \text{true}$; $bY_2 := \text{false}$;
  if (bX_3) **then** $rhs_2 := rhs_3$; $bX_2 := \text{true}$; $bX_3 := \text{false}$;
  if (bY_3) **then** $rhs_2 := rhs_3$; $bY_2 := \text{true}$; $bY_3 := \text{false}$;

Flush with shifting

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
fence

Write Buffer 0:

Buffer shift

Y := 2
X := 0

Shared Memory:

X = 1
Y = 0

Flush $\xrightarrow{\text{translated to}}$

while $((bX_1 \vee bY_1) \wedge \text{random})$ **do**
  if (bX_1) **then** $X := rhs_1;\ \ bX_1 := \text{false};$
  if (bY_1) **then** $Y := rhs_1;\ \ bY_1 := \text{false};$
  if (bX_2) **then** $rhs_1 := rhs_2;\ \ bX_1 := \text{true};\ \ bX_2 := \text{false};$
  if (bY_2) **then** $rhs_1 := rhs_2;\ \ bY_1 := \text{true};\ \ bY_2 := \text{false};$
  if (bX_3) **then** $rhs_2 := rhs_3;\ \ bX_2 := \text{true};\ \ bX_3 := \text{false};$
  if (bY_3) **then** $rhs_2 := rhs_3;\ \ bY_2 := \text{true};\ \ bY_3 := \text{false};$

Buffer Shift

Flush procedure

Appears after each translated statement.

Its complexity is due mostly to the buffer shifting operation

Problem: This can lead to more work for the analysis and loss of precision.

Talk outline

Direct translation [SAS ‘14]

Abstraction-aware translation:

1. Leverage more refined abstract domain
2. Buffer semantics without shifting [Abstraction]

Evaluation

Flushing without shifting

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
→ **fence**

Write Buffer 0:

X := 1
Y := 2
X := 0

Shared Memory:

X = 0
Y = 0

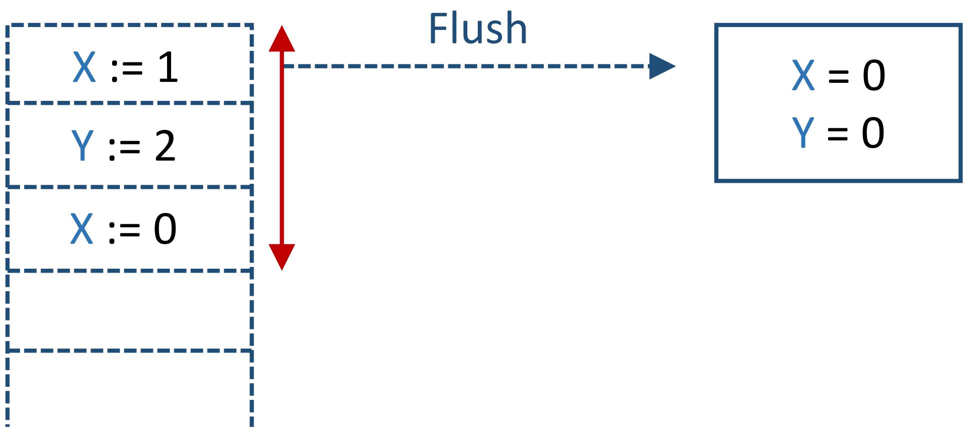
Flushing without shifting

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
→ **fence**

Write Buffer 0:

Shared Memory:



Flushing without shifting

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
fence

Write Buffer 0:

Y := 2
X := 0

Shared Memory:

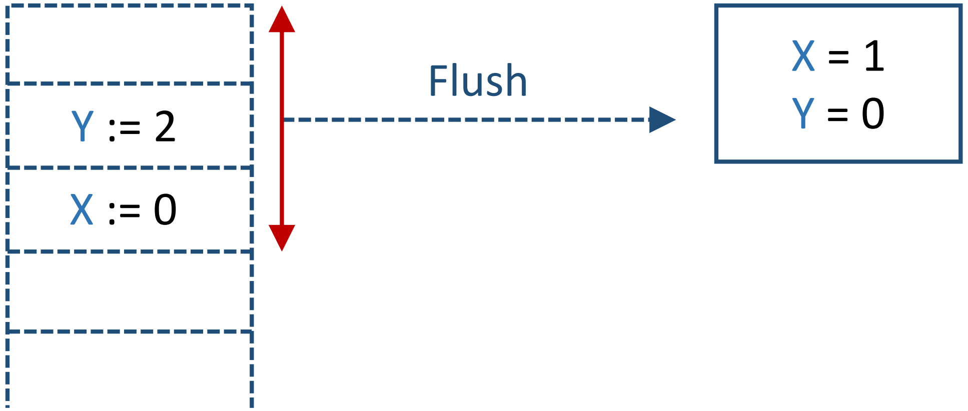
X = 1
Y = 0

Flushing without shifting

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
→ **fence**

Write Buffer 0: **Shared Memory:**

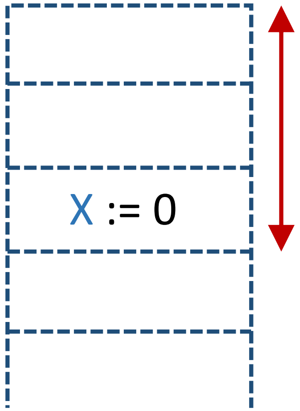


Flushing without shifting

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
→ **fence**

Write Buffer 0:



Shared Memory:

X = 1
Y = 2

Flushing without shifting

Thread 0:

X := 1
a := X
Y := a + 1
X := a – 1
→ **fence**

Write Buffer 0:

X := 0

Shared Memory:

X = 1
Y = 2

Flush —translated to→

while (random) **do**
  if (bX_1) **then** X := rhs_1; bX_1 := false;
  else if (bY_1) **then** Y := rhs_1; bY_1 := false;
  else if (bX_2) **then** X := rhs_2; bX_2 := false;
  else if (bY_2) **then** Y := rhs_2; bY_2 := false;
  else if (bX_3) **then** X := rhs_3; bX_3 := false;
  else if (bY_3) **then** Y := rhs_3; bY_3 := false;

Flushing without shifting

Eliminating buffer shifting:

- is sound

- is an abstraction

    - may introduce additional cases of imprecision,
      not the case for any of our benchmarks

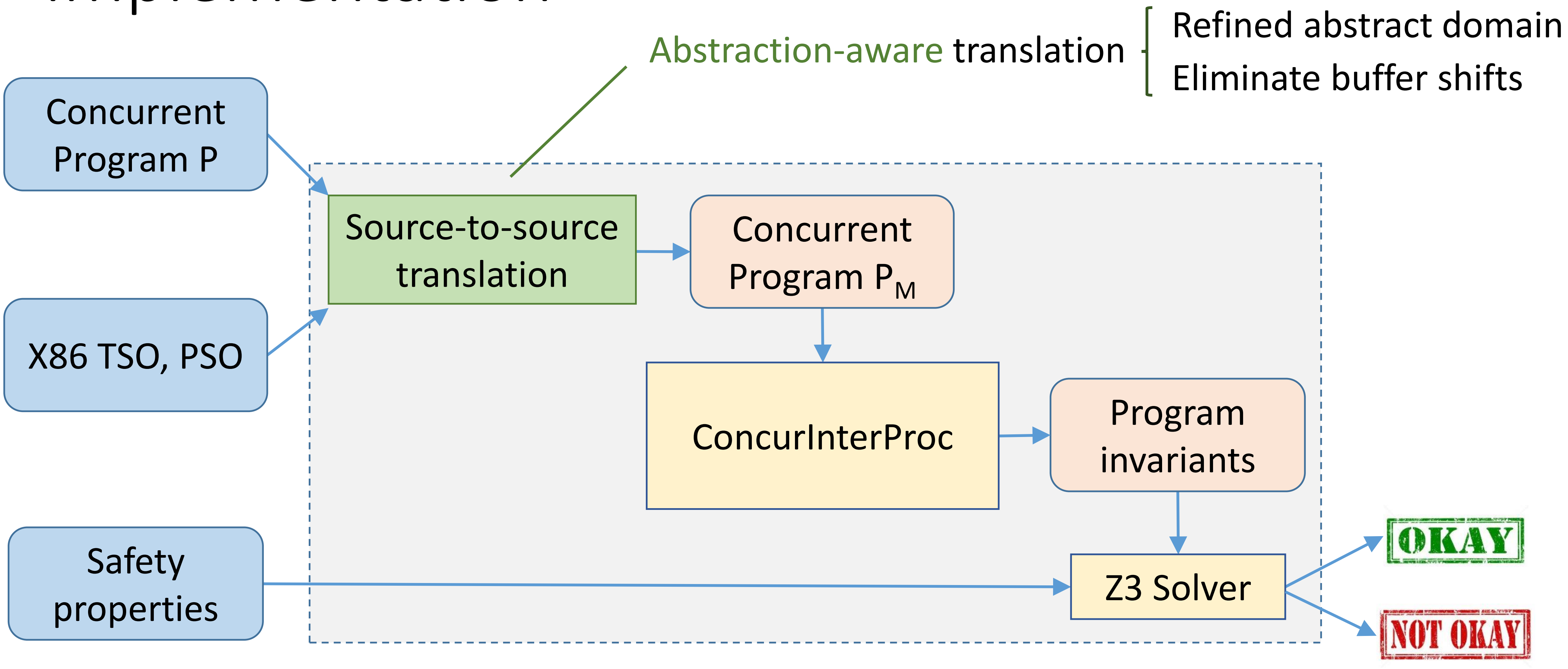
Comparing Translations

Original program:		Abstraction-aware translation:	Direct translation [SAS '14]:
$X := 1$	translated to →	$rhs_1 := 1;$ $bX_1 := \text{true};$	$lhs_1 := \text{'X'};\quad rhs_1 := 1;$ $cnt := cnt + 1$

Flush	translated to →	**while** $((bX_1 \lor bY_1) \land \text{random})$ **do**
		  if (bX_1) **then** $X := rhs_1;\ bX_1 := \text{false};$
		  else if (bY_1) **then** $Y := rhs_1;\ bY_1 := \text{false};$
		  else if (bX_2) **then** $X := rhs_2;\ bX_2 := \text{false};$
		  else if (bY_2) **then** $Y := rhs_2;\ bY_2 := \text{false};$
			while $(cnt > 0 \land \text{random})$ **do**
			  if $(lhs_1 = \text{'X'})$ **then** $X := rhs_1;$
			  if $(lhs_1 = \text{'Y'})$ **then** $Y := rhs_1;$
			  if $(cnt > 1)$ **then**
			    $lhs_1 := lhs_2;\ rhs_1 := rhs_2;$
			  $cnt := cnt - 1$

$a := X$	translated to →	**if** (bX_1) **then** $a := rhs_1;$ **else** $a := X;$	**if** $(cnt \geq 1 \land lhs_1 = \text{'X'})$ **then** $a := rhs_1;$ **else** $a := X;$

Implementation



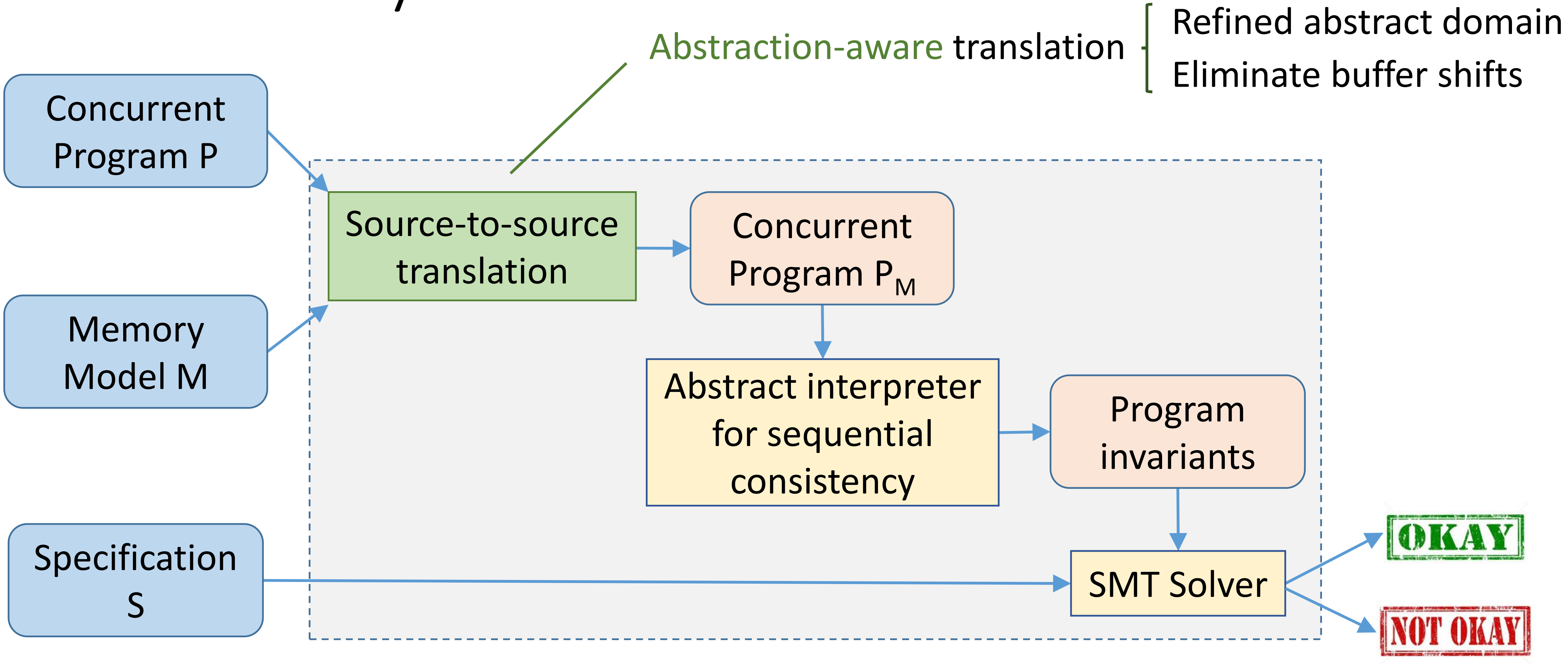
Evaluation for x86 TSO

	Abstraction-aware Translation			Direct Translation [SAS '14]		
Program	# Fences	Time (sec)	Memory (MB)	# Fences	Time (sec)	Memory (MB)
Abp	0	5	189	0	14	352
Bakery	4	1148	4749	8	3181	6575
Concloop	2	8	547	2	18	891
Dekker	4	227	2233	10	615	1004
Kessel	4	14	357	4	15	424
Queue	1	1	101	1	1	115
Szymanski	3	1066	3781	8	124	1770
WSQ THE	4	125	1646	6	t/o	-
WSQ Chase-Lev	2	17	550	4	30	789

Evaluation for x86 TSO

	Abstraction-aware Translation			Direct Translation [SAS '14]		
Program	# Fences	Time (sec)	Memory (MB)	# Fences	Time (sec)	Memory (MB)
Abp	0	5	189	0	14	352
Bakery	4	1148	4749	8	3181	6575
Concloop	2	8	547	2	18	891
Dekker	4	227	2233	10	615	1004
Kessel	4	14	357	4	15	424
Queue	1	1	101	1	1	115
Szymanski	3	1066	3781	8	124	1770
WSQ THE	4	125	1646	6	t/o	-
WSQ Chase-Lev	2	17	550	4	30	789

In summary



Additional details: www.practicalsynthesis.org/fender