Automata-based Analysis of Threaded Programs

Markus Müller-Olm
Westfälische Wilhelms-Universität Münster, Germany

Workshop HOMC + CDPS
Singapore, September 19-23, 2016

What This Talk is About

Last decades:

Tremendous progress on automatic analysis of infinite-state systems

One line of research:

Automata-based methods / regular model-checking

This talk:

Automata-based analysis of recursive multi-threaded programs synchronizing via locks/monitors

Communicating	Synchronization via lock
Distributed	Parallelism, no globally shared state
Parameterised	Dynamic thread creation
Systems	Networks of pushdown systems

Dynamic Pushdown Networks (DPNs)

- DPN: An automata-based model for multi-threaded recursive programs

- A natural extension of push-down systems:

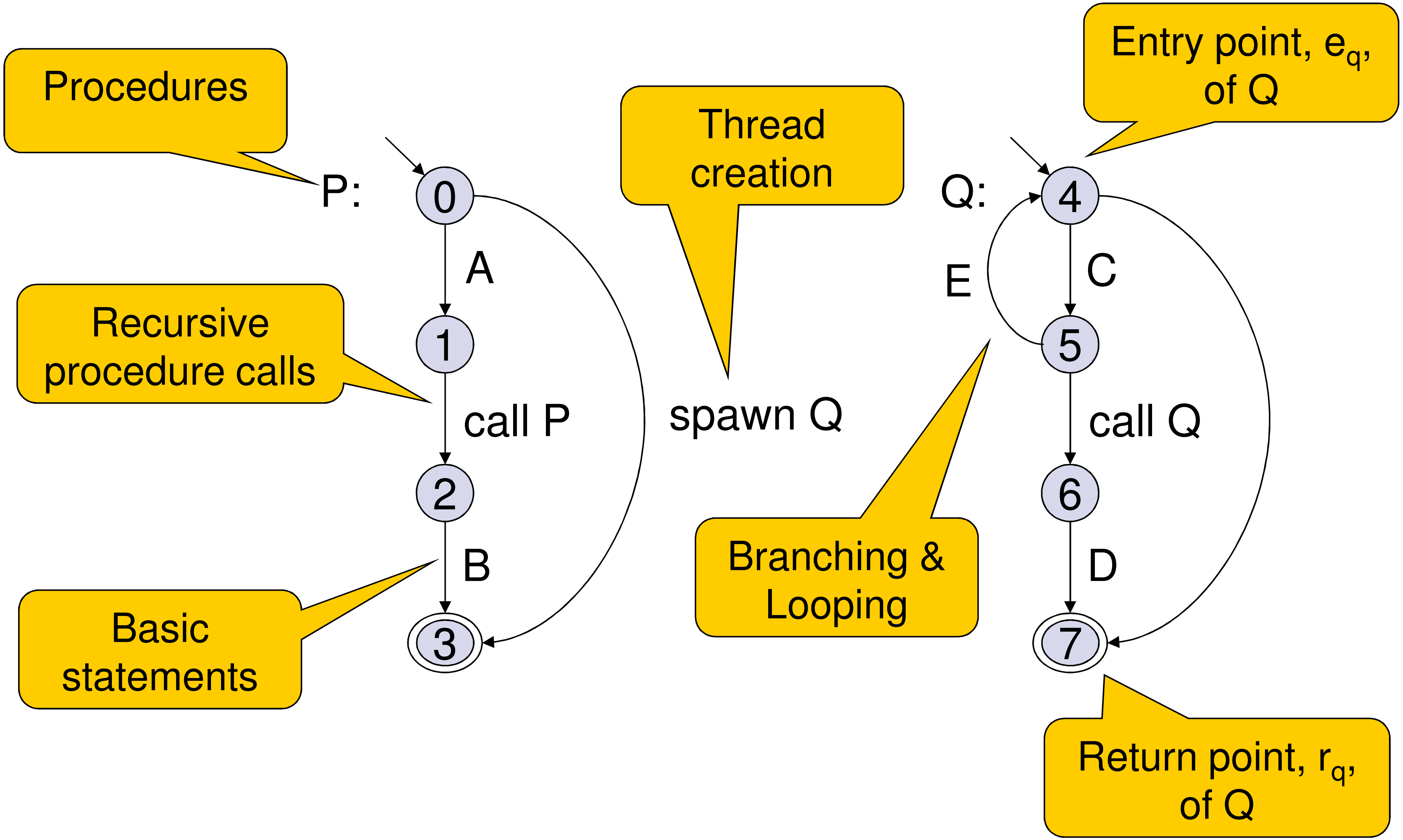
$$p\gamma \xrightarrow{a} qw \qquad\qquad |w| \leq 2$$

$$p\gamma \xrightarrow{a} qw \rhd q'\gamma' \qquad\qquad |w| = 1$$

- Generic methods for lock-sensitive iterated reachability analysis based on word- and tree-automata

- Applied for data-race and information-flow analysis of Java

Recursive Programs with Thread Creation



+ finite-state abstraction of (thread-local) global and local variables

Modelling Program Behavior with DPNs

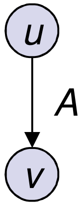
à la [Esparza/Knoop, FOSSACS'99]

abstraction of global state

abstraction of local state

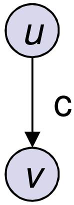
current control point

for basic edge e:



$$g\langle l,u\rangle \xrightarrow{e} g'\langle l',v\rangle, \qquad \text{if } \big((g,l),(g',l')\big)\in \mathit{Abstr(A)}$$

for call edge e:

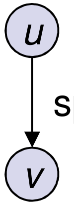


$$g\langle l,u\rangle \xrightarrow{e} g\langle l_{\text{init}},e_P\rangle\langle l,v\rangle$$

for return point of each procedure

$$g\langle l,r_P\rangle \xrightarrow{ret} g$$

for spawn edge e:



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

Execution Semantics of DPNs on Word-shaped Configurations

A configuration of a DPN is a word in (PΓ*)⁺:

$$p_1 w_1 p_2 w_2 \cdots p_k w_k \qquad\qquad (\text{with } p_i \in P, w_i \in \Gamma^*, k > 0)$$

... an infinite state space

The transition relation of a DPN:

$$\frac{\left(p\gamma \xrightarrow{a} qw\right) \in \Delta}{u\,p\,\gamma v \xrightarrow{a} u\,q\,wv} \qquad\qquad \frac{\left(p\gamma \xrightarrow{a} qw \rhd q'w'\right) \in \Delta}{u\,p\,\gamma v \xrightarrow{a} u\,q'w'qwv}$$

Example

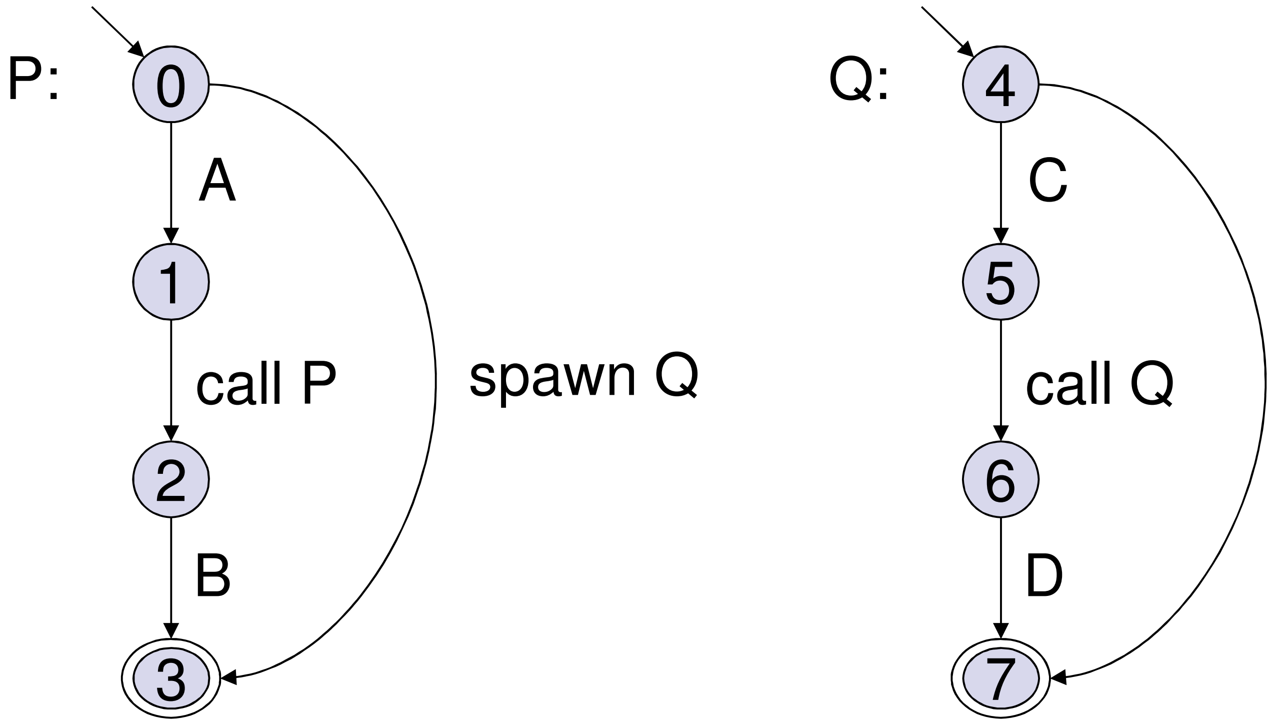
A DPN: $p\gamma \xrightarrow{spawn} p\gamma\gamma \triangleright q_0\gamma$ $\quad q_0\gamma \xrightarrow{hello} q_1\gamma$

$q_1\gamma \xrightarrow{world} q_2$

One of its many execution sequences:

$$p\gamma \xrightarrow{spawn} q_0\gamma p\gamma\gamma \xrightarrow{spawn} q_0\gamma q_0\gamma p\gamma\gamma\gamma \xrightarrow{hello} q_1\gamma q_0\gamma p\gamma\gamma\gamma \xrightarrow{world} q_2 q_0\gamma p\gamma\gamma\gamma \xrightarrow{hello} q_2 q_1\gamma p\gamma\gamma\gamma$$

Spawns are Fundamentally Different from Parallel Procedure Calls



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

$$\begin{aligned} \mathrm{pre}^*[L](C) &:= \{c \mid \exists d \in C, w \in L : c \xrightarrow{w}{}^* d\} \\ \mathrm{post}^*[L](C) &:= \{d \mid \exists c \in C, w \in L : c \xrightarrow{w}{}^* d\} \end{aligned}$$

Forward-Reachability

☹ 1) post*[Act*](C) is in general non-regular for regular C.

😐 2) post*[A*](C) is effectiv. context-free for context-free C and A $\subseteq$ Act (in polytime)

☹ 3) Membership in post*[L](C) is in general undecidable for regular L.

Backward-Reachability

☺ 1) pre*[A*](C) is effectively regular for regular C and A $\subseteq$ Act (in polytime).

☹ 2) Membership in pre*[L](C) is in general undecidable for regular L.

Single Steps

☺ 1) pre*[A](C) and post*[A](C) are effectively regular for regular C and A $\subseteq$ Act (in polyn. time).

Example: Backward Reachability Analysis for DPNs

Consider a DPN with just the rule

$$p\gamma \xrightarrow{spawn} p\gamma\gamma \triangleright q\gamma$$

and the infinite set of states

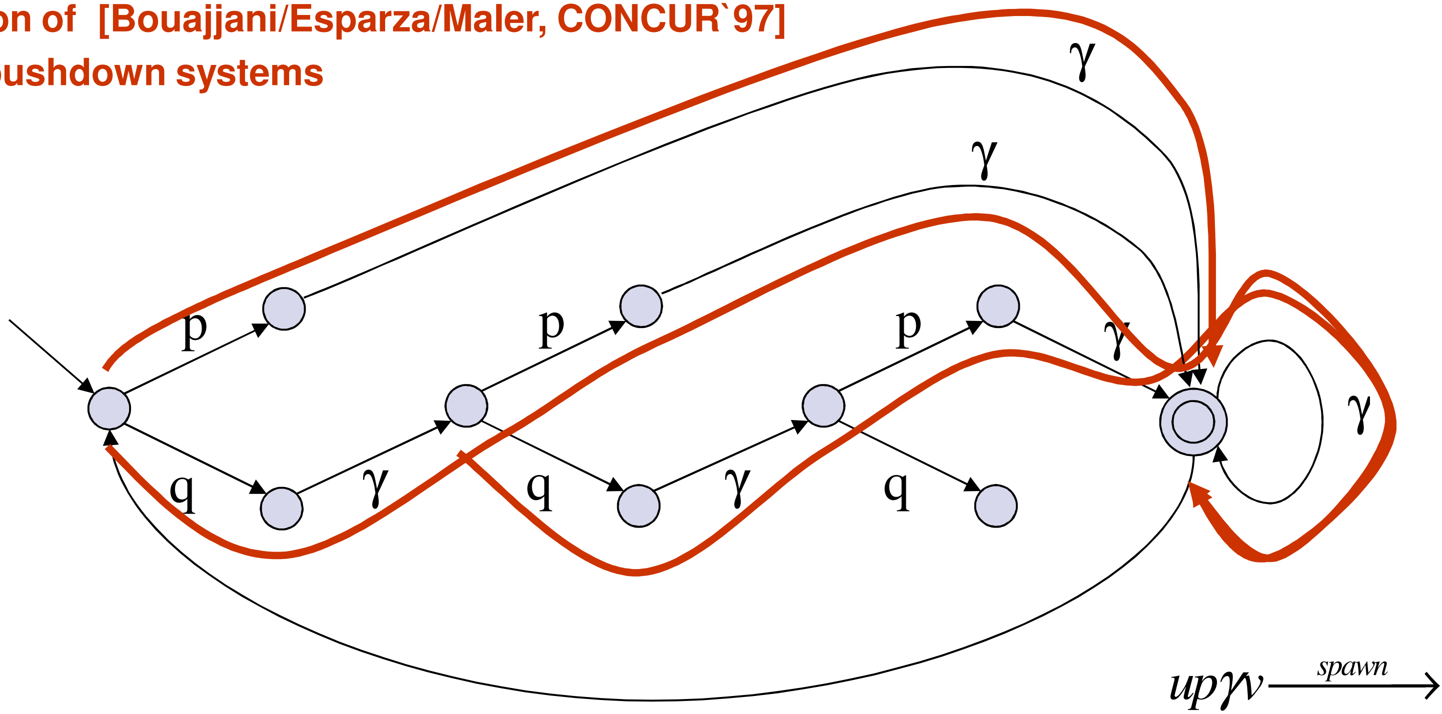
$$\mathrm{Bad} = \left(q\gamma q\gamma p\gamma^{+}\right)^{+} = L(A)$$

Analysis problem: can Bad be reached from pγ ?

Example: Backward Reachability Analysis for DPNs

1. Step: Saturate automaton for Bad with the DPN rule: $p\gamma \xrightarrow{spawn} p\gamma\gamma \triangleright q\gamma$

Generalization of [Bouajjani/Esparza/Maler, CONCUR`97] method for pushdown systems



$up\gamma v \xrightarrow{spawn} uq\gamma p\gamma\gamma v$

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

2. Step: Check, whether pγ is accepted by A_{pre*} or not

Result: Bad is reachable from pγ, as A_{pre*} accepts pγ !

Some Applications of pre*-Computations with unrestricted L (i.e. L = Act*)

Reachability of regular sets of configurations, e,g. conflict analysis, data race analysis etc.

Set Bad of configurations is reachable from initial configuration $p_0\gamma_0$

iff

$p_0\gamma_0 \in pre^*[Act^*](Bad)$

used in JMoped of Schwoon/Esparza

Bounded model checking

By iterated pre*-computations alternating with single steps corresponding to synchronizations/communications

à la [Esparza/Knoop, FOSSACS'99]

Bit-vector data-flow analysis problems

Variable x is live at program point u

iff $\quad g_{init}\left\langle l_{init}, e_{Main}\right\rangle \in pre^*[Act^*]\big(At_u \cap pre^*[NonDef_x^*]\big(pre^*[Use_x](Conf)\big)\big)$

Lock-/Monitor-sensitive Analysis

- Assume finite set of locks (or monitors)

- Have acquire- and release actions
  - acq L, rel $L \in$ Act f.a. locks L

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

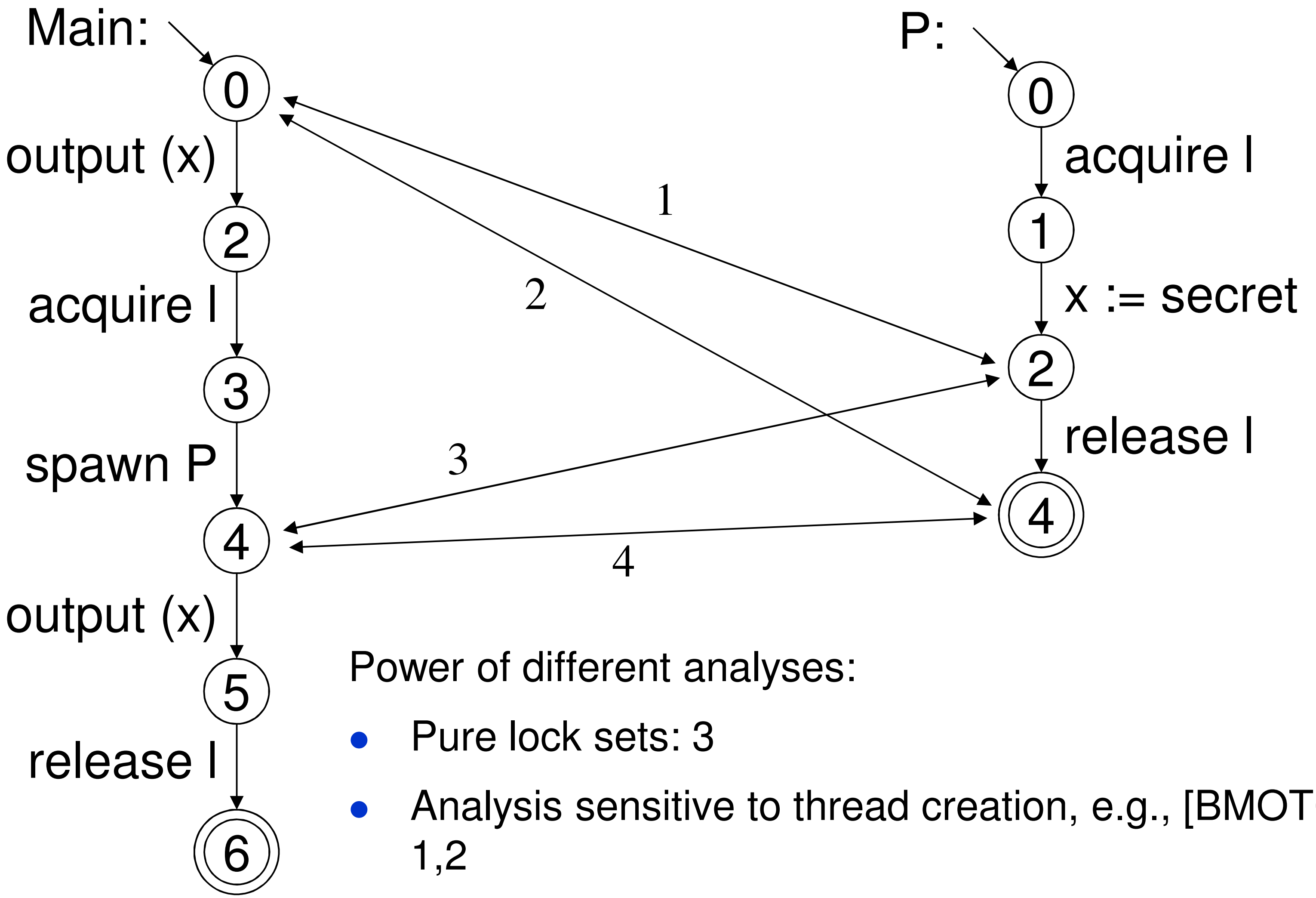
- The Goal: lock-sensitive analysis

A Multi-Threaded Java Program

```
class MyThread extends Thread {
    private Objekt l;
    private int secret = 42;
    private int x = 0;

    public MyThread (Object l)  {
            this.l = l;
    }
    public void run() {
            synchronized (l) {
                    x= secret;
            }
    }
    public static void main (...) {
  ...  // see right column
    }
}
```

```
public static void main (String[] args) {
    Object l = new Object();
    MyThread t = new MyThread(l);
    System.out.println(t.x);
    synchronized (l) {
        t.start ();
        System.out.println(t.x);
    }
}
```

Lock-sensitive Analysis



Power of different analyses:

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

Of course, we also treat branching, loops, recursion !

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

Must-Lockset computed at X: { L1 }

Must-Lockset computed at Y: { L2 }

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

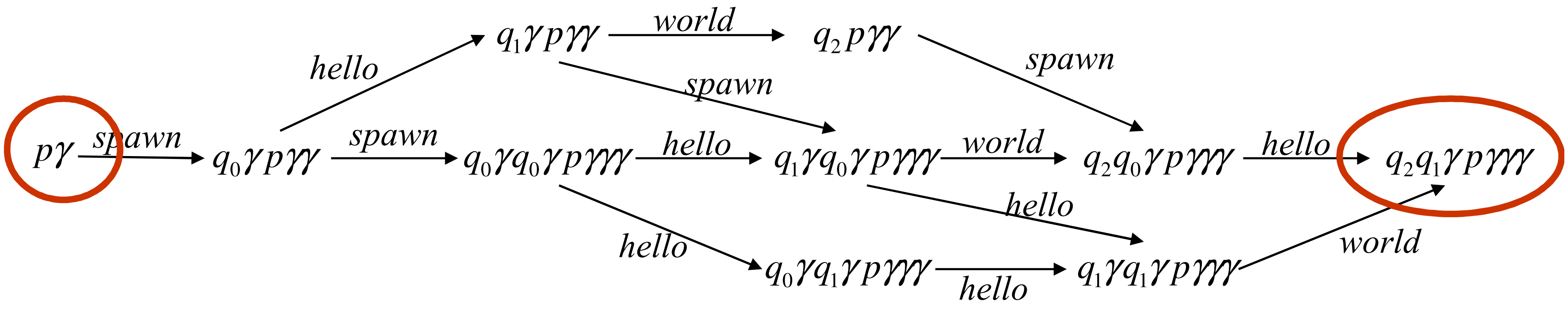
Nevertheless, X and Y are not reachable simultaneously !

A Tree-Based View of Executions: Action Trees

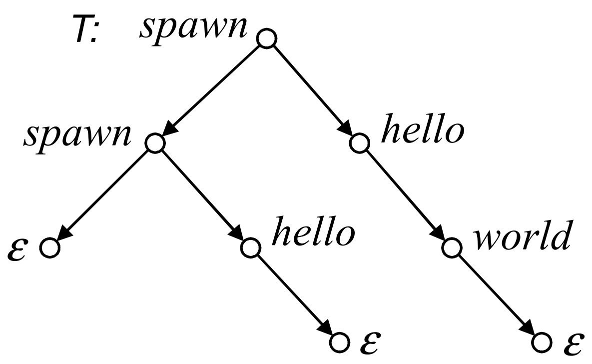
A DPN: $p\gamma \xrightarrow{spawn} p\gamma\gamma \triangleright q_0\gamma$ $q_0\gamma \xrightarrow{hello} q_1\gamma$

$q_1\gamma \xrightarrow{world} q_2$

Execution sequences:



Action tree:



We write: $p\gamma \xrightarrow{T}{}^{*} q_2 q_1\gamma p\gamma\gamma\gamma$

A Tree-Based View of Executions

Definition

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

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

Recall:

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

Theorem 1 [Lammich, MO, Wenner, CAV 2009]

preT*[M](C) is effectively regular for regular C and regular M (on trees).

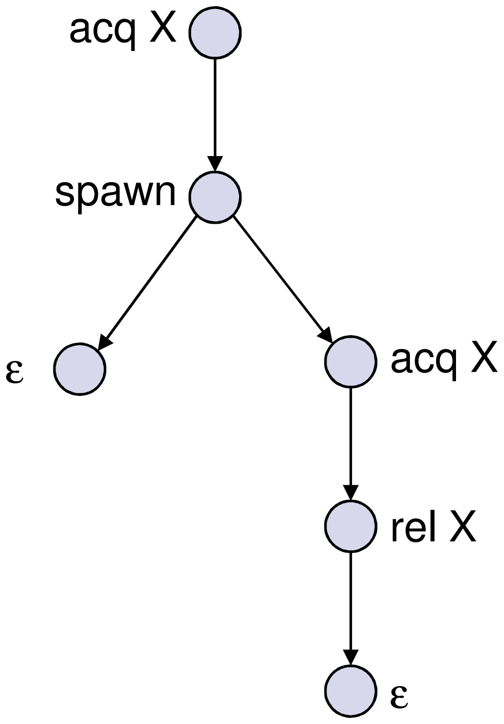
Theorem 2 [Lammich, MO, Wenner, CAV 2009]

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.

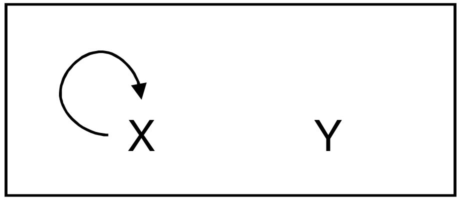
Idea of proof: Generalize Kahlon and Gupta‘s acquisition histories.

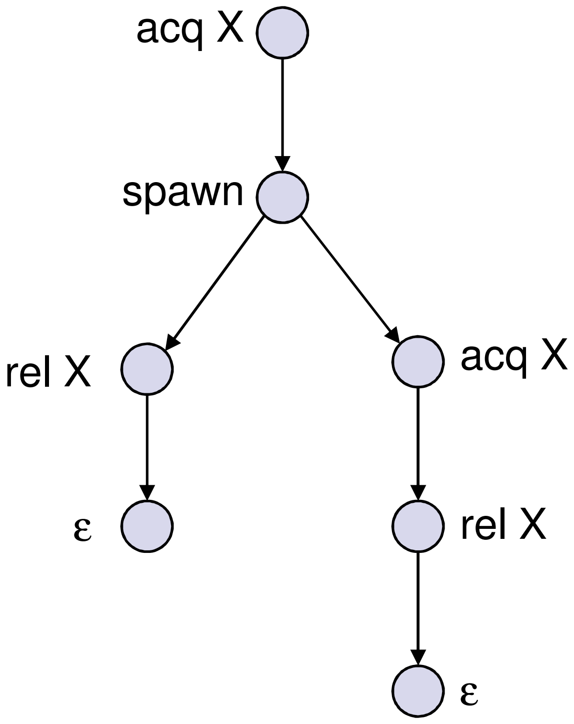
Size of automaton exponential in number of locks...

Which of these action trees have a lock-sensitive schedule?

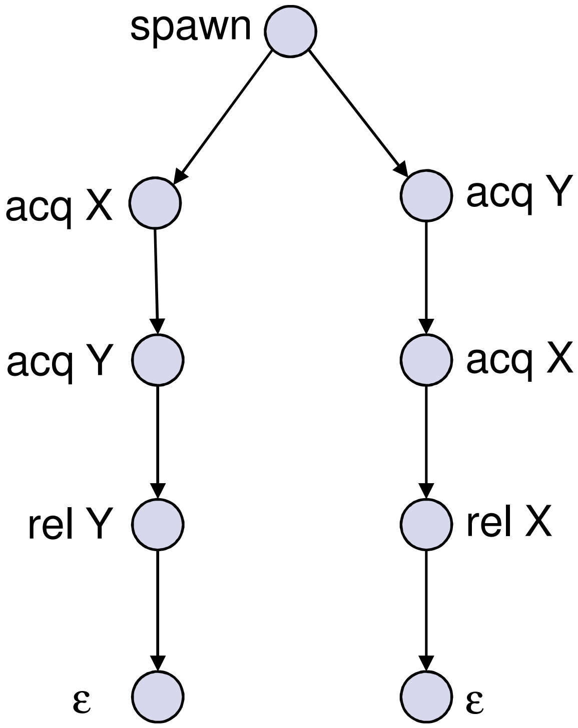


No!

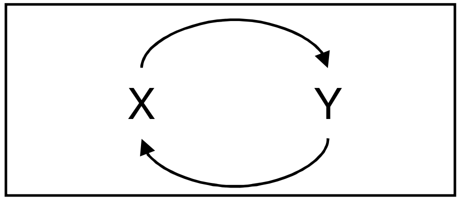




Yes: (0,acq X),(0,sp),(0,rel X),
(1,acq X),(1,rel X)



No!



An Even More Regular View to Executions: Execution Trees

Joint work (VMCAI‘11) with:

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

Realised for Java analysis: Benedikt Nordhoff‘s diploma thesis

Example:

$$Call: \quad p\gamma \xrightarrow{call} p'\gamma\gamma$$

$$Spawn: \quad p'\gamma \xrightarrow{spawn} p\gamma \triangleright q\gamma$$

$$NoMore: \quad p\gamma \xrightarrow{no\_more} p''\gamma$$

$$Return: \quad p''\gamma \xrightarrow{return} p''$$

$$Hello: \quad q\gamma \xrightarrow{hello} q$$

An Even More Regular View to Executions

The DPN:

$Call: \quad p\gamma \xrightarrow{call} p'\gamma\gamma$

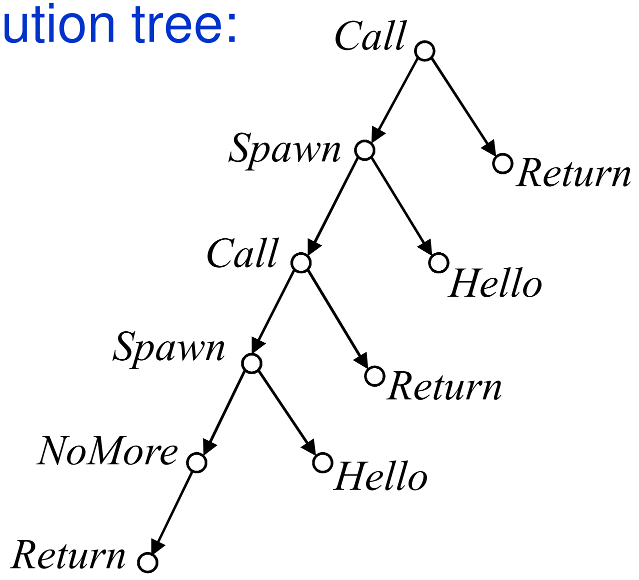
$Spawn: \quad p'\gamma \xrightarrow{spawn} p\gamma \triangleright q\gamma$

$NoMore: \quad p\gamma \xrightarrow{no\_more} p''\gamma$

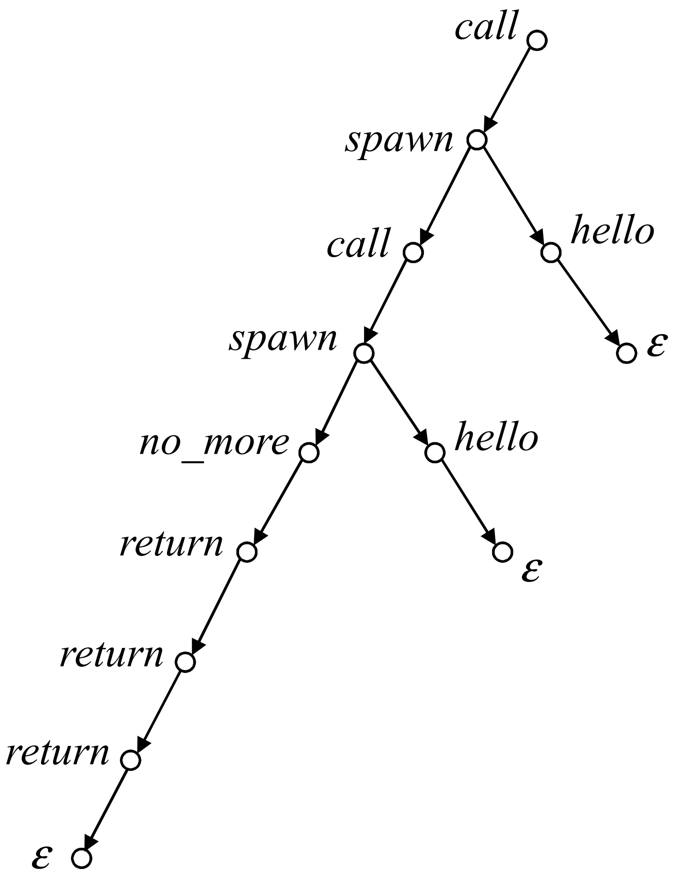
$Return: \quad p''\gamma \xrightarrow{return} p''$

$Hello: \quad q\gamma \xrightarrow{hello} q$

Execution tree:



Action tree:



Execution Trees

Recall: post*[Act*]($p_0\gamma_0$) is non-regular in general.

Observation 1:

Set of all execution trees from given initial config., 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 (even 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

postE*($p_0\gamma_0$) $\cap$ LockSensTrees $\cap$ ExecTrees(C) is non-empty.

Applications

Lock-join-sensitive ...

- ... reachability analysis to regular sets of configurations, e.g. conflict analysis, data race analysis etc.
- ... bounded model checking
- ... DFA of bitvector problems

Realization for Java

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

- Object creation sites visited at most once
- Experiments with Kidd et. al.'s random isolation technique

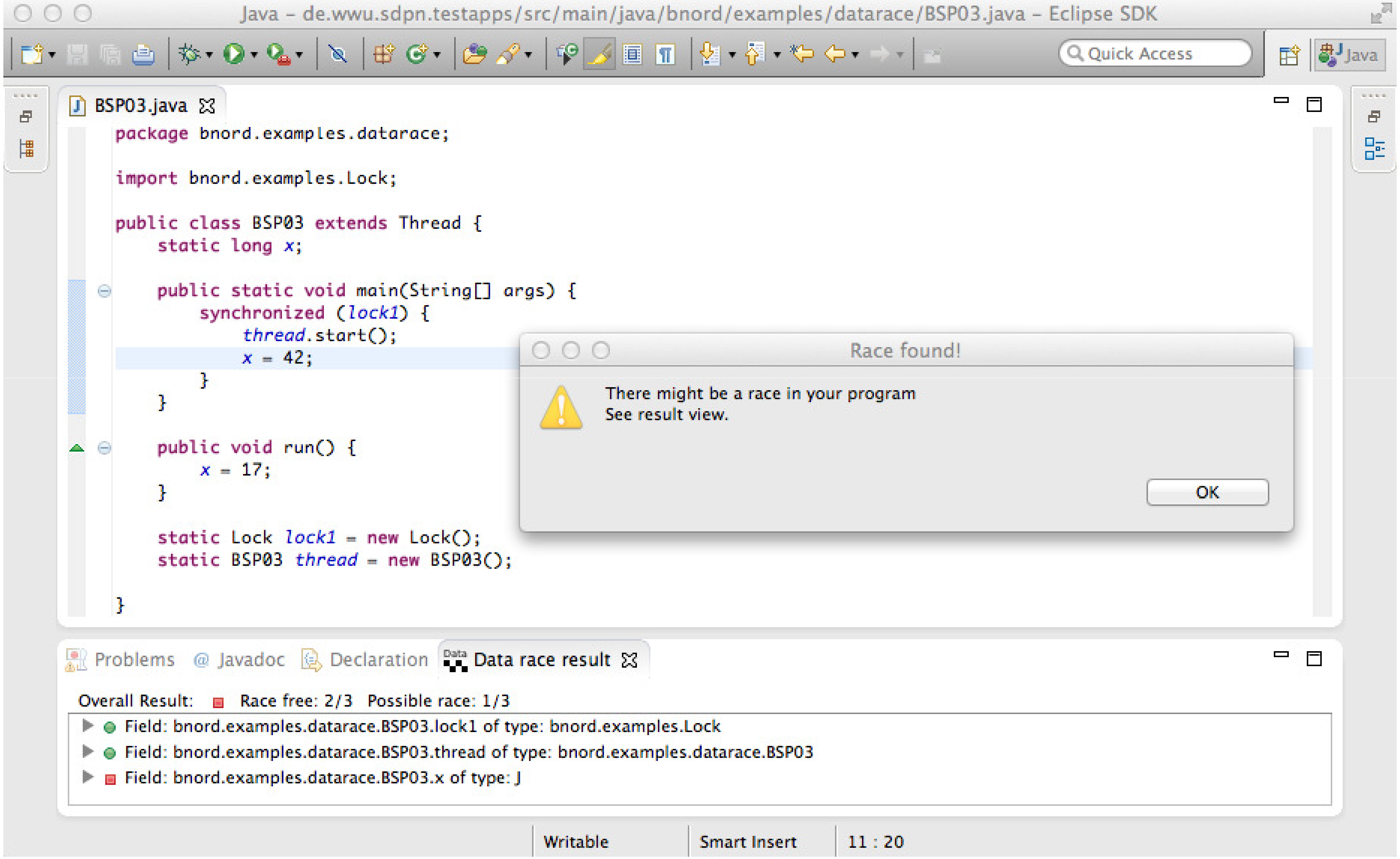
For practicality:

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

Experimental applications:

- Lock-sensitive data-race analyzer for Java
- With KIT: Improve PDG-based IFC analysis of concurrent Java programs

Java Data-Race Finder: Screenshot 1



Java – de.wwu.sdpn.testapps/src/main/java/bnord/examples/datarace/BSP03.java – Eclipse SDK

```
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();

}
```

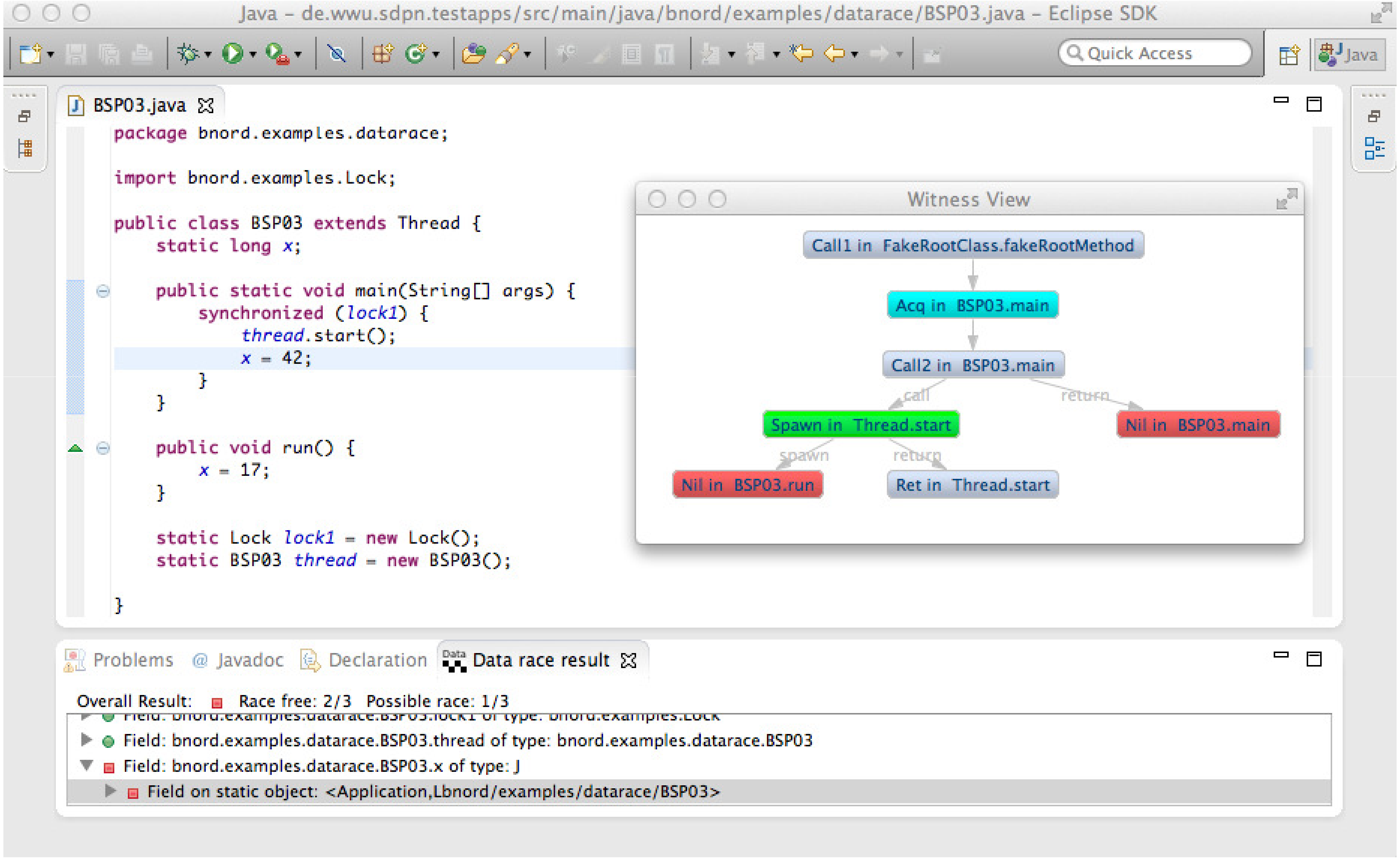
Race found!

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

Java Data-Race Finder: Screenshot 2

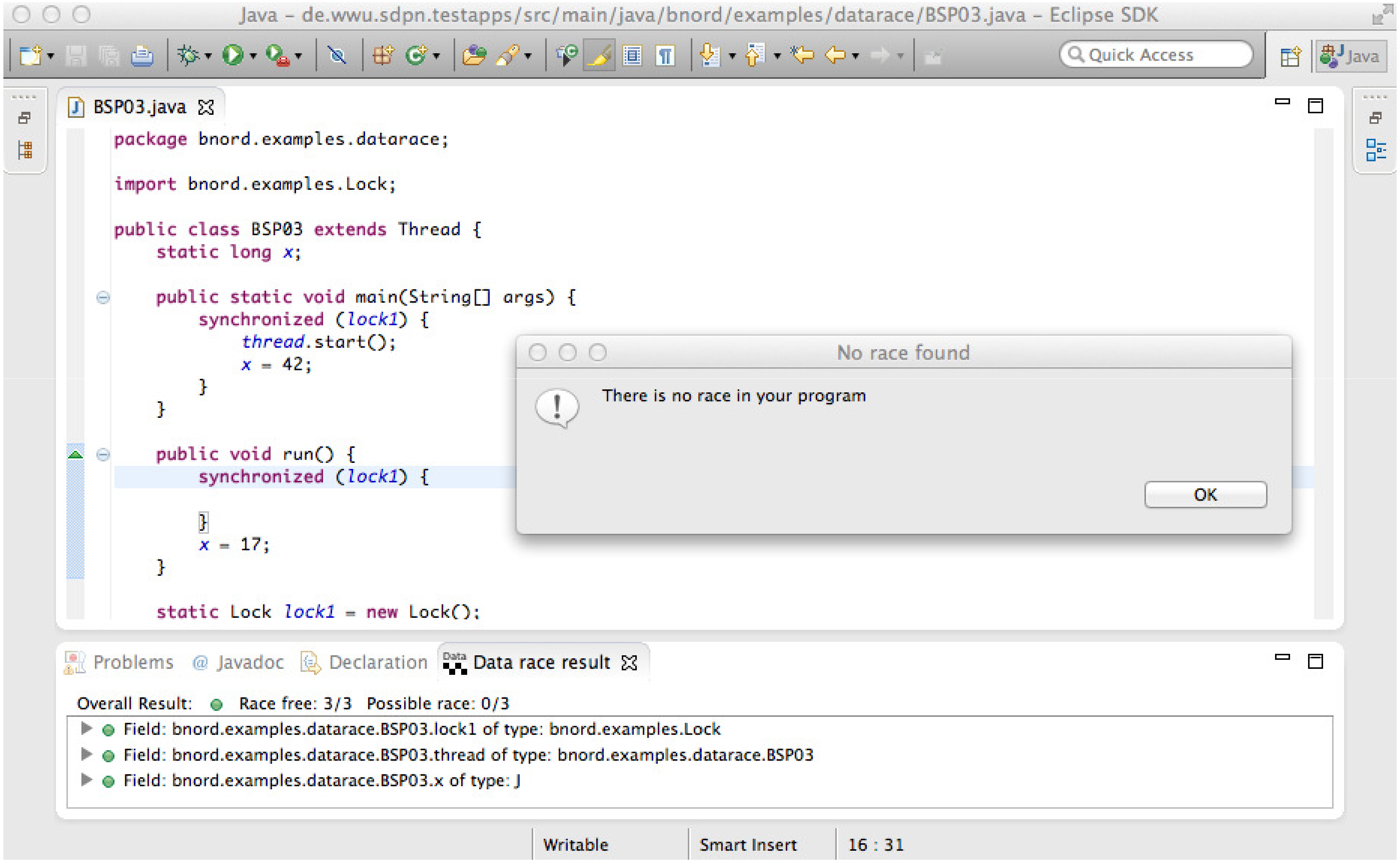


```
package bnord.examples.datarace;

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();

}
```

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
  - ▶ 🟥 Field on static object: <Application,Lbnord/examples/datarace/BSP03>

Java Data-Race Finder: Screenshot 3



Java – de.wwu.sdpn.testapps/src/main/java/bnord/examples/datarace/BSP03.java – Eclipse SDK

```
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() {
        synchronized (lock1) {

        }
        x = 17;
    }

    static Lock lock1 = new Lock();
```

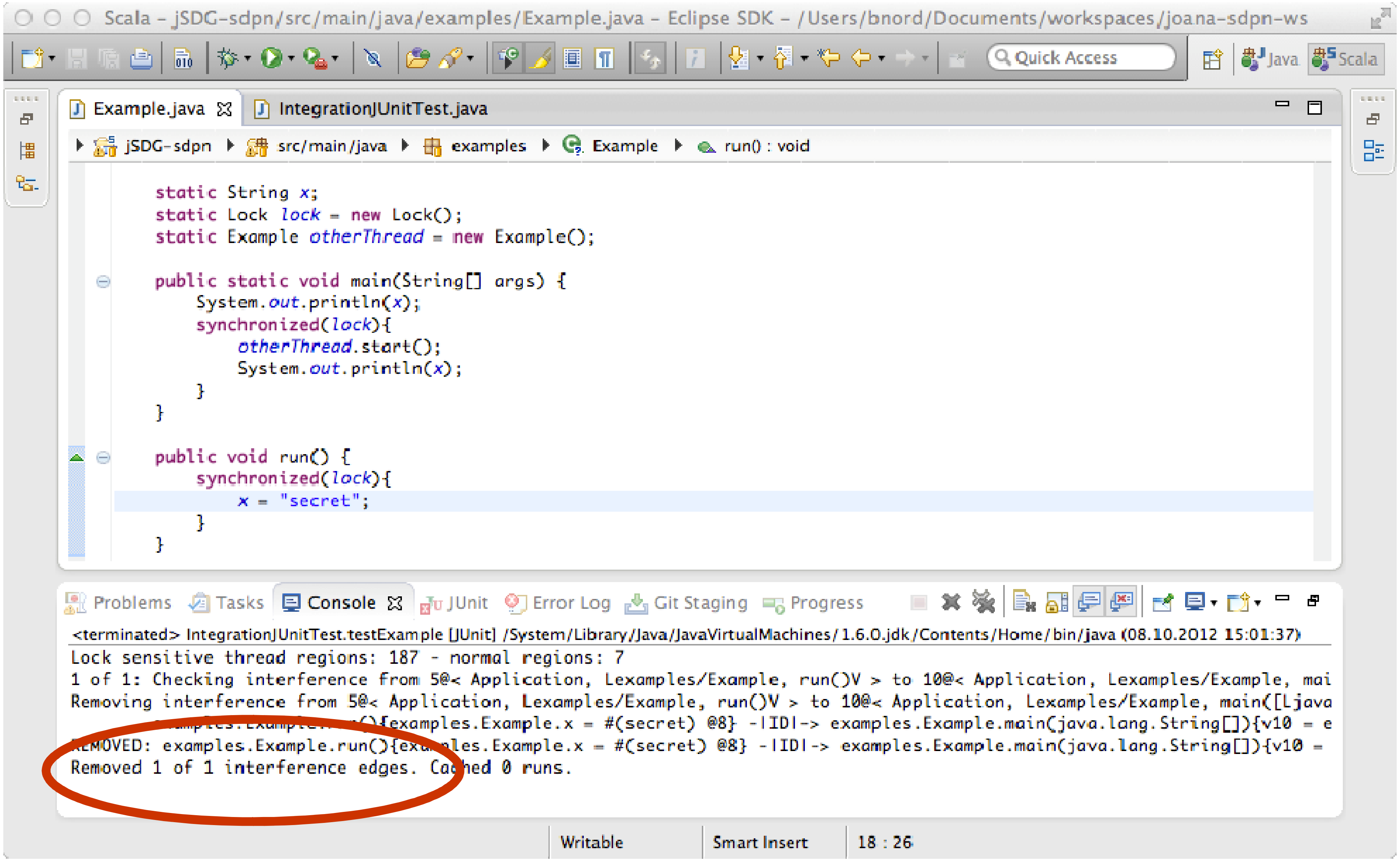
No race found

There is no race in your program

Overall Result: ● Race free: 3/3 Possible race: 0/3

- ● Field: 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

Experimental Integration with Joana: Screenshot



```java
static String x;
static Lock lock = new Lock();
static Example otherThread = new Example();

public static void main(String[] args) {
    System.out.println(x);
    synchronized(lock){
        otherThread.start();
        System.out.println(x);
    }
}

public void run() {
    synchronized(lock){
        x = "secret";
    }
}
```

```
<terminated> IntegrationJUnitTest.testExample [JUnit] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home/bin/java (08.10.2012 15:01:37)
Lock sensitive thread regions: 187 - normal regions: 7
1 of 1: Checking interference from 5@< Application, Lexamples/Example, run()V > to 10@< Application, Lexamples/Example, mai
Removing interference from 5@< Application, Lexamples/Example, run()V > to 10@< Application, Lexamples/Example, main([Ljava
    examples.Example.run(){examples.Example.x = #(secret) @8} -|ID|-> examples.Example.main(java.lang.String[]){v10 = e
REMOVED: examples.Example.run(){examples.Example.x = #(secret) @8} -|ID|-> examples.Example.main(java.lang.String[]){v10 =
Removed 1 of 1 interference edges. Cached 0 runs.
```

Conclusion

- Lock-join-sensitive analysis using automata
- Finite state + recursion + thread creation + locks + joins
- Experimental applications for Java

- SAS‘13: Extension to „contextual locking“
- LOPSTR‘15: Application to information-flow analysis
- Ongoing work: Unbounded number of locks

Thank you !