Synthesis for Concurrency

Martin Vechev
ETH Zurich

(joint work with Eran Yahav, Greta Yorsh)
Concurrency is Everywhere
Unreliable Concurrency is Everywhere
2003 Northeast Blackout

concurrency error triggers massive power outage

economic effect : ~ 6 billion $USD
Unreliable Concurrency: Data Structures

val popRight() {
    while (true) {
        rh = RightHat;
        lh = LeftHat;
        if (rh->R == rh) return "empty";
        if (rh == lh) {
            if (DCAS(&RightHat, &LeftHat, rh, lh, Dummy, Dummy))
                return rh->V;
        } else {
            rhL = rh->L;
            if (DCAS(&RightHat, &rh->L, rh, rhL, rhL, rh)) {
                result = rh->V;
                rh->R = Dummy;
                return result;
            }
        }
    }
}

“Even better DCAS-based concurrent deques”, DISC 2000
Unreliable Concurrency: Data Structures

val popRight() {

Bad News:

2 errors in < 20 lines of code

“Even better DCAS-based concurrent deques”, DISC 2000
Unreliable Concurrency: Weak Memory Models

```c
int take()
    long b = bot - 1;
    item_t *q = wsq;
    bot = b
    long t = top
if (b < t) {
    bot = t;
    return EMPTY;
}
task = q->ap[b%q->sz];
if (b > t)
    return task
if (!CAS(&top,t,t+1))
    return EMPTY;
bot = t + 1;
return task;
```

```c
void push(int task)
    long b = bot;
    long t = top;
    item_t *q = wsq;
    if (b-t ≥ q->sz-1){
        wsq = expand();
        q = wsq;
    }
    q->ap[b%q->sz]=task;
    bot = b+1;
```

```c
int steal()
    long t = top;
    long b = bot;
    item_t *q = wsq;
    if (t < b)
        return EMPTY;
    task=q->ap[t%q->sz];
    if (!CAS(&top,t,t+1))
        return ABORT;
    return task;
```
Unreliable Concurrency: Weak Memory Models

```c
int take() {
    long b = bot-1;
    item_t *q = wsq;
    bot = b
    long t = top
    if (b < t) {
        bot = t;
        return EMPTY;
    }
    task = q->ap[b%q->sz];
    if (b > t)
        return task
    if (!CAS(&top,t,t+1))
        return EMPTY;
    bot = t + 1;
    return task;
}

void push(int task) {
    long b = bot;
    long t = top;
    item_t *q = wsq;
    if (b-t >= q->sz-1) {
        wsq = expand();
        q = wsq;
    }
    q->ap[b%q->sz]=task;
    bot = b+1;
}

int steal() {
    long t = top;
    long b = bot;
    item_t *q = wsq;
    if (t < b) {
        task=q->ap[t%q->sz];
        if (!CAS(&top,t,t+1))
            return ABORT;
        return task;
    }
    bot = bot-1;
    return task;
}
```
bool add(int key) {
    Entry *pred,*curr,*entry
restart:
    locate(pred,curr,key)
    k = (curr->key == key)
if (k) return false
    entry = new Entry()
    entry->next = curr
    val=CAS(&pred->next, entry)
if (val) goto restart
    return true
}

bool remove(int key) {
    Entry *pred,*curr,*r
restart:
    locate(pred,curr,key)
    k = (curr->key ≠ key)
if (k) return false
    <r,m> = curr->next
    lval=CAS(&curr->next, curr, curr, entry)
if (lval) goto restart
    pval=CAS(&pred->next, curr, curr)
if (pval) goto restart
    return true
}

bool contains(int key) {
    Entry *pred,*curr
    locate(pred,curr,key)
    k = (curr->key == key)
if (k) return false
    return true
}
bool add(int key) {
    Entry *pred,*curr,*entry
    restart:
    locate(pred,curr,key)
    k = (curr->key == key)
    if (k) return false
    entry = new Entry()
    entry->next = curr
    val=CAS(&pred->next,
            <curr,0>,<entry,0>)
    if (!val) goto restart
    return true
}

bool remove(int key) {
    Entry *pred,*curr,*r
    restart:
    locate(pred,curr,key)
    k = (curr->key != key)
    if (k) return false
    r,m> = curr->next
    lval=CAS(&curr->next,
             <r,m>,<r,1>)
    if (!lval) goto restart
    pval=CAS(&pred->next,
             <curr,0>,<r,0>)
    if (!pval) goto restart
    return true
}

bool contains(int key) {
    Entry *pred,*curr
    locate(pred,curr,key)
    k = (curr->key == key)
    if (k) return false
    return true
}
bool add(int key) {
    Entry *pred, *curr, *entry
    restart:
    locate(pred, curr, key)
    k = (curr->key == key)
    if (k) return false
    entry = new Entry()
    entry->next = curr
    val = CAS(&pred->next, curr, 0, entry, 0)
    if (!val) goto restart
    return true
}

bool remove(int key) {
    Entry *pred, *curr, *r
    restart:
    locate(pred, curr, key)
    k = (curr->key != key)
    if (k) return false
    m = curr->next
    lval = CAS(&curr->next, m, 1)
    if (!lval) goto restart
    pval = CAS(&pred->next, curr, 0)
    if (!pval) goto restart
    return true
}

bool contains(int key) {
    Entry *pred, *curr
    locate(pred, curr, key)
    k = (curr->key == key)
    if (k) return false
    return true
}
bool add(int key) {
    Entry *pred, *curr, *entry
    locate(pred, curr, key)
    k = (curr->key == key)
    if (k) return false
    entry = new Entry()
    entry->next = curr
    val = CAS(&pred->next, entry, 0)
    if (!val) goto restart
    return true
}

bool remove(int key) {
    Entry *pred, *curr, *r
    restart:
    locate(pred, curr, key)
    k = (curr->key != key)
    if (k) return false
    r = curr->next
    lval = CAS(&curr->next, r, 1)
    if (!lval) goto restart
    pval = CAS(&pred->next, r, 0)
    if (!pval) goto restart
    return true
}

bool contains(int key) {
    Entry *pred, *curr
    locate(pred, curr, key)
    k = (curr->key == key)
    if (k) return false
    return true
}

Concurrent Set Algorithm

The Good
• New algorithm
• Fine-grained synchronization (CAS)

The Bad
• Can you understand what it does?
• Can you show it is correct?

The Ugly
• How did you get it?
• Anything repeatable?
• Any other similar algorithms?
bool add(int key) {
    atomic
    Entry *pred, *curr, *entry
    locate(pred, curr, key);
    k = (curr->key == key)
    if (k) return false
    entry = new Entry()
    entry->next = curr
    pred->next = entry
    return true
}

bool remove(int key) {
    atomic
    Entry *pred, *curr, *r
    locate(pred, curr, key)
    k = (curr->key \neq key)
    if (k) return false
    r = curr->next
    pred->next = r
    return true
}

bool contains(int key) {
    atomic
    Entry *pred, *curr
    locate(pred, curr, key)
    k = (curr->key == key)
    if (k) return false
    return true
}
Sequential Set Algorithm

```c
bool add(int key) {
    atomic
    Entry *pred, *curr, *entry
    locate(pred, curr, key);
    k = (curr->key == key)
    if (k) return false
    entry = new Entry()
    entry->next = curr
    pred->next = entry
    return true
}
```

```c
bool remove(int key) {
    atomic
    Entry *pred, *curr, *r
    locate(pred, curr, key)
    k = (curr->key != key)
    if (k) return false
    r = curr->next
    pred->next = r
    return true
}
```

```c
bool contains(int key) {
    atomic
    Entry *pred, *curr
    locate(pred, curr, key)
    k = (curr->key == key)
    if (k) return false
    return true
}
```

- Understandable
- Proving correctness easier
bool add(int key) {
    atomic
    Entry *pred,*curr,*entry
    locate(pred,curr,key);
    k = (curr->key == key)
    if (k) return false
    entry = new Entry()
    entry->next = curr
    pred->next = entry
    return true
}

bool remove(int key) {
    atomic
    Entry *pred,*curr,*r
    locate(pred,curr,key)
    k = (curr->key ≠ key)
    if (k) return false
    r = curr->next
    pred->next = r
    return true
}
Sequential to Highly Concurrent

```c
bool add(int key) {
    atomic
    Entry *pred,*curr,*entry
    locate(pred,curr,key);
    k = (curr->key == key)
    if (k) return false
    entry = new Entry()
    entry->next = curr
    pred->next = entry
    return true
}
```

```c
bool remove(int key) {
    atomic
    Entry *pred,*curr,*r
    locate(pred,curr,key)
    k = (curr->key ≠ key)
    if (k) return false
    r = curr->next
    pred->next = r
    return true
}
```

```c
bool add(int key) {
    Entry *pred,*curr,*entry
    restart:
    locate(pred,curr,key)
    k = (curr->key == key)
    if (k) return false
    entry = new Entry()
    entry->next = curr
    pred->next = entry
    val=CAS(&pred->next,
             { curr,0 },
             { entry,0 })
    if (!val) goto restart
    return true
}
```

```c
bool remove(int key) {
    Entry *pred,*curr,*r
    restart:
    locate(pred,curr,key)
    k = (curr->key ≠ key)
    if (k) return false
    r = curr->next
    pred->next = r
    lval=CAS(&curr->next,
             { r,0 },
             { r,1 })
    if (!lval) goto restart
    pval=CAS(&pred->next,
             { curr,0 },
             { r,0 })
    if (!pval) goto restart
    return true
}
```
bool add(int key) {
    atomic
    Entry *pred,*curr,*entry
    locate(pred,curr,key);
    k = (curr->key == key)
    if (k) return false
    entry = new Entry()
    entry->next = curr
    pred->next = entry
    return true
}

bool remove(int key) {
    atomic
    Entry *pred,*curr,*r
    locate(pred,curr,key)
    k = (curr->key != key)
    if (k) return false
    r = curr->next
    pred->next = r
    return true
}
Bridging the Gap

- Find **small** repeatable transformations
  - **NO** magic insights

- Combination of manual and automatic transformations
## Transformations

<table>
<thead>
<tr>
<th>Transformation</th>
<th>Diagram</th>
</tr>
</thead>
<tbody>
<tr>
<td>Removing redundant atomicity</td>
<td><img src="transformation1.png" alt="Diagram" /></td>
</tr>
<tr>
<td>Reordering statements</td>
<td><img src="transformation2.png" alt="Diagram" /></td>
</tr>
<tr>
<td>Optimistic concurrency</td>
<td><img src="transformation3.png" alt="Diagram" /></td>
</tr>
<tr>
<td>Add synchronization meta-data</td>
<td><img src="transformation4.png" alt="Diagram" /></td>
</tr>
</tbody>
</table>
## Transformations

<table>
<thead>
<tr>
<th>Machine</th>
<th>Machine + Human</th>
<th>Only Human</th>
</tr>
</thead>
<tbody>
<tr>
<td>Removing redundant atomicity</td>
<td>Reordering statements</td>
<td>Optimistic concurrency</td>
</tr>
</tbody>
</table>
| \[
\begin{array}{c}
S_1 \\
S_2 \\
\end{array}
\parallel
\begin{array}{c}
S_3 \\
S_4 \\
\end{array}
\rightarrow
\begin{array}{c}
S_1 \\
S_2 \\
\end{array}
\parallel
\begin{array}{c}
S_3 \\
S_4 \\
\end{array}
\] | \[
\begin{array}{c}
S_1 \\
S_2 \\
\end{array}
\parallel
\begin{array}{c}
S_3 \\
S_4 \\
\end{array}
\rightarrow
\begin{array}{c}
S_2 \\
S_1 \\
\end{array}
\parallel
\begin{array}{c}
S_3 \\
S_4 \\
\end{array}
\] | \[
\begin{array}{c}
\text{read} \\
\text{update} \\
\end{array}
\rightarrow
\begin{array}{c}
\text{read} \\
\text{If (validate)} \\
\text{update} \\
\text{else} \\
\text{restart} \\
\end{array}
\] | \[
\begin{array}{c}
S_1 \\
S_2 \\
\end{array}
\parallel
\begin{array}{c}
S_3 \\
S_4 \\
\end{array}
\rightarrow
\begin{array}{c}
S_1 \\
\text{If (t > 0)} \\
S_2 \\
\end{array}
\parallel
\begin{array}{c}
S_3 \\
S_4 \\
\end{array}
\] |

Machine + Human: 

- \[
\begin{array}{c}
S_1 \\
S_2 \\
\end{array}
\parallel
\begin{array}{c}
S_3 \\
S_4 \\
\end{array}
\rightarrow
\begin{array}{c}
S_1 \\
S_2 \\
\end{array}
\parallel
\begin{array}{c}
S_3 \\
S_4 \\
\end{array}
\]
Synthesis Process

Machine

Verifier

Synthesizer

Program

Yes/counterexample

• Exploration
• Validation

Schema

human

checked algorithms

algorithms

synthesis process
Program Synthesis

Verifier -> Synthesizer

Program -> Machine

Yes/Counterexample
Program Synthesis

Machine

verifier → synthesizer

program

yes/counterexample

analyze program
Program Synthesis

Machine

verifier

program

synthesizer

yes/counterexample

analyze program
Program Synthesis

Machine

verifier

program

synthesizer

yes/counterexample

analyze program
Program Synthesis

Machine

verifier → program

yes/counterexample → synthesizer

analyze program
Program Synthesis

Machine

verifier → program → synthesizer

yes/counterexample

propagate correctness
Program Synthesis

Machine

verifier → program → synthesizer

yes/counterexample

analyze program
Program Synthesis

verifier \rightarrow \text{synthesizer}

program \rightarrow \text{yes/counterexample}

Machine

propagate incorrectness
Sequential To Concurrent

Sequential

DCAS
DCAS
Michael (PODC’02)
Heller et al. (OPODIS’05)
CAS
CAS with LOCKS
CAS/DCAS
Priority Queue
Stack
Trieber Stack

Schema
Correct Algorithm
Existing Algorithm
New Algorithm
bool remove(int key) {
    Entry *pred, *curr, *r

    atomic
    locate(pred, curr, key)
    k = (curr->key != key)
    if (k) return false
    r = curr->next
    pred->next = r
    return true
}

bool remove(int key) {
    Entry *pred, *curr, *r

    restart:
    atomic
    Read
    if (validate) {

        Update
        }

    goto restart
}
bool remove(int key) {
    Entry *pred, *curr, *r
    atomic
        locate(pred, curr, key)
        k = (curr->key ≠ key)
        if (k) return false
        r = curr->next
        pred->next = r
    return true
}
bool remove(int key) {
    Entry *pred, *curr, *r
    restart:
    locate(pred, curr, key)
    atomic
    if (validate)
    {
        k = (curr->key != key)
        if (k) return false
        r = curr->next
        pred->next = r
        return true
    }
    goto restart
}
step 2: Examine Counterexample

Thread 1: add(4)  Thread 2: remove(1)

Addition of key 4 is lost

How to deal with removed nodes?
But What Now?

Space

Synchronization

Time
STEP 3: SYNCHRONIZATION METADATA

\[
\text{REMOVE} = \{
\begin{align*}
\text{R1: } k &= (\text{curr->key} \neq \text{key}) \\
\text{R2: } &\text{if (k) return false} \\
\text{R3: } &\text{curr->marked = true} \\
\text{R4: } &\text{mp = pred->marked} \\
\text{R5: } &\text{mc = curr->marked} \\
\text{R6: } &\text{val = (pred->next == curr) (\land mp)? (\land mc)?} \\
\text{R7: } &\text{if (!val) goto restart} \\
\text{R8: } &\text{r = curr->next} \\
\text{R9: } &\text{pred->next = r}
\end{align*}
\]
bool remove(int key)
{
    Entry *pred,*curr,*r
    restart:
        locate(pred,curr,key)
    REMOVE
        return true
}

REMOVE = {
R1: k = (curr->key ≠ key)
R2: if (k) return false
R3: curr->marked = true
R4: mp = ¬pred->marked
R5: mc = ¬curr->marked
R6: val= (pred->next == curr) ? (¬mp) ? (¬mc)
R7: if (¬val) goto restart
R8: r = curr->next
R9: pred->next = r
}
bool add(int key) {
    Entry *pred, *curr, *entry

    restart:
    locate(pred, curr, key)
    k = (curr->key == key)
    if (k) return false

    entry = new Entry()
    entry->next = curr

    val = CAS(&pred->next,
               curr, 0
               entry, 0)

    if (!val) goto restart

    return true
}

bool remove(int key) {
    Entry *pred, *curr, *r

    restart:
    locate(pred, curr, key)
    k = (curr->key != key)
    if (k) return false

    r = curr->next

    lval = CAS(&curr->next,
               r, m
               r, 1)

    if (!lval) goto restart

    pval = CAS(&pred->next,
               curr, 0
               r, 0)

    if (!pval) goto restart

    return true
}

bool contains(int key) {
    Entry *pred, *curr

    locate(pred, curr, key)
    k = (curr->key == key)
    if (k) return false

    return true
}
Concurrent Set Algorithms Recap

Sequential

DCAS

DCAS

Michael (PODC’02)

Heller et al. (OPODIS’05)

CAS

CAS with LOCKS

CAS/DCAS

Trieber Stack

Priority Queue

Stack

Schema

Correct Algorithm

Existing Algorithm

New Algorithm
Synthesis Process

Machine

verifier

program

synthesizer

yes/counterexample

• Exploration
• Validation

Checked Algorithms

Human

Schema

Checked Algorithms
Key questions

- Are schemas a natural input for a designer?
  - can be difficult for designers, prefer programs

- Why are search and verification separate?
  - Verifier overwhelmed with many incorrect or similar programs

- Can synchronization inference be automated?
  - atomicity, ordering, space (big challenge)
Another approach: verification-driven

- The input to synthesizer is a program
  - synthesizer “repairs” or “fixes” an incorrect program

- Verifier does synthesis
  - can output a program, not just yes/no

- Automates synchronization inference
  - some synchronization (but not all, e.g. space is not)
Synthesizer produces various repairs versions of the program

verifier extended to do synthesis

Synthesizer “repairs” the program by synthesizing synchronization

Checked Algorithms

Human

Program
Challenge

- Find **minimal synchronization** that makes the program satisfy the specification
  - Avoid all bad schedules while permitting as many good schedules as possible

- Assumption: we can prove that serial executions satisfy the specification
  - Interested in bad behaviors due to concurrency

- Handle infinite-state programs
Abstraction-Guided Synthesis of Synchronization

- Synthesis of synchronization via abstract interpretation
  - Compute *over-approximation* of all possible program executions
  - Add *minimal synchronization* to avoid (over-approximation of) bad schedules

- Interplay between abstraction and synchronization
  - Finer abstraction may enable finer synchronization
  - Coarse synchronization may allow coarser abstraction
A Standard Approach: Abstraction Refinement

Program $P$

Specification $S$

Abstraction $\alpha$

Verify

Abstract counter example

Abstract Refinement

Valid

Change the \textit{abstraction} to match the \textit{program}
Abstraction-Guided Synthesis [VYY-POPL’10]

Program $P$

Specification $S$

Abstraction $\alpha$

Program $\varphi$

Program Restriction

Implement

Abstract counter example

Abstract counter example

Change the **program** to match the **abstraction**
AGS Algorithm – High Level

Input: Program $P$, Specification $S$, Abstraction $\alpha$

Output: Program $P'$ satisfying $S$ under $\alpha'$

$\phi = true$

while(true) {
    BadTraces = \{ $\pi \mid \pi \in ([P]_\alpha \cap [\phi])$ and $\pi \not\in S$ \}
    if (BadTraces is empty) return implement($P, \phi$)
    select $\pi \in$ BadTraces
    if (?) {
        $\psi = avoid(\pi)$
        if ($\psi \neq false$) $\phi = \phi \land \psi$
        else abort
    } else {
        $\alpha' = refine(\alpha, \pi)$
        if ($\alpha' \neq \alpha$) $\alpha = \alpha'$
        else abort
    }
}

Input: Program $P$, Specification $S$, Abstraction $\alpha$
Output: Program $P'$ satisfying $S$ under $\alpha'$
Avoid and Implement

- Desired output – program satisfying the spec
  - Constraint has to be implementable in the program
- Implementation mechanism guides the choice of constraint language
- Implementation mechanisms
  - Atomic sections [POPL’10]
  - Conditional critical regions (CCRs) [TACAS’09]
  - Memory fences (for relaxed memory models) [FMCAD’10 + PLDI’11]
  - Barriers [SAS’13]
  - ...

35
Avoiding a schedule with atomic sections

- Adding atomicity constraints
  - Atomicity predicate \([l_1, l_2]\) – statements at \(l_1\) and \(l_2\) must execute together atomically

- \(\text{avoid}(\pi)\)
  - A disjunction of all possible atomicity predicates that would prevent \(\pi\)
  - Captures all ways to avoid \(\pi\) using atomic sections

\[
\text{avoid}(\pi) = \bigvee\{ [\text{label}(\pi_i), \text{label}(\pi_j)] \mid 0 \leq i < |\pi|, j > i + 1. \text{tid}(\pi_i) = \text{tid}(\pi_j) \text{ s.t. } \forall k. i < k < j, \text{tid}(\pi_i) \neq \text{tid}(\pi_k) \}
\]
Avoid with atomicity constraints

\[ \pi = A_1 B_1 A_2 B_2 \]

\[ \text{avoid}(\pi) = [A_1, A_2] \lor [B_1, B_2] \]

Thread A
1: A1
2: A2

Thread B
1: B1
2: B2
Avoiding the good with the bad

- $\psi = \text{avoid}(\pi)$
- Enforcing $\psi$ avoids any abstract trace $\pi'$ such that $\pi' \not\equiv \psi$
- Potentially avoiding “good traces” as well

- $\text{avoid}(\pi_1) = \{A_1, A_2\}$
### Simple Example

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: x += z</td>
<td>1: z++</td>
<td>1: y1 = f(x)</td>
</tr>
<tr>
<td>2: x += z</td>
<td>2: z++</td>
<td>2: y2 = x</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3: assert(y1 != y2)</td>
</tr>
</tbody>
</table>

```plaintext
f(x) {
    if (x == 1) return 3
    else if (x == 2) return 6
    else return 5
}
```

- Initially $x = z = 0$
- Every single statement is atomic
Example: Concrete Values

Concrete values

```
x += z; x += z; z++; z++; y1 = f(x); y2 = x; assert → y1 = 5, y2 = 0
```

```
z++; x += z; y1 = f(x); z++; x += z; y2 = x; assert → y1 = 3, y2 = 3
```

```
... 
```

```
f(x) {
    if (x == 1) return 3
    else if (x == 2) return 6
    else return 5
}
```

```
T1
```

```
1: x += z
2: x += z
```

```
T2
```

```
1: z++
2: z++
```

```
T3
```

```
1: y1 = f(x)
2: y2 = x
3: assert(y1 != y2)
```
Example: Parity Abstraction

Concrete values

Parity abstraction (even/odd)

\[ x += z; x += z; z++; z++; y1 = f(x); y2 = x; \text{assert } \Rightarrow y1 = \text{Odd}, y2 = \text{Even} \]

\[
\begin{array}{c|c|c|c}
\text{T1} & \text{T2} & \text{T3} \\
1: x += z & 1: z++ & 1: y1 = f(x) \\
2: x += z & 2: z++ & 2: y2 = x \\
\end{array}
\]

\[
f(x) \begin{cases} 
\text{if } (x == 1) \text{ return 3} \\
\text{else if } (x == 2) \text{ return 6} \\
\text{else return 5} 
\end{cases}
\]
Example: Avoiding Bad Interleavings

\[ \phi = \text{true} \]

while (true) {

BadTraces = \{ \pi | \pi \in ([P]_a \cap [\phi]) \text{ and } \pi \not\in S \}

if (BadTraces is empty)
    return implement(P, \phi)

select \pi \in \text{BadTraces}

if (\text{?}) {
    \phi = \phi \land \text{avoid}(\pi)
} else {
    \alpha = \text{refine}(\alpha, \pi)
}
}

avoid(\pi_1) = [z++, z++]

\[ \phi = \text{true} \]
Example: Avoiding Bad Interleavings

\[\varphi = \text{true}\]

while(true) {
  \[
  \text{BadTraces} = \{ \pi | \pi \in (\llbracket P \rrbracket_a \cap \llbracket \varphi \rrbracket) \text{ and } \pi \not\in S \}\]
  
  if (BadTraces is empty)
    return implement(P, \varphi)
  
  select \pi \in \text{BadTraces}
  
  if (?) {
    \[\varphi = \varphi \land \text{avoid}(\pi)\]
  } else {
    \[\alpha = \text{refine}(\alpha, \pi)\]
  }

} 

\text{avoid}(\pi_2) = [x+=z, x+=z]

\[\varphi = [z++, z++] \land [x+=z, x+=z]\]
Example: Avoiding Bad Interleavings

\[
\varphi = \text{true}
\]

while (true) {
    BadTraces = \{ \pi \mid \pi \in ([P]_a \cap [\varphi]) \quad \text{and} \quad \pi \notin S \}
    if (BadTraces is empty) {
        return implement (P, \varphi)
    }
    select \pi \in \text{BadTraces}
    if (?) {
        \varphi = \varphi \land \text{avoid}(\pi)
    } else {
        \alpha = \text{refine}(\alpha, \pi)
    }
}

\[
\varphi = [z++, z++] \land [x+=z, x+=z]
\]
Example: Avoiding Bad Interleavings

But we can also change the abstraction...
(a) (b) (c)

parity

T1 \( x+=z; \)
\( x+=z \)
T2 \( z++; \)
\( z++; \)
T3 \( y1=f(x) \)
\( y2=x \)
assert 
\( y1!= y2 \)

interval

T1 \( x+=z; \)
\( x+=z \)
T2 \( z++; \)
\( z++; \)
T3 \( y1=f(x) \)
\( y2=x \)
assert 
\( y1!= y2 \)

octagon

T1 \( x+=z; \)
\( x+=z \)
T2 \( z++; \)
\( z++; \)
T3 \( y1=f(x) \)
\( y2=x \)
assert 
\( y1!= y2 \)
Multiple Solutions

- Interval abstraction for our example produces the atomicity constraint:

\[
([x+=z, x+=z] \lor [z++, z++]) \\
\land ([y_1=f(x), y_2=x] \lor [x+=z, x+=z] \lor [z++, z++])
\]

- Performance: smallest atomic sections

- Minimal satisfying assignments
  - \( \Gamma_1 = [z++, z++] \)
  - \( \Gamma_2 = [x+=z, x+=z] \)
Implementability

- Separation between schedule constraints and how they are realized
  - Can realize in program: atomic sections, locks, ...
  - Can realize in scheduler: benevolent scheduler

<table>
<thead>
<tr>
<th>T1</th>
<th>T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: while(*) {</td>
<td>1: assert (x != 1)</td>
</tr>
<tr>
<td>2: x++</td>
<td></td>
</tr>
<tr>
<td>3: x++</td>
<td></td>
</tr>
<tr>
<td>4: }</td>
<td></td>
</tr>
</tbody>
</table>

- No program transformations (e.g., loop unrolling)
- Memoryless strategy
Abstraction-Guided Synthesis

Program P

Specification S

Abstraction α

Verify

φ

Program Restriction

Implement

P'

Abstract counter example

Abstract counter example

Abstraction Refinement
## AGS instances

<table>
<thead>
<tr>
<th>Avoid</th>
<th>Implementation Mechanism</th>
<th>Abstraction Space (examples)</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context switch</td>
<td>Atomic sections</td>
<td>Numerical abstractions</td>
<td>[POPL’10]</td>
</tr>
<tr>
<td>Inter-thread ordering</td>
<td>Synchronization barriers</td>
<td>Numerical abstractions</td>
<td>[SAS’13]</td>
</tr>
<tr>
<td>Intra-thread ordering</td>
<td>Memory fences</td>
<td>Partial-Coherence abstractions for store buffers</td>
<td>[FMCAD’10]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[PLDI’11]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>[PLDI’12]</td>
</tr>
<tr>
<td>Transition</td>
<td>Conditional critical regions (CCRs)</td>
<td>Observability</td>
<td>[TACAS’09]</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
General Setting Revisited

Change the specification to match the abstraction
Summary

- Synthesis of Concurrent Data Structures
  - Search-based synthesis: schema + semantic search
  - Discovered new algorithmic variants

- Synthesis of Synchronization
  - Input is a program, synthesizer repairs programs
  - Instances: atomics, barriers, memory fences, etc...

- Future Challenges
  - Synthesis of “space” is key
  - Apply techniques in symbolic or dynamic execution
  - Combining synchronization primitives
Thank You

Questions?