Part 2: Reachability analysis of stack-based systems

From Finite to Infinite-State Systems
- So far, algorithms for systems with finite state spaces
  - semi-algorithms in the presence of recursion

Decidability of reachability analysis

<table>
<thead>
<tr>
<th>Single thread of control:</th>
<th>Control</th>
<th>Finite</th>
<th>Acyclic</th>
<th>Looping</th>
<th>Infinite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finite</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Infinite</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

Decidability of reachability analysis

<table>
<thead>
<tr>
<th>Multiple threads of control:</th>
<th>Control</th>
<th>Finite</th>
<th>Acyclic</th>
<th>Looping</th>
<th>Infinite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Finite</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Infinite</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td></td>
</tr>
</tbody>
</table>

Decidability vs. Expressiveness
- Unbounded state ≠ Undecidable
- Is the unbounded system able to encode a Turing machine?
  - Single-counter machines? NO
  - Two-counter machines? YES
  - Single-stack machines? NO
  - Two-stack machines? YES

State representation
- Explicit representation infeasible
- Symbolic representation is the key
  - For the transition system
  - For the reachable states
Pushdown systems

\[(G, L, g_0, l_0, \to)\]

- \(g, h \in G\) : finite set of control states
- \(l, m \in L\) : finite set of stack symbols
- \(g_0\) : initial control state
- \(l_0\) : initial stack symbol
- \(\to\) : set of transitions

Remarks

The classical definition of a pushdown system has, in addition, an alphabet \(I\) of input symbols.

Each transition depends on the control state, the top of the stack, and the input symbol.

The language \(L \subseteq I^*\) of a classical pushdown system contains those input sequences for which there is an execution leading to the empty stack.

We are only concerned with reachability analysis and will therefore ignore \(I\).

Three kinds of transitions:

- \((g, l) \rightarrow (h, m)\) (step)
- \((g, l) \rightarrow (h, m n)\) (call)
- \((g, l) \rightarrow (h, e)\) (return)

Configuration: \(g, \begin{array}{c} l \hline \vdots \hline m \end{array} \Rightarrow h, \begin{array}{c} l \hline \vdots \hline n \end{array}\)

Modeling sequential programs

- An element in \(G\) is a valuation to global variables
- An element in \(L\) is a valuation to local variables and
  - current instruction address for the frame at the top of the stack
  - return instruction address for the other frames

Example

```c
bool a = F;
void main() {
    L1: a = T;
    L2: flip(a);
    L3: }
    void flip(bool x) {
    L4: a = !x;
    L5: }
```

Reachability problem

Given pushdown system \((G, L, g_0, l_0, \to)\) and control state \(g\), does there exist a stack \(l \in L^*\) such that \((g_0, l_0) \Rightarrow^* (g, l_0)\)?
**Naïve algorithm**

Add \((g_0, l_0)\) to \(R\)

\( (g, ls) \in R \quad \Rightarrow \quad (g', ls') \)

Add \((g', ls')\) to \(R\)

---

**Problem with the naïve algorithm**

- \(R\) is unbounded so algorithm won’t terminate
- Two solutions:
  - Summary-based (a.k.a. interprocedural dataflow analysis)
  - Automata-based

---

**Automata-based algorithm**

Add \((g_0, l_0)\) to \(R\)

\( (g, ls) \in R \quad \Rightarrow \quad (g', ls') \)

Add \((g', ls')\) to \(R\)

Key idea:
Use a finite automaton to symbolically represent \(R\)

---

**Symbolic representation**

Pushdown system \((G, L, g_0, l_0, \rightarrow)\)

Representation automaton \((Q, L, T, G, F)\)
- \(Q \supseteq G\) is the set of states
- \(L\) is the alphabet
- \(T\) is the transition relation
- \(G\) is the set of initial states
- \(F\) is the set of final states

---

**Remarks**

The classical definition of a pushdown system has, in addition, an alphabet \(I\) of input symbols.

Buchi’s theorem does not contradict the fact that pushdown systems can accept non-regular languages over the input alphabet \(I\).

The language of reachable stack configurations is a language over the alphabet \(L\).
The accepted language is a language over the alphabet \(I\).
Pushdown system:
\[(G, L, g_0, l_0, \rightarrow)\]
- \(G = (g_0, g_1, g_2)\)
- \(L = \{l_0, l_1, l_2\}\)
- \((g_0, l_0) \rightarrow (g_1, l_1)\)
- \((g_1, l_1) \rightarrow (g_2, l_1)\)
- \((g_2, l_1) \rightarrow (g_0, l_0)\)
- \((g_0, l_0) \rightarrow (g_0, e)\)

Reachability analysis for concurrent pushdown systems
- Undecidable in general
- Three approaches
  - restrict computation model, e.g., Esparza-Podolski 00
  - sound and imprecise approaches, e.g., Bouajjani-Esparza-Touili 03, Flanagan-Qadeer 03
  - unsound but precise approaches

Context-bounded verification of concurrent software

Why context-bounded analysis?
- Many subtle concurrency errors are manifested in executions with a small number of context switches
- Context-bounded analysis can be performed efficiently

Analyze all executions with small number of context switches

Different from bounded-depth model checking
- no bound on the computation within each context
KISS: a static analysis tool

- Technique to use any sequential checker to perform context-bounded concurrency analysis
- Found a number of concurrency errors in NT device drivers even with a context-switch bound of two

<table>
<thead>
<tr>
<th>Driver</th>
<th>KLOC</th>
<th># Fields</th>
<th># Races</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trace devour</td>
<td>0.5</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>Malfilter</td>
<td>1.0</td>
<td>14</td>
<td>0</td>
</tr>
<tr>
<td>Kufilter</td>
<td>1.1</td>
<td>15</td>
<td>0</td>
</tr>
<tr>
<td>Imce</td>
<td>1.1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>Startia</td>
<td>1.1</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>Toaster/toastmon</td>
<td>1.4</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Diskperf</td>
<td>2.4</td>
<td>16</td>
<td>0</td>
</tr>
<tr>
<td>1394mod</td>
<td>2.7</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>1394modv</td>
<td>2.8</td>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>Fakeboard</td>
<td>2.8</td>
<td>39</td>
<td>6</td>
</tr>
<tr>
<td>Toaster/bus</td>
<td>5.0</td>
<td>30</td>
<td>0</td>
</tr>
<tr>
<td>Serenium</td>
<td>5.9</td>
<td>41</td>
<td>2</td>
</tr>
<tr>
<td>Toaster/func</td>
<td>6.6</td>
<td>24</td>
<td>5</td>
</tr>
<tr>
<td>Mouclass</td>
<td>7.0</td>
<td>34</td>
<td>1</td>
</tr>
<tr>
<td>Blobclass</td>
<td>7.4</td>
<td>36</td>
<td>1</td>
</tr>
<tr>
<td>Mouser</td>
<td>7.6</td>
<td>34</td>
<td>1</td>
</tr>
<tr>
<td>Fido</td>
<td>9.2</td>
<td>92</td>
<td>9</td>
</tr>
</tbody>
</table>

Total: 30 races

Zing: an explicit-state model checker

- Case study (Naik-Rehof 04): Concurrent transaction management code from Microsoft product group
- Analyzed by the Zing model checker after automatically translating to the Zing input language
  - Found three bugs each requiring between three and four context switches

Why context-bounded analysis?

- Many subtle concurrency errors are manifested in executions with a small number of context switches
- Context-bounded analysis can be performed efficiently

Polynomially-bounded executions

- Context bounding leads to polynomial bound on the number of executions
  - n threads, each executing k steps
  - total no. of executions = \( \Omega(n^k) \)
  - With context bound c, no. of executions = \( O((n^c.k)^2) \)
Reachability analysis

- Reachability analysis of finite-data concurrent programs is decidable for bounded number of context switches

**KISS: A static checker for concurrent software**

No error found

<table>
<thead>
<tr>
<th>Concurrent program P</th>
<th>KISS</th>
<th>Sequential program Q</th>
<th>Sequential Checker</th>
</tr>
</thead>
<tbody>
<tr>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Error in Q indicates error in P</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**KISS strategy**

- Q encodes executions of P with small number of context switches
  - Instrumentation introduces lots of extra paths to mimic context switches
- Leverage all-path analysis of sequential checkers
KISS features (I)

- KISS trades off soundness for scalability
- Sound for event-driven systems
  - embedded software, TinyOS
- Unsoundness is precisely quantifiable for other systems
  - e.g., for 2-thread program, explores all executions with up to two context switches

KISS features (II)

- Cost of analyzing a concurrent program \( P \) = cost of analyzing a sequential program \( Q \)
  - Size of \( Q \) asymptotically same as size of \( P \)
- Allows any sequential checker to analyze concurrency
However...

- Hard limit on number of explored contexts
- E.g., two context switches for concurrent program with two threads

Is a tuning knob possible?

Given a concurrent boolean program $P$ and a positive integer $c$, does $P$ go wrong by failing an assertion via an execution with at most $c$ contexts?

Sequential boolean program

Global store $g$, valuation to global variables
Local store $l$, valuation to local variables
Stack $s$, sequence of local stores
State $(g, s)$

Problem:
- Unbounded computation possible within each context!
- Unbounded execution depth and reachable state space
- Different from bounded-depth model checking

Reachability problem for sequential boolean program

$\text{Reach}(g, s) = \{ (g', s') \mid (g, s) \rightarrow^* (g', s') \}$

Given $(g, s)$, is there $s'$ such that $(g, s) \rightarrow^* \text{(error, s')}$?
**Aggregate state**

Set of stacks \( \mathcal{S} \)  
Aggregate state \( (g, \mathcal{S}) = \{(g, s) \mid s \in \mathcal{S}\} \)  
Reach \( (g, \mathcal{S}) = \bigcup \{\text{Reach}(g, s) \mid s \in \mathcal{S}\} \)

**Aggregate transition relation**

- Suppose \( G = \{g_1, \ldots, g_n\} \)  
- There is a unique partition of Reach \((g, \mathcal{S})\) into aggregate states: \( (g_1', \mathcal{S}_1') \cup \cdots \cup (g_n', \mathcal{S}_n') \)  
\( (g, \mathcal{S}) \Rightarrow (g_1', \mathcal{S}_1') \)  
\( (g, \mathcal{S}) \Rightarrow (g_2', \mathcal{S}_2') \)  
\( \ldots \)  
\( (g, \mathcal{S}) \Rightarrow (g_n', \mathcal{S}_n') \)  
iff Reach \((g, \mathcal{S}) = (g_1', \mathcal{S}_1') \cup \cdots \cup (g_n', \mathcal{S}_n') \)

**Theorem (Buchi, Schwoon00)**

- If \( \mathcal{S} \) is regular and \( (g, \mathcal{S}) \Rightarrow (g', \mathcal{S}') \), then \( \mathcal{S}' \) is regular.  
- If \( \mathcal{S} \) is given as a finite automaton \( A \), then a finite automaton \( A' \) for \( \mathcal{S}' \) can be constructed from \( A \) in polynomial time.

**Algorithm**

Problem:  
Given \((g, s)\), is there \( s' \) such that \((g, s) \Rightarrow^{*} \text{(error, } s')\)?

Solution:  
Compute automaton for \( \mathcal{S}' \) such that \((g, \{s\}) \Rightarrow \text{(error, } \mathcal{S}')\) and check if \( \mathcal{S}' \) is nonempty.

**Concurrent boolean program**

Global store \( g \), valuation to global variables  
Local store \( i \), valuation to local variables  
Stack \( s \), sequence of local stores  
State \( (g, s_1, s_2) \)

Transition relation:  
\((g, s_1) \rightarrow (g', s_1')\) in thread 1  
\((g, s_2) \rightarrow (g', s_2')\) in thread 2  
\((g, s_1, s_2) \rightarrow (g', s_1', s_2')\)

**Reachability problem for concurrent boolean program**

Given \((g, s_1, s_2)\), are there \( s_1' \) and \( s_2' \) such that \((g, s_1, s_2)\) reaches \((\text{error, } s_1', s_2')\) via an execution with at most \( c \) contexts?
**Aggregate transition relation**

\[ (g, ss_1, ss_2) = \{ (g, s_1, s_2) \mid s_1 \in ss_1, s_2 \in ss_2 \} \]

\[
\begin{align*}
(g, ss_1) & \Rightarrow (g', ss'_1) \text{ in thread 1} \\
(g, ss_1, ss_2) & \Rightarrow (g', ss'_1, ss_2)
\end{align*}
\]

\[
\begin{align*}
(g, ss_2) & \Rightarrow (g', ss'_2) \text{ in thread 2} \\
(g, ss_1, ss_2) & \Rightarrow (g', ss'_1, ss'_2)
\end{align*}
\]

**Algorithm: 2 threads, c contexts**

Compute the set of reachable aggregate states. Report an error if \((g, ss_1, ss_2)\) is reachable and \(g \equiv \text{error}, ss_1 \) is nonempty, and \(ss_2 \) is nonempty.

**Complexity: 2 threads, c contexts**

Depth of tree = context bound \(c\)
Branching factor bounded by \(G \times 2 (G = \# \text{ of global stores})\)
Number of edges bounded by \((G \times 2)^{(c-1)}\)
Each edge computable in polynomial time

**Results**

- Algorithm for checking if a concurrent boolean program \(P\) fails an assertion via an execution with at most \(c\) contexts
- Algorithm for checking if a concurrent boolean program \(P\) with unbounded fork-join parallelism fails an assertion via an execution with at most \(c\) contexts