Pattern-based Verification of Concurrent Programs

Tomáš Poch, Pierre Ganty

IMDEA internship talk
Jan 20, 2011
Reachability for sequential/concurrent programs

L1: bit = F;
    if bit == T
        goto error;
    else
        goto L1;

error : print "busted";

<table>
<thead>
<tr>
<th>error reachability</th>
<th>unbounded data</th>
<th>bounded data</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential prgs</td>
<td>Undecidable</td>
<td>Decidable</td>
</tr>
</tbody>
</table>
Reachability for sequential/concurrent programs

Thread $t_1$

L1: bit = F;
    if bit == T
    goto error;
    else
    goto L1;

error: print "busted";

Thread $t_2$

L: bit = T;
   goto L;
Reachability for sequential/concurrent programs

Thread $t_1$

L1: bit = F;
    if bit == T
       goto error;
    else
       goto L1;
error:
    print "busted";

Thread $t_2$

L: bit = T;
goto L;

SM: bit
Reachability for sequential/concurrent programs

Thread $t_1$

L1: \[ \text{bit} = \text{F}; \]
    \[ \text{if bit} == \text{T} \]
    \[ \text{goto error}; \]
    \[ \text{else} \]
    \[ \text{goto L1}; \]
error : \[ \text{print "busted"}; \]

Thread $t_2$

L: \[ \text{bit} = \text{T}; \]
   \[ \text{goto L}; \]

SM: bit

Is error reachable?
Reachability for sequential/concurrent programs

Thread $t_1$

$\gg L1$: bit = F;
if bit == T
  goto error;
else
  goto L1;
print "busted";

error :

Thread $t_2$

$\gg L$: bit
goto

SM: bit

Is error reachable?
Reachability for sequential/concurrent programs

Thread $t_1$

L1: bit = F;
    if bit == T
        goto error;
    else
        goto L1;
    print "busted";

Thread $t_2$

L: bit goto

error :

Is error reachable?
Reachability for sequential/concurrent programs

Thread $t_1$

L1: bit = F
if bit = T
  goto error;
else
  goto L1;
print "busted";

error :

Thread $t_2$

L: bit = T;
goto L;

SM: bit

bit = F

Is error reachable?
Reachability for sequential/concurrent programs

Thread $t_1$

L1: bit = F;
if bit = T
  goto error;
else
  goto L1;
  print "busted";

error:

Thread $t_2$

L: bit = T;
goto L;

SM: bit
bit = F

Is error reachable?
Reachability for sequential/concurrent programs

Thread $t_1$

L1: bit = F;
   if bit == T
      goto error;
   else
      goto L1;
   print "busted";

error:

SM: bit

Is error reachable?
Reachability for sequential/concurrent programs

Thread $t_1$

L1: bit = F;
    if bit == T
goto error;
else
    goto L1;

error: print "busted";

Thread $t_2$

L: bit

Is error reachable?
Reachability for sequential/concurrent programs

Thread $t_1$

L1: bit = F;
    if bit == T
        goto error;
    else
        goto L1;
    print "busted";

error:

Thread $t_2$

L: bit = T
    goto

Is error reachable?
Reachability for sequential/concurrent programs

```plaintext
Thread $t_1$

L1: bit = F;
    if bit == T
        goto error;
    else
        goto L1;
    print "busted";

Thread $t_2$

L: bit = T
    goto L;

error:
```

<table>
<thead>
<tr>
<th>error reachability</th>
<th>unbounded data</th>
<th>bounded data</th>
</tr>
</thead>
<tbody>
<tr>
<td>sequential prgs</td>
<td>Undecidable</td>
<td>Decidable</td>
</tr>
<tr>
<td>multithreaded prgs</td>
<td>Undecidable</td>
<td>Undecidable</td>
</tr>
</tbody>
</table>
Reachability for sequential/concurrent programs

Thread $t_1$

L1: bit = F;
    if bit == T
    goto error;
    else
    goto L1;
    print "busted";

→ error :

Thread $t_2$

L: bit = T
    goto

SM: bit

Is error reachable?

Decidability by further restricting SM visited states coverage
Reachability for sequential/concurrent programs

L1: bit = F;
    if bit == T
        goto error;
    else
        goto L1;
    print "busted";

L: bit = T
goto

error : print "busted";

Is error reachable for $k$ writes to SM?

$k$-context switches reachability is decidable, and NPC

* Shaz Qadeer, Jakob Rehof. *Context-Bounded Model Checking of Concurrent Software* in TACAS '05
Reachability for sequential/concurrent programs

Thread $t_1$

L1: bit = F;
    if bit == T
        goto error;
    else
        goto L1;
    print "busted";

Thread $t_2$

L: bit
    goto

error:

Is error reachable for pattern $(bit = F \cdot bit = T)^*$?

Pattern-based reachability is decidable $\dagger$

\[\dagger\] Pierre Ganty, Rupak Majumdar, Benjamin Monmege. *Bounded Underapproximations* in CAV '10
Reachability for sequential/concurrent programs

Pattern takes the form $w_1^* w_2^* \ldots w_n^*$

$w_i$ is a word, symbols represent data in SM

Pattern-based reachability is decidable

† Pierre Ganty, Rupak Majumdar, Benjamin Monmege. *Bounded Underapproximations* in CAV '10
From SM programs to grammars

Shared memory program consist of
- Set of procedures accessing local and global variables (bounded data)
- Set of threads having initial points

Message passing program consist of
- Set of threads and procedures accessing local variables (bounded data), sending/receiving messages
void main()
0  int x = 5;
1  while (x>0) {
2    b();
3    x--;
4  }
5}

void b()
6  { int y;
6    reicv(ch1,y);
7    send(ch2,y);
7}

• The example copies 5 items from the channel ch1 to ch2

• Semantics
  – receive(ch1,var)
    • Block until someone calls send(ch1,data) and all the others call receive(ch1,var2)
    • assign var=data
  – send(ch1,data)
    • Block until all other threads call receive(ch1,var)
Shared memory as message passing

- Modify SM program:
  - All variables are local
  - At each program location simulate context switch
    - Send the content of ‘global’ data
    - Switch to the inactive state
    - Wait until someone else sends the content of memory

- Two messages for each context switch
  - yield(gmem) – go to inactive state
  - go(TID) – go back to the active state
Context switching as msg passing

t₁ computation

\[ \rightarrow \quad \rightarrow \quad \ldots \quad \rightarrow \]

\( t_1 \) is active
Stack changes
No messages produced
Context switching as msg passing

$t_1$ computation

$t_1$ is active
Stack changes
No messages produced
Context switching as msg passing

$t_1$ computation

$t_1$ is active
Stack changes
No messages produced

$yield(\ldots)$
$go\ldots$

$t_1$ inactive
Other threads can go,
and write whatever value

$yield$
Context switching as msg passing

t₁ computation

→ ... →

t₁ is active
Stack changes
No messages produced

→ Yield (bit=..) → yield → ... →

t₁ inactive
Other threads can go, and write whatever value

→ Yield → go (TID=..) → ... →

t₁ is active again
Context switching as msg passing

$t_1$ computation

$t_1$ is active
Stack changes
No terminals produced

Per context switch

$<\text{go, TID}>,<\text{yield}(g_1)>$

$t_1$ inactive
Other threads can go, and write whatever value
Context switching as msg passing

\( t_1 \) computation

\( t_1 \) is active
Stack changes
No terminals produced

\( t_1 \) inactive
Other threads can go, and write whatever value

\( t_2 \) computation

\( t_2 \) inactive

\( t_2 \) is active

\( t_2 \) inactive

\( t_1 \) is active again
Modification of SM program

0  bit=F;
1  if bit==T
2    goto 4;
3  return;
4  print "busted"
Inactive states, communication
<table>
<thead>
<tr>
<th>TID=0</th>
<th>bit=F</th>
<th>bit=T</th>
<th>Inactive bit=F</th>
<th>Inactive bit=T</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Image" /></td>
<td><img src="image2.png" alt="Image" /></td>
<td><img src="image3.png" alt="Image" /></td>
<td><img src="image4.png" alt="Image" /></td>
<td><img src="image5.png" alt="Image" /></td>
</tr>
</tbody>
</table>

**Inactive states, communication**
Inactive states, communication

TID=0

bit=F

Nondet-choice

bit=T

send(yield, bit=F)

Inactive
bit=F

Inactive
bit=T
Inactive states, communication

TID=0

bit=F | bit=T | Inactive bit=F | Inactive bit=T

send(yield, bit=F)

reicv(go, 1)
Inactive states, communication

TID=0

<table>
<thead>
<tr>
<th>bit=F</th>
<th>bit=T</th>
</tr>
</thead>
<tbody>
<tr>
<td>send(yield, bit=F)</td>
<td>reicv(yield, bit=T)</td>
</tr>
</tbody>
</table>

Inactive bit=F

Inactive bit=T

recv(go, 1)
Inactive states, communication

TID=0

- **bit=F**
  - `send(yield, bit=F)`

- **bit=T**
  - `reicv(yield, bit=T)`

- **Inactive bit=F**
  - `reicv(go, 1)`

- **Inactive bit=T**
  - `send(go, 0)`
Inactive states, communication

TID=0

- **bit=F**:
  - `send(yield, bit=F)`
- **bit=T**:
  - `send(yield, bit=T)`
  - `recv(yield, bit=T)`
- **Inactive bit=F**:
  - `recv(yield, bit=F)`
  - `send(go, 0)`
- **Inactive bit=T**:
  - `recv(go, 1)`
  - `recv(yield, bit=F)`
  - `send(go, 0)`
Modification of SM program

TID = 0

0  bit=F;

1  if bit==T

2  goto 4;

3  return;

4  print “busted”

yield, bit=...
go, TID=...
# Modification of SM program

<table>
<thead>
<tr>
<th>TID = 0</th>
<th>bit=F</th>
<th>bit=T</th>
<th>Inactive bit=F</th>
<th>Inactive bit=T</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>bit=F;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>if bit==T</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>goto 4;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>return;</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>print “busted”</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
yield, bit=...
go, TID=...
```
Msg passing prg. as ctx-free grammar

• $t_i$ represented by context-free grammar $G_i$
  – Non-terminals encode program positions and variables
  – Grammar rules simulate program transition
    • Context-free grammar needed to support function calls
  – Terminals encode communication

• $w \in L(G_i)$ – sequence of $t_i$ communications
• $w \in L(G_1) \cap L(G_2)$ – history allowed by both threads

Emptyness of $L(G_1) \cap L(G_2)$ is undecidable
void main()
0  int x = 5;
1  while (x>0) {
2    b();
3    x--;
4  }
5}

void b(){
6  reicv(ch);
7}
void main()
0 int x = 5;
1 while (x>0) {
2   b();
3   x--;
4 }
5
void b(){
6   reicv(ch);
7}
Msg passing as grammar

```c
void main() {
    int x = 5;
    while (x>0) {
        b();
        x--;
    }
}

void b() {
    reicv(ch);
}
```

<table>
<thead>
<tr>
<th>State</th>
<th>Transition</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>( x' = 5, pc ) = 1</td>
<td>[ x \geq 0, \text{pc} = 1 ] -&gt; [ x' = x, \text{pc} = 2 ]</td>
</tr>
<tr>
<td></td>
<td></td>
<td>[ x = 0, \text{pc} = 1 ] -&gt; [ x' = 0, \text{pc} = 5 ]</td>
</tr>
<tr>
<td>2 b()</td>
<td></td>
<td>[ x, \text{pc} = 2 ] -&gt; [ \text{pc} = 6 ][ x' = x, \text{pc} = 3 ]</td>
</tr>
<tr>
<td>3 x--</td>
<td></td>
<td>[ x, \text{pc} = 3 ] -&gt; [ x' = x - 1, \text{pc} = 1 ]</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>[ x, \text{pc} = 5 ] -&gt; ( \epsilon )</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>[ \text{pc} = 6 ] -&gt; &lt;ch&gt;</td>
</tr>
</tbody>
</table>
Msg passing as grammar

```c
void main()
{
    int x = 5;
    while (x>0) {
        b();
        x--;
    }
}
```

```c
void b()
{
    recv(ch);
}
```

<table>
<thead>
<tr>
<th>State</th>
<th>Transition</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Init</td>
<td>x = 5, pc = 1</td>
<td>x = 0, pc = 1, x = x, pc = 2</td>
</tr>
<tr>
<td></td>
<td>x = 0, pc = 1</td>
<td>x = 0, pc = 5</td>
</tr>
<tr>
<td></td>
<td>x = 0, pc = 5</td>
<td>ε</td>
</tr>
<tr>
<td></td>
<td>pc = 6</td>
<td>&lt;ch&gt;</td>
</tr>
</tbody>
</table>

Init \(\rightarrow^*\) <msg><msg><msg><msg><msg><msg>

The process will reach its final position (5) only if the cooperating thread can produce 5 messages as well
Decision procedure

- Reachability in concurrent program as a language problem
  - Intersection of context-free languages
    - Emptiness of \( L(G_1) \cap L(G_2) \) undecidable

- Context bounded verification
  [Shaz Qadeer, Jakob Rehof. Context-Bounded Model Checking of Concurrent Software in Tacas ’05]
  - At most \( k \) context switches
    - Emptiness of \( L(G_1) \cap L(G_2) \cap \{\text{go(TID)}, \text{yield(gmem)}\}^{2k} \)

- Pattern based verification
  [Pierre Ganty, Rupak Majumdar, Benjamin Monmege. Bounded Underapproximations in CAV’10]
  - Context switches follow the pattern
    - Emptiness of \( L(G_1) \cap w_1 \ldots w_n \cap L(G_2) \)
    - \( w_i \in \{\text{go(TID)}, \text{yield(gmem)}\}^* \)
    - Example – at most \( k \) ctx sw : \((\text{go(0)} \ast \text{go(1)} \ast \text{yield(true)} \ast \text{yield(false)} \ast)^k\)
Decision Procedure

$L(G_1) \cap w_1^* \ldots w_n^* \cap L(G_2) = \emptyset$
Decision procedure

• Counting of \( w \) matters
  \[ w_1^i \ w_2^j \ldots \ w_n^k \in L(G) \cap w_1^* \ldots w_n^* \]

• Modify \( G \) to \( G' \)
  – Accept only words from pattern
    • CFL are closed to intersection w/ regular languages
  – Produce single terminal \( a_p \) instead of the word \( w_p \)

• \( w_1^i \ldots w_n^k \in L(G) \cap w_1^* \ldots w_n^* \iff a_1^i \ldots a_n^k \in L(G') \)
Parikh image

- Fixed linear order over alphabet
  - $\Sigma=\{a_1, a_2 \ldots a_p\}$

- Parikh image of $w \in \Sigma^*$ is a $p$-dimensions vector
  - $i$-th part is the number of occurrences of $i$-th symbol in $w$
  - $\Pi(w) = <i_1, i_2, \ldots, i_p>$, $\Pi(a_1a_1a_1a_2) = <3, 1, 0, \ldots, 0>$

- Parikh image of language $L \subseteq \Sigma^*$
  - set of Parikh images of words from $L$
  - $\Pi(L) = \{\pi, \exists w \in L \Pi(w) = \pi\}$

- Parikh image omits the order of symbols
Decision procedure

- \( w_1^i \ldots w_n^k \in L(G) \cap w_1^* \ldots w_n^* \Leftrightarrow a_1^i \ldots a_n^k \in L(G') \)
  - \( a_p \) are distinct, fit on their position by construction of \( G' \)
  - \( \pi \in \Pi(G') \Leftrightarrow a_1^{\pi(1)} \ldots a_n^{\pi(k)} \in L(G') \)

- Parikh image of a context free language can be characterized by an existential Presburger formula
  - \( \Psi_{G'}(\pi) = \text{True} \Leftrightarrow \pi \in \Pi(G') \)

- Satisfiability of existential formula is NP-complete
  [Verma, Seidl, Schwentick: On the Complexity of Equational Horn Clauses, 2005]

\[ \text{Existential Presburger formula } \phi \]
\[ t ::= 0 | 1 | x | t_1 + t_2 | t_1 - t_2 \]
\[ \phi ::= t_1 = t_2 | t_1 > t_2 | \phi_1 \land \phi_2 | \phi_1 \lor \phi_2 | \exists x \phi \]
From Language to Formula

\[ L(G_{T1}) \cap w_1^* \ w_2^* \ \ldots \ w_n^* \cap L(G_{T2}) = \emptyset \]

\[ \iff \]

\[ L(G_{T1}') \cap L(G_{T2}') = \emptyset \]

\[ \iff \]

\[ \Pi(G_{T1}') \cap \Pi(G_{T2}') = \emptyset \]

\[ \iff \]

\[ \Psi_{T1}' \ \& \ \Psi_{T2}' \ \text{is unsatisfiable} \]
Construction of formula

• Petri-net intuition
  – net is simulating the grammar but disregards the ordering of terminals

• Structure
  – Place for each terminal and non-terminal
  – Transition for each rule
  – One token to the initial non-terminal

\[ X \rightarrow aXb \]
PN example

\[ X \rightarrow aXb \]
\[ X \rightarrow \varepsilon \]
PN example

\[
X \rightarrow aXb \\
X \rightarrow \varepsilon
\]
PN example

$X \rightarrow aXb$

$X \rightarrow \varepsilon$
PN example

\[ X \rightarrow aXb \]

\[ X \rightarrow \varepsilon \]
$X \rightarrow axb$

$X \rightarrow \varepsilon$
PN example

\[ X \rightarrow aXb \]
\[ X \rightarrow \epsilon \]
PN example

\[ X \rightarrow aXb \]
\[ X \rightarrow \varepsilon \]

\( L(X) = a^i b^i \)
\( \Pi(X) = \langle i, i \rangle \)

Configurations corresponding \( w \in L(X) \) → all tokens in terminal places
PN examples

\[
\begin{align*}
X & \rightarrow aXb \\
X & \rightarrow \varepsilon \\
X & \rightarrow abX \\
X & \rightarrow \varepsilon \\
X & \rightarrow Xba \\
X & \rightarrow \varepsilon
\end{align*}
\]

Configurations corresponding to \( w \in L(\Pi(X)) \):

\( \Pi(X) = <i,i> \)

\( \rightarrow \) all tokens in terminal places
• Petri net is communication-free
  – Each transition has one input place
  – Context-free grammar (one NT on left-hand side)
• Set of admissible configurations of CF-PN can be characterized by Presburger formula
Formula

- Formula based on
  - Kirchhoff-like rules
    - For each place “# of tokens” = “# of input transition applications” - “# of output transition applications”
  - Reachability rules
    - Each applied transition is reachable from the initial place

- Variables
  - For each place A, $x_A$ is number of tokens in the place, $z_A$ distance from initial place
  - For each transition $y_i$ is the number of applications

Verma, Seidl, Schwentick: On the Complexity of Equational Horn Clauses, 2005
Javier Esparza: Petri Nets, Commutative Context-Free Grammars, and Basic Parallel Processes, 97
Implementation

• Input
  – Control-flow for each thread
  – Definition of global and local variables
  – Pattern

• Goal
  – Transform grammars into formula, run solver
Transformation chain for thread

Petri-net transformation

Instantiation

Instantiated grammar

Pattern

Pattern grammar

Thread Symbolic Grammar

Intersect.

Homomorph

Word $w_p \rightarrow \text{symbol } a_p$

Symbolic grammar

Instantiated grammar

Instantiation

Abstract ph.
Transformation chain for thread

Pattern grammar

Intersect.

Homomorph

Word $w_p \rightarrow \text{symbol } a_p$

Tedious to write manually

Thread Symbolic Grammar

Petri-net transformation

Instantiated grammar

Instantiation

Abstract ph.

Symbolic grammar

Formula
Transformation chain for thread

- Data definition
- Data grammar
- Intersect.
- Ctx Sw
- Restrict
- Intersect.
- Pattern
- Pattern grammar
- Petri-net transformation
- Instantiated grammar
- Instantiation
- Symbolic grammar
- Formula
- Abstract ph.

- Ctrlflow grammar
- Homomorph
  \( w_p \rightarrow a_p \)

- Keep only yield/go
Input – Ctrlflow

• In form of context-free grammar
  – Non-terminals – program locations
  – Terminals – data access
Input – Ctrlflow

• In form of context-free grammar
  – Non-terminals – program locations
  – Terminals – data test/operations

L0  bit=F;
L1  if bit==T
L2  goto 4;
L3  return;
L4  print “busted”
Input – Ctrlflow

• In form of context-free grammar
  – Non-terminals – program locations
  – Terminals – data test/operations

L0  bit=F;
L0 -> <bit=F> L1

L1  if bit==T
L1 -> <bit==T> L2
L1 -> <bit==F> L3

L2  goto 4;
L2 -> L4

L3  return;
L3 -> ε

L4  print “busted”
L4 print “busted”
Input – data definition

- Used to generate data grammar

- Data grammar
  - Non-terminals – data value
  - Terminals – data access
    - The same terminals are in control flow grammar
  - Regular rules, symbolic
    - $[x=C, y=D] \rightarrow <x==C>[x=C, y=D]$

- Generated language ~ valid memory behavior
  - $<x=4>$ can be followed by $<x==4>$ but not $<x==5>$
  - Intersected with the control flow grammar to provide semantics
Input - pattern

- Pattern expression
  - List of words (yield/go)
  - Transformed into regular grammar
Transformation chain for thread

- Mem definition
- Data grammar
- Intersect.
- Ctrlflow grammar
- Intersect.
- Ctx Sw
- Restrict
- Pattern
- Pattern grammar
- Intersect.

- Petri-net transformation
- Instantiated grammar
- Instantiation
- Symbolic grammar
- Abstract ph.
- Formula

Keep only yield/go
Word $w_p \rightarrow$ symbol $a_p$
Homomorph
Instantiation

• Non-terminals in symbolic rules use variables
  – One symbolic rule stands for number of ‘ground’ rules

• Petri-net transformation cannot process symbolic rules
  – Formula needs to have fixed number of variables
    (~nonterminals,rules)
Instantiation

- **Goal** – Provide the set of grounded rules used by the grammar
  - Omit unreachable combinations of program positions and variables
- **Algorithm**
  - Find all non-terms reachable (from initial non-terminal) and generating (can be rewritten to sequence of terminals)
  - Transitive closure
Instantiation

- **Goal** – Provide the set of grounded rules used by the grammar
  - Omit unreachable combinations of program positions and variables
- **Algorithm**
  - Find all non-terms reachable (from initial non-terminal) and generating (can be rewritten to sequence of terminals)
  - Transitive closure
Instantiation

- **Goal** – Provide the set of grounded rules used by the grammar
  - Omit unreachable combinations of program positions and variables
- **Algorithm**
  - Find all non-terms reachable (from initial non-terminal) and generating (can be rewritten to sequence of terminals)
  - Transitive closure
Instantiation

- Experience: Number of non-terminals in the first phase tends to be large, intersection is relatively small

- Abstraction phase
  - Omit the local variables
  - Run the instantiation to get legal combinations of global variables and program locations A.
  - Run the instantiation on the original grammar, use A as superset of the result
Instantiation

- Experience: Number of non-terminals in the first phase tends to be large, intersection is relatively small

- Abstraction phase
  - Omit the local variables
  - Run the instantiation to get legal combinations of global variables and program locations A.
  - Run the instantiation on the original grammar, use A as superset of the result
Instantiation

- Experience: Number of non-terminals in the first phase tends to be large, intersection is relatively small

- Abstraction phase
  - Omit the local variables
  - Run the instantiation to get legal combinations of global variables and program locations A.
  - Run the instantiation on the original grammar, use A as superset of the result
Instantiation

- Experience: Number of non-terminals in the first phase tends to be large, intersection is relatively small

- Abstraction phase
  - Omit the local variables
  - Run the instantiation to get legal combinations of global variables and program locations A.
  - Run the instantiation on the original grammar, use A as superset of the result
Experience: Number of non-terminals in the first phase tends to be large, intersection is relatively small

Abstraction phase
- Omit the local variables
- Run the instantiation to get legal combinations of global variables and program locations $A$.
- Run the instantiation on the original grammar, use $A$ as superset of the result
Transformation chain

Data definition

T₁ Ctrlflow \rightarrow \text{Chain} \rightarrow T₁ formula

T₂ Ctrlflow \rightarrow \text{Chain} \rightarrow T₂ formula

Tₙ Ctrlflow \rightarrow \text{Chain} \rightarrow Tₙ formula

Formula

Solver

Satisfiable \rightarrow \text{point is reachable}

Trace given by solution
Validation

- Windows NT bluetooth driver example
  - Several variants, race conditions reported in Suwimonteerabuth, Esparza, Schwoon: Symbolic Context-Bounded Analysis of Multithreaded Java Programs, SPIN '08
  - We can detect them all, given the proper pattern
- Still toy example
  - Input simplified to preserve essence of the bug

<table>
<thead>
<tr>
<th>Task</th>
<th>Formula clauses</th>
<th>Transform Time[s]</th>
<th>Yices Time[s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>bt1</td>
<td>1074</td>
<td>62</td>
<td>1</td>
</tr>
<tr>
<td>bt2</td>
<td>6664</td>
<td>1003</td>
<td>1</td>
</tr>
<tr>
<td>bt2</td>
<td>2924</td>
<td>816</td>
<td>1</td>
</tr>
<tr>
<td>cavpp</td>
<td>5840</td>
<td>12</td>
<td>236</td>
</tr>
</tbody>
</table>

The size of data bothers the transformation, length of trace bothers yices
Abstraction phase helps

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bt1/add</td>
<td>skip</td>
<td>skip</td>
<td>465822</td>
<td>326</td>
<td>438</td>
</tr>
<tr>
<td>bt1/add</td>
<td>46005</td>
<td>251</td>
<td>2394</td>
<td>326</td>
<td>26</td>
</tr>
<tr>
<td>bt3/add</td>
<td>150022</td>
<td>922</td>
<td>7445</td>
<td>787</td>
<td>280</td>
</tr>
<tr>
<td>bt3/stop</td>
<td>294773</td>
<td>520</td>
<td>3557</td>
<td>48</td>
<td>257</td>
</tr>
</tbody>
</table>

• Two variants of BT example runs out of memory if the abstraction phase is off
Conclusion

• Theory works
  – Bluetooth example is small, but real

• Tool runs
  – Lots of technical details solved
  – Provides result for all toy examples we have

• Future directions
  – More, larger, examples
  – Instantiation phase
    • Skip it – push the instantiation phase into formula
    • Smarter approximation (e.g. abstract interpretation)
Thank you