Automata-Based Analysis of Recursive Concurrent Programs

Markus Müller-Olm
Westfälische Wilhelms-Universität Münster, Germany

2nd Tutorial of
SPP RS3: Reliably Secure Software Systems

Schloss Buchenau, September 3-6, 2012
Introduction

- Optimal Analysis:
  - Complete analysis of well-specified abstract model

- Threads & recursive procedures

- Locks & monitors

- Regular model checking
Lock-join-sensitive Analysis

Main:

0
output (x)

1
acquire l

2
output (x)

3
spawn P

4
release l

5
join

6
output (x)

7

P:

0
acquire l

1
x := secret

2
release l

3
x := 0

4

Power of different analyses:

- Pure lock sets: 3
- Analysis sensitive to thread creation, e.g., [BMOT05], [LMO07]: 1, 2
- Lock-sensitive analysis [LMO08], [LMOW09]: 1, 2, 3, 4
- Lock-join-sensitive analysis [GLMOSW11]: 1, 2, 3, 4, 5, 6

Of course, also treat branching, loops, recursion!
Reachability Analysis of Programs with Procedures and Thread Creation

Theorem [Ramalingam]

Reachability is **undecidable** in programs with two threads, synchronous communication, and procedures.

Proof:

Reduction of intersection problem \((L_1 \cap L_2 \neq \emptyset)\) of contextfree languages \(L_1,L_2\).

\[\Rightarrow\] abstract from synchronous communication (for now).
A Model of Recursive Programs with Thread-creation: DPNs: Dynamic Pushdown-Networks

- A *dynamic pushdown-network (DPN)* over finite set of actions $\text{Act}$ consists of:
  - $P$, a finite set of control symbols
  - $\Gamma$, a finite set of stack symbols
  - $\Delta$, a finite set of rules of the following form
    
    $p\gamma \rightarrow^a p_1w_1$  [ with $|w_1| \leq 2$ ]
    
    $p\gamma \rightarrow^a p_1w_1 \triangleright p_2w_2$  [ with $|w_1|=1$ and $|w_2|=1$ ]

    (with $p,p_1,p_2 \in P$, $\gamma \in \Gamma$, $w_1,w_2 \in \Gamma^*$, $a \in \text{Act}$).

- DPNs can model recursive programs with thread-creation primitives using finite abstractions of (thread-local) global variables and local variables of procedures.
A Configuration of a DPN is a word in $\left(P\Gamma^*\right)^+$:

$$p_1w_1p_2w_2\cdots p_kw_k$$

(with $p_i \in P, w_i \in \Gamma^*, k > 0$)

... an infinite state space

The transition relation of a DPN:

$$\left(p\gamma \xrightarrow{a} p_1w_1\right) \in \Delta: \quad u\ p\gamma v \xrightarrow{a} u\ p_1w_1v$$

$$\left(p\gamma \xrightarrow{a} p_1w_1 \triangleright p_2w_2\right) \in \Delta: \quad u\ p\gamma v \xrightarrow{a} u\ p_2w_2p_1w_1v$$
Example

Consider the following DPN with a single rule

\[ p\gamma \xrightarrow{a} p\gamma \\delta \theta q \gamma \]

Transitions:

\[ p\gamma \]
\[ q\gamma \delta \theta \gamma \]
\[ q\gamma \delta \theta \gamma \delta \theta \gamma \]
\[ q\gamma \delta \theta \gamma \delta \theta \gamma \delta \theta \gamma \]
\[ q\gamma \delta \theta \gamma \delta \theta \gamma \delta \theta \gamma \delta \theta \gamma \]
\[ \vdots \]
\[ \vdots \]
Recursive Programs with Thread Creation

Procedures

Recursive procedure calls

Basic actions

P:

0

A

call P

1

B

2

3

spawn Q

Q:

4

C

call Q

5

D

6

7

Spawn commands

Entry point, \( e_q \), of Q

Return point, \( x_q \), of Q

+ finite-state abstraction of (thread-local) global and local variables
Modelling Programs with DPNs

à la [Esparza/Knoop, FOSSACS'99]

for basic edge $e$: $u \xrightarrow{e} v$

$$
g(l, u) \xrightarrow{e} g'(l', v), \quad \text{if } ((g, l), (g', l')) \in [A]^#$$

for call edge $e$: call $P$

$$
g(l, u) \xrightarrow{e} g(l_{\text{init}}, e_P)(l, v)$$

for return point of each procedure

$$
g(l, r_p) \xrightarrow{\text{ret}} g$$

for spawn edge $e$: spawn $P$

$$
g(l, u) \xrightarrow{e} g(l, v) \triangleright g_{\text{init}}(l_{\text{init}}, e_P)$$

abstraction of global state

abstraction of local state

current control point
Spawns are Fundamentally Different

P induces trace language: \[ L = \bigcup \{ A^n \cdot (B^m \otimes (C^i \cdot D^j)) \mid n \geq m \geq 0, i \geq j \geq 0 \} \]

Cannot characterize L by constraint system with „·“ and „⊗“. Trace languages of DPNs differ from those of PA processes.
[Bouajjani, MO, Touili: CONCUR 2005]
Basic Results on Reachability Analysis of DPNs

[Bouajjani, MO, Touili, CONCUR 2005]

Definition

\[
\text{pre}^*[L](C) := \{ c \mid \exists d \in C, w \in L : c \xrightarrow{w} d \} \\
\text{post}^*[L](C) := \{ d \mid \exists c \in C, w \in L : c \xrightarrow{w} d \}
\]

Forward-Reachability

1) \text{post}^*[\text{Act}^*](C) is in general non-regular for regular C.

2) \text{post}^*[\text{Act}^*](C) is effectively context-free for context-free C (in polyn. time).

3) Membership in \text{post}^*[L](C) is in general undecidable for regular L.

Backward-Reachability

1) \text{pre}^*[\text{A}^*](C) is effectively regular for regular C and A \subseteq \text{Act} (in polyn. time).

2) Membership in \text{pre}^*[L](C) is in general undecidable for regular L.

Single Steps

1) \text{pre}^*[A](C) and \text{post}^*[A](C) are effectively regular for regular C and A \subseteq \text{Act} (in polyn. time).
Example: Backward Reachability Analysis for DPNs

Consider again DPN with the rule

\[ p\gamma \xrightarrow{a} p\gamma \triangleright q\gamma \]

and the infinite set of states

\[ \text{Bad} = (q\gamma q\gamma p\gamma^+)^+ = L(A) \]

Analysis problem: can Bad be reached from p\gamma?
Example: Backward Reachability Analysis for DPNs

1. Step: Saturate automaton for Bad with the DPN rule:

Resulting automaton $A_{pre^*}$ represents $pre^*(Bad)$!

2. Step: Check, whether $p\gamma$ is accepted by $A_{pre^*}$ or not

Result: Bad is reachable from $p\gamma$, as $A_{pre^*}$ accepts $p\gamma$!
Some Applications of $\text{pre}^*$-Computations with unrestricted $L$ (i.e. $L = \text{Act}^*$)

Reachability of regular sets of configurations
Set $\text{Bad}$ of configurations is reachable from initial configuration $p_0 \gamma_0$
iff
$p_0 \gamma_0 \in \text{pre}^*\left[\text{Act}^*\right](\text{Bad})$

Bounded model checking
By iterated $\text{pre}^*$-computations alternating with single steps corresponding to synchronizations/communications

Bit-vector data-flow analysis problems
Variable $x$ is live at program point $u$
iff
$e_{\text{Main}} \in \text{pre}^*\left[\text{Act}^*\right]\left(At_u \cap \text{pre}^*\left[\text{NonDef}_x^*\right]\left(\text{pre}^*\left[\text{Use}_x\right](\text{Conf})\right)\right)\right)$

used in JMoped of Schwoon/Esparza
Exploiting a Tree-Shaped View of Configurations
CDPNs: Constrained Dynamic Pushdown-Networks

Idea:
Add (regular, stable) pre-conditions over current control symbols of
children threads to DPN rules.

A *constrained dynamic pushdown-network* (CDPN) consists of:

- $P$, a finite set of control symbols
- $\Gamma$, a finite set of stack symbols
- $\Delta$, a finite set of rules of the following form

$$
\phi: p\gamma \xrightarrow{a} p_1w_1 \quad \text{where } \phi \subseteq P^*
$$

$$
\phi: p\gamma \xrightarrow{a} p_1w_1 \triangleright p_2w_2 \quad \text{where } \phi \subseteq P^*
$$

(with $p, p_1, p_2 \in P$, $\gamma \in \Gamma$, $w_1, w_2 \in \Gamma'$, $a \in \text{Act}$)
Example: A CDPN

1. Phase: \[ p\gamma \xrightarrow{a} p\gamma \triangleright q_0\gamma \]
   \[ \phi: p\gamma \xrightarrow{b} p' \quad \text{with } \phi = ((q_1 + q_2)q_2)^* \]
   \[ q_0\gamma \xrightarrow{d} q_1\gamma \]
   \[ q_1\gamma \xrightarrow{e} q_2\gamma \]

2. Phase: \[ p'\gamma \xrightarrow{c} p' \]

Constraint \( \phi \) means: Proceed to second phase only if:
- an even number of children threads has been created,
- each second child has terminated, and
- each child has performed at least one step.
Reachability Analysis of CDPNs

Definition
Constraint $\phi$ is called **stable for $\Delta$** if:

\[
upv \in \phi, (\psi : p \gamma^a \rightarrow p_1 w_1) \in \Delta \quad \text{implies} \quad up_1 v \in \phi, \text{ and}
\]

\[
upv \in \phi, (\psi : p \gamma^a \rightarrow p_1 w_1 > p_2 w_2) \in \Delta \quad \text{implies} \quad up_1 v \in \phi
\]

Theorem for CDPNs [Bouajjani, MO, Touili, CONCUR 2005]

pre*[Act*](C) is effectively regular for regular C and $A \subseteq \text{Act}$, if all constraints $\phi$ occuring in rules of the CDPN are regular and stable for $\Delta$.

Problem at least PSPACE-hard

Modelling power of stable constraints:
Parallel procedure calls, various join-statements, return values from parallel procedure calls, phased execution.
Modelling Power of Stable Regular Constraints

As a preparation: Indicate termination of son threads by a special control state §:

\[ g\langle l, u \rangle \xrightarrow{e} g\langle l, v \rangle > g_{\text{init}}\langle l_{\text{init}}, e_P \rangle \]

for spawn edge \( e \):

\[ \text{spawn } P \]

one special type of rules:

\[ p\$ \xrightarrow{e} \$

Model parallel call to two procedures:

\[ g\langle l, u \rangle \xrightarrow{p||} g\langle l, u^1 \rangle > g_{\text{init}}\langle l_{\text{init}}, e_P \rangle \]

\[ g\langle l, u^1 \rangle \xrightarrow{j\&Q} g\langle l, u^2 \rangle > g_{\text{init}}\langle l_{\text{init}}, e_Q \rangle \]

\[ P*\$\$ : g\langle l, u^2 \rangle \xrightarrow{p||Q} g\langle l, v \rangle \]
Modelling Power of Stable Regular Constraints (Ctd.)

Model various types of join-statements:
- proceed if all children have terminated: \( \$^*: \ldots \)
- proceed if last child has terminated: \( P^*\$: \ldots \)
- proceed if some child has terminated: \( P^*\$P^*: \ldots \)
- proceed if every second child has terminated: \( (P^\$)^*(P+\epsilon): \ldots \)
- ...

Model return values of parallel procedures (beyond PA!):

\[
P^*\$^p\$^q_g: g \langle l, u^2 \rangle \xrightarrow{p\parallel Q} g_{pq} \langle l, v \rangle
\]

Model phased execution

...
Synchronization via Locks

- Assume finite set of locks

- Have acquire- and release actions
  - \( \text{acq } L, \text{ rel } L \in \text{Act} \) f.a. locks \( L \)

- Intuition: At any time a lock can be hold by at most one thread

- Goal of lock-sensitive analysis
The Results of Kahlon and Gupta

Theorem 1  [Kahlon/Gupta, LICS 2006]

Reachability is undecidable for two pushdown-systems running in parallel and synchronizing by release- and acquire-operations used in an unstructured way.

Idea: Can simulate synchronous communication

Theorem 2  [Kahlon/Gupta, LICS 2006]

Reachability is decidable for two pushdown-systems running in parallel and synchronizing by release- and acquire-operations used in a nested fashion.

Idea: Collect information about lock usage of each process in "acquisition histories" and check mutual consistency of the collected histories.

Our goal: Lock-sensitive analysis for systems with thread creation
Example: Locksets are not Precise Enough

Thread 1:
   acquire L1
   acquire L2
   release L2
X:

Must-Lockset computed at X: \{ L1 \}

Thread 2:
   acquire L2;
   acquire L1;
   release L1;
Y:

Must-Lockset computed at Y: \{ L2 \}

We have disjoint locksets at X and Y: \{ L1 \} \cap \{ L2 \} = \{ \}.

Nevertheless, X and Y are not reachable simultaneously!
A Tree-Based View of Executions: Action Trees

A DPN:

\[ p\gamma \xrightarrow{sp} p\gamma \triangleright q_0\gamma \quad q_0\gamma \xrightarrow{d} q_1\gamma \]
\[ q_1\gamma \xrightarrow{e} q_2\gamma \]

Action sequences:

Action tree:

We write: \[ p\gamma \xrightarrow{T} q_2\gamma q_1\gamma p\gamma \]
A Tree-Based View of Executions

Definition

\[
\text{pre}^*[L](C) := \{ c \mid \exists d \in C, w \in L : c \xrightarrow{w}^* d \}
\quad \text{where } L \subseteq \text{Act}^*
\]

\[
\text{preT}^*[M](C) := \{ c \mid \exists d \in C, T \in M : c \xrightarrow{T}^* d \}
\quad \text{where } M \subseteq \text{Trees}(\text{Act})
\]

Recall:

Membership in \(\text{pre}^*[L](C)\) is undecidable for regular \(L\) already for very simple languages \(C\) (e.g. singletons).

Theorem for DPNs [Lammich, MO, Wenner, CAV 2009]

\(\text{preT}^*[M](C)\) is effectively regular for regular \(C\) and regular \(M\) (on trees).

Theorem 2 [Lammich, MO, Wenner, CAV 2009]

In a DPN that uses locks in a well-nested and non-reentrant fashion:

Set of tree-shaped executions having a lock-sensitive schedule is regular.

Idea of proof: Generalize Kahlon and Gupta’s acquisition histories.

Size of automaton exponential in number of locks...
Which of these trees have a lock-sensitive schedule?

No!

Yes: (0, acq X), (0, sp), (0, rel X), (1, acq X), (1, rel X)

No!
Applications

Lock-(join-)sensitive ...

- ... reachability analysis to regular sets of configurations
- ... bounded model checking
- ... DFA of bitvector problems
More Recent Work (VMCAI 2011): An Even More Regular View to Executions: Execution Trees

Joint work with:
- Thomas Gawlitza, Helmut Seidl (TU München)
- Peter Lammich, Alexander Wenner (WWU Münster)

Realised for Java analysis in Benedikt Nordhoff’s diploma thesis

Example:

\[ \begin{align*}
Call_0 &: \quad p\gamma \xrightarrow{cl} p'\gamma \\
Spawn_1 &: \quad p'\gamma \xrightarrow{sp} p\gamma > q\gamma \\
Ret_2 &: \quad p\gamma \xrightarrow{ret_2} p'' \\
Ret_3 &: \quad p''\gamma \xrightarrow{ret_3} p'' \\
Ret_4 &: \quad q\gamma \xrightarrow{ret_4} q
\end{align*} \]
Execution Tree vs. Action Tree

The DPN:

- **Call₀**: \( p\gamma \xrightarrow{cl} p'\gamma \)
- **Spawn₁**: \( p'\gamma \xrightarrow{sp} p\gamma \triangleright q\gamma \)
- **Ret₂**: \( p\gamma \xrightarrow{ret₂} p'' \)
- **Ret₃**: \( p''\gamma \xrightarrow{ret₃} p'' \)
- **Ret₄**: \( q\gamma \xrightarrow{ret₄} q \)

Action tree:
Recall: \( \text{post}^*[\text{Act}^*](p_0 \gamma_0) \) is non-regular in general.

**Observation 1:**

Set of all execution trees from given initial config., \( \text{postE}^*(p_0 \gamma_0) \), is regular!

**Observation 2:**

Set of execution trees that have a lock-sensitive schedule is regular, e.g. for:
- nested non-reentrant locking with structured form of joins
- reentrant block-structured locking (monitors, synchronized-blocks)

**Observation 3:**

Set of execution trees reaching a given regular set \( C \) of configs is regular.

**Obtain homogenous approach to, e.g., lock-sensitive reachability:**

Reg. set \( C \) is lock-sensitively reachable from start config \( p_0 \gamma_0 \) iff

\[
\text{postE}^*(p_0 \gamma_0) \cap \text{LockSensTrees} \cap \text{ExecTrees}(C) \text{ is non-empty.}
\]
(Finite) Tree-Automata

Definition

Let $\Sigma$ be a finite ranked alphabet.

(Finite bottom-up) tree automaton (over $\Sigma$):
A structure $T = (Q, Q_F, \delta)$ with:
- $Q$: finite set of states
- $Q_F \subseteq Q$: accepting states
- $\delta$: set of rules of the form:
  $f(q_1, \ldots, q_k) \rightarrow q$ with $q, q_1, \ldots, q_k \in Q$, $f \in \Sigma$ of rank $k \geq 0$

Acceptance:

a) If
- $T$ accepts trees $t_1, \ldots, t_k$ in states $q_1, \ldots, q_k$ and
- $T$ has rule $f(q_1, \ldots, q_k) \rightarrow q$
then
- $T$ accepts tree $f(t_1, \ldots, t_k)$ in state $q$

b) $T$ accepts a tree $t$
if $T$ accepts $t$ in an accepting state $q \in Q_F$
Example: (Finite) Tree-Automaton

Ranked alphabet $\Sigma$:
- Rank 0: true, false
- Rank 1: not
- Rank 2: and, or

Tree automaton:
$T = ( \{ \bot, \top \}, \{ \top \}, \delta )$ with
- $\delta :$ true $\rightarrow \top$  not($\bot$) $\rightarrow \top$
- false $\rightarrow \bot$  not($\top$) $\rightarrow \bot$
- or($\bot, \bot$) $\rightarrow \bot$  and($\bot, \bot$) $\rightarrow \bot$
- or($\bot, \top$) $\rightarrow \top$  and($\bot, \top$) $\rightarrow \bot$
- or($\top, \bot$) $\rightarrow \bot$  and($\top, \bot$) $\rightarrow \bot$
- or($\top, \top$) $\rightarrow \top$  and($\top, \top$) $\rightarrow \top$

Acceptance of example tree:
not ( and ( or (true, false), false ))
Tree Automaton for Execution Trees of a DPN

States:

\( (p, \gamma, c) \) with \( p \in P, \gamma \in \Gamma, p' \in P \cup \{N\} \)

Idea:

\( (p, \gamma, c) \) accepts tree \( T \) iff

a) \( c \in P \) and \( T \) represents terminating executions from \( p\gamma \) to \( c \), or
b) \( c = N \) and \( T \) represents non-terminating executions from \( p\gamma \)

Rules:

Nil: \([nil_{p\gamma}] \rightarrow (p, \gamma, N)\)

Base rules: \([p\gamma \xrightarrow{a} p'\gamma']((p', \gamma', c)) \rightarrow (p, \gamma, c)\)

Call rules: \([p\gamma \xrightarrow{x} p'\gamma'\gamma''((p', \gamma', p''),(p'', \gamma'', c)) \rightarrow (p, \gamma, c)\)
\([p\gamma \xrightarrow{x} p'\gamma'\gamma''((p', \gamma', N)) \rightarrow (p, \gamma, N)\)

Return rules: \([p\gamma \xrightarrow{a} p'] \rightarrow (p, \gamma, p')\)

Spawn rules: \([p\gamma \xrightarrow{a} p'\gamma' \triangleright p''\gamma'']((p', \gamma', c),(p'', \gamma'', \varnothing)) \rightarrow (p, \gamma, c)\)
Tree Automaton for Execution Trees with Lock-Sensitive Schedule

**States:** \((G,A,U)\)  
with \(A,U \subseteq \text{Locks}, G \subseteq \text{Locks} \times \text{Locks}\), accepting if \(G\) is acyclic

**Idea:** \((G,A,U)\) accepts tree \(T\)

iff

a) no lock is finally acquired more than once in \(T\),

b) \(G\) contains edge \(x \rightarrow y\) if lock \(y\) is used in \(T\) after lock \(x\) has been finally acquired,

c) \(A\) is the set of finally acquired locks, and

d) \(U\) is the set of used locks.

**Rules:**

**Nil:**  
\([\text{nil}_\gamma'] \rightarrow (\emptyset,\emptyset,\emptyset)\)

**Base rules:**  
\([p\gamma \xrightarrow{a} p'\gamma'](G,A,U) \rightarrow (G,A,U)\)

**Call rules:**  
\([p\gamma \xrightarrow{X} p'\gamma'\gamma''](G,A,U),(G',A',U')) \rightarrow\)

if \(A \cap A' = \emptyset\)

\([p\gamma \xrightarrow{X} p'\gamma'\gamma''](G,A,U) \rightarrow (G \cup X \times A, A \cup X, U)\)

if \(A \cap X = \emptyset\)

**Return rules:**  
\([p\gamma \xrightarrow{a} p'] \rightarrow (\emptyset,\emptyset,\emptyset)\)

**Spawn rules:**  
\([p\gamma \xrightarrow{a} p'\gamma' \triangleright p''\gamma''](G,A,U),(G',A',U') \rightarrow (G \cup G', A \cup A', U \cup U')\)

if \(A \cap A' = \emptyset\)
Realization for Java

Diploma thesis of Benedikt Nordhoff

Uses:
- WALA from IBM: T.J. Watson Libraries for Analysis
- XSB: A Prolog-like system with tabulating evaluation

Identifies object references that can be used as locks

For practicality:
- Pre-analysis of WALA flow graph and (massive) pruning
- Modular reformulation of automata-based analysis
- Clever evaluation strategy for tree automata construction

Experimental applications:
- Monitor-sensitive data-race analyzer for Java
- RS3 context: Improve PDG-based IFC analysis of concurrent Java
Java Data-Race Finder: Screenshot 1

```java
package bnord.examples.datarace;
import bnord.examples.Lock;
public class BSP03 extends Thread {
    static long x;
    public static void main(String[] args) {
        synchronized (lock1) {
            thread.start();
            x = 42;
        }
    }
    public void run() {
        x = 17;
    }
    static Lock lock1 = new Lock();
    static BSP03 thread = new BSP03();
}
```

Race found!

There might be a race in your program
See result view.

Overall Result: Race free: 2/3 Possible race: 1/3
- Field: bnord.examples.datarace.BSP03.lock1 of type: bnord.examples.Lock
- Field: bnord.examples.datarace.BSP03.thread of type: bnord.examples.datarace.BSP03
- Field: bnord.examples.datarace.BSP03.x of type: J

Data race result
Java Data-Race Finder: Screenshot 2

```java
package bnord.examples.datarace;

import bnord.examples.Lock;

public class BSP03 extends Thread {
    static long x;

    public static void main(String[] args) {
        synchronized (lock1) {
            thread.start();
            x = 42;
        }
        x = 17;

        static Lock lock1 = new Lock();
        static BSP03 thread = new BSP03();
    }

    public void run() {
    }

    Data race result

    Overall Result: 
    Race free: 2/3  Possible race: 1/3

    Field: bnord.examples.datarace.BSP03.thread of type: bnord.examples.datarace.BSP03
    Field: bnord.examples.datarace.BSP03.x of type: j
    Field on static object: <Application, Lbnord/examples/datarace/BSP03>
    ```
Java Data-Race Finder: Screenshot 3

package bndon.examples.datarace;
import bndon.examples.lock;
public class BSP03 extends Thread {
    static long x;
    public static void main(String[] args) {
        synchronized (lock1) {
            thread.start();
            x = 42;
        }
    }
    public void run() {
        synchronized (lock1) {
            x = 17;
        }
    }
    static Lock lock1 = new Lock();
}

No race found

There is no race in your program

Overall Result: Race free: 3/3 Possible race: 0/3
Field: bndon.examples.datarace.BSP03.lock1 of type: bndon.examples.Lock
Field: bndon.examples.datarace.BSP03.thread of type: bndon.examples.datarace.BSP03
Field: bndon.examples.datarace.BSP03.x of type: J
Conclusion

- Lock-join-sensitive analysis using automata
- Finite state + recursion + thread creation + locks + joins
- Experimental applications
- Trees are better than words
- Keeping more structure in the trees is even better
Thank you!