

Automata-based Analysis of Threaded Programs

Markus Müller-Olm
Westfälische Wilhelms-Universität Münster, Germany

IFIP WG 2.2 Meeting
Lucca, September 21-23, 2015

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

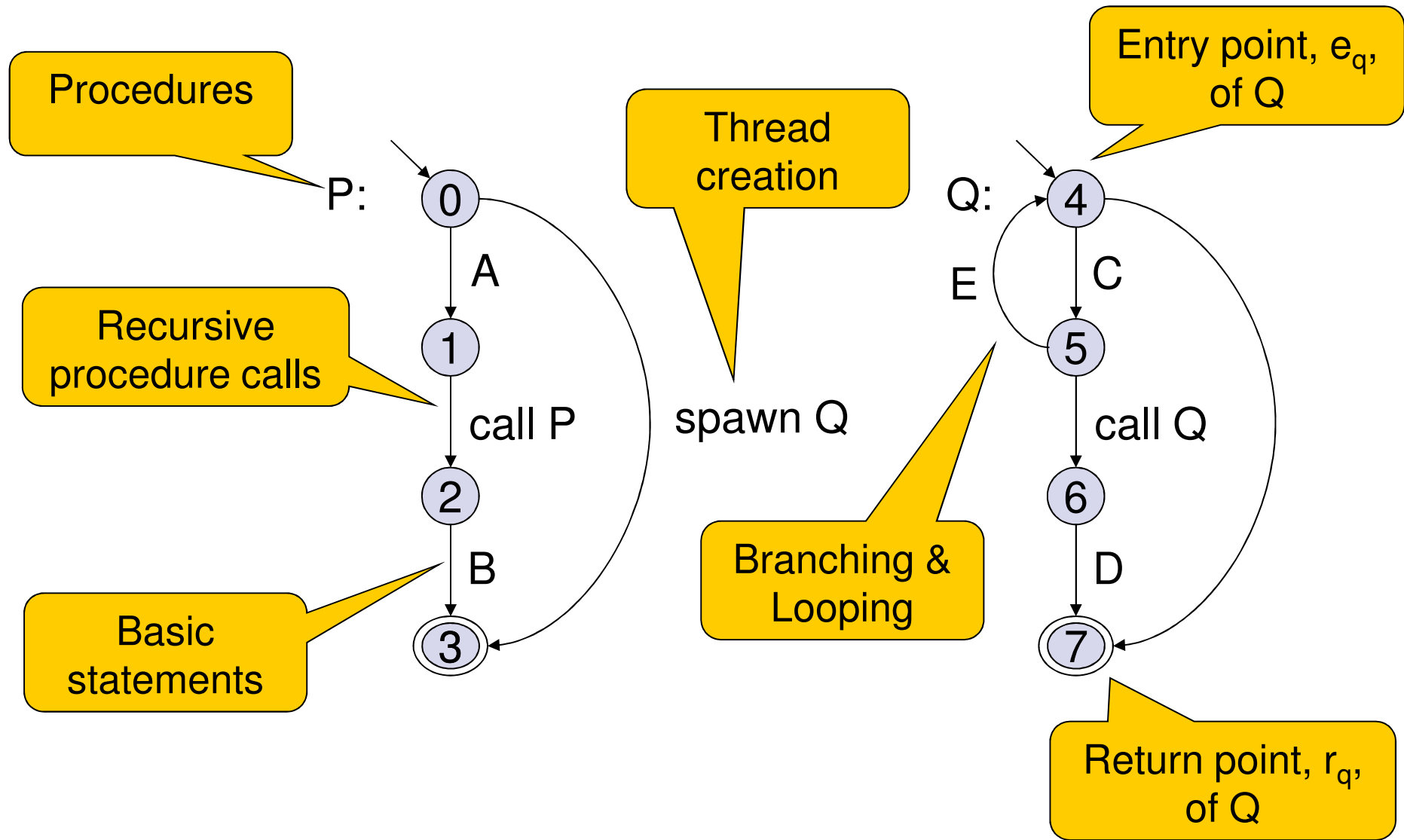
Dynamic Pushdown Networks (DPNs)

- DPN: An automata-based model for multi-threaded recursive programs
- A natural extension of push-down systems:

$$p\gamma \xrightarrow{a} qw \quad [\triangleright q'w'] \quad |w| \leq 2, |w'| = 1$$

- Generic methods for lock-sensitive iterated reachability analysis based on word- and tree-automata
- Applied for data-race and information-flow analysis of Java

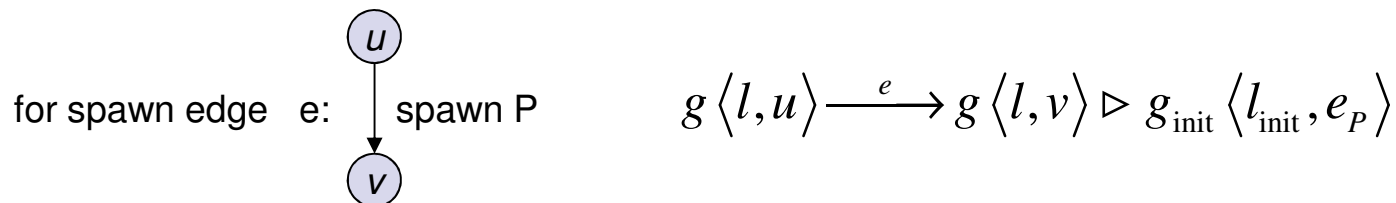
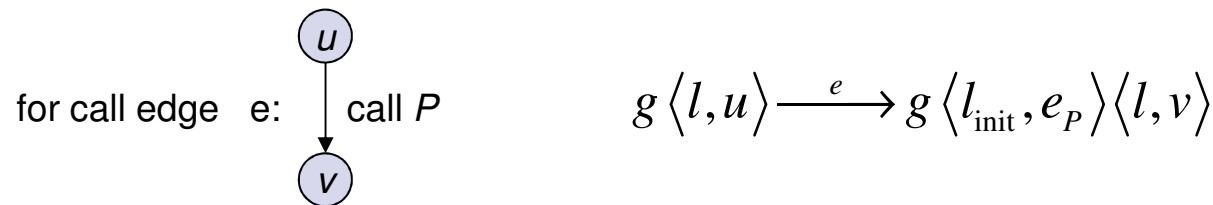
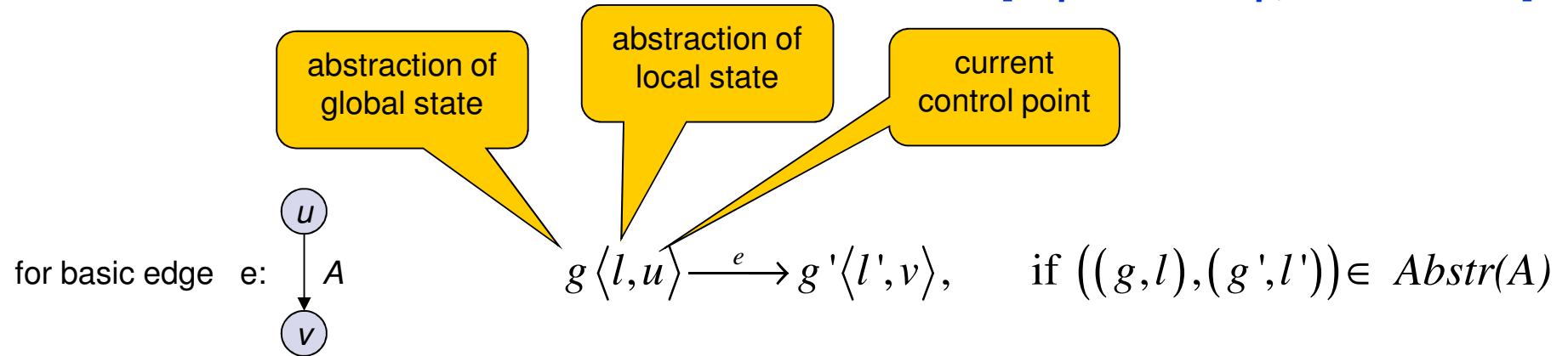
Recursive Programs with Thread Creation



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

Modelling Program Behavior with DPNs

à la [Esparza/Knoop, FOSSACS'99]



Execution Semantics of DPNs on Word-shaped Configurations

A **configuration** of a DPN is a word in $(P\Gamma^*)^+$:

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

... an infinite state space

The transition relation of a DPN:

$$(p\gamma \xrightarrow{a} qw) \in \Delta: \quad u p \gamma v \xrightarrow{a} u q w v$$

$$(p\gamma \xrightarrow{a} qw \triangleright q'w') \in \Delta: \quad u p \gamma v \xrightarrow{a} u q'w'qwv$$

Example

A DPN: $p\gamma \xrightarrow{\text{spawn}} p\mathcal{N} \triangleright q_0\gamma$ $q_0\gamma \xrightarrow{\text{hello}} q_1\gamma$
 $q_1\gamma \xrightarrow{\text{world}} q_2$

One of its many execution sequences:

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

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) $\text{post}^*[A^*](C)$ is effectiv. context-free for context-free C and $A \subseteq \text{Act}$ (in polytime)
-  3) Membership in $\text{post}^*[L](C)$ is in general undecidable for regular L .

Backward-Reachability

-  1) $\text{pre}^*[A^*](C)$ is effectively regular for regular C and $A \subseteq \text{Act}$ (in polytime).
-  2) Membership in $\text{pre}^*[L](C)$ is in general undecidable for regular L .

Single Steps

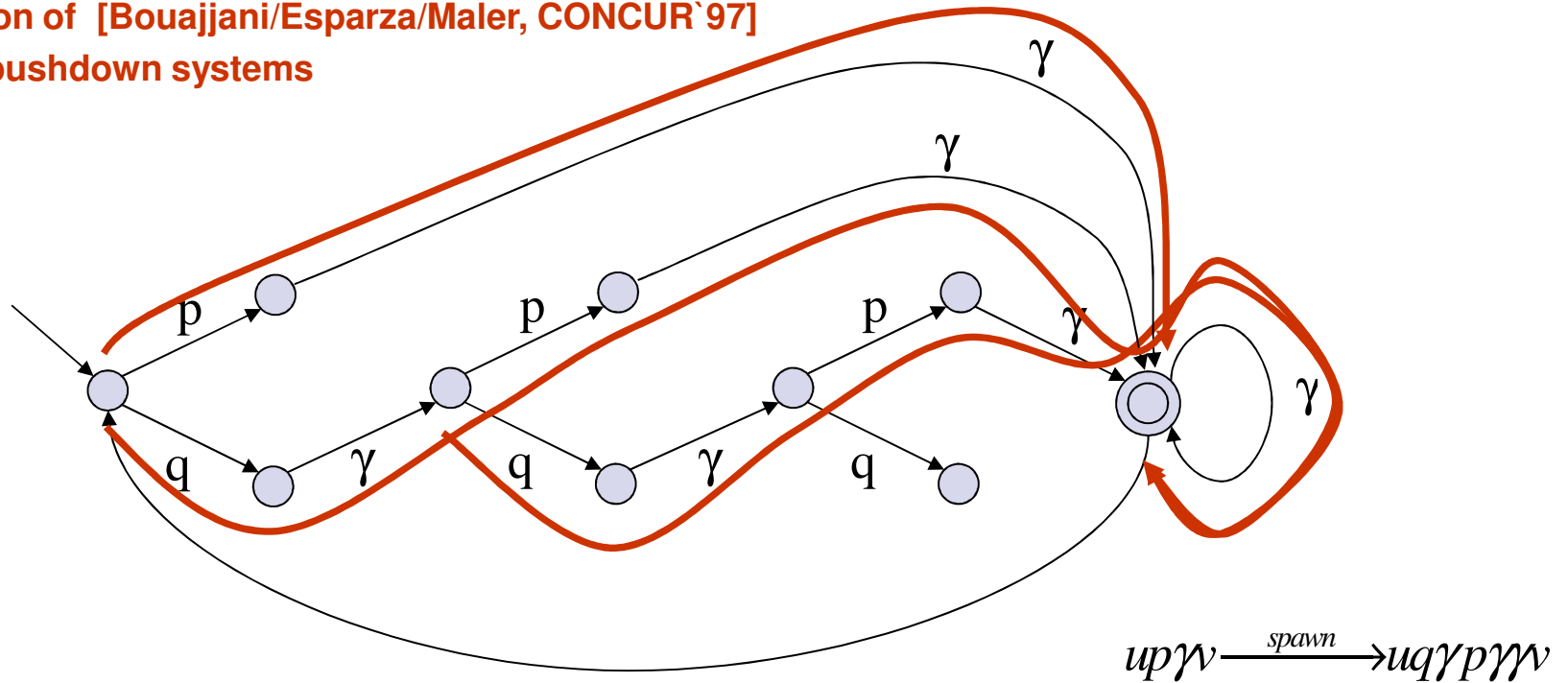
-  1) $\text{pre}^*[A](C)$ and $\text{post}^*[A](C)$ are effectively regular for regular C and $A \subseteq \text{Act}$ (in polyn. time).

Example: Backward Reachability Analysis for DPNs

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

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

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



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

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 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

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

Lock-/Monitor-sensitive Analysis

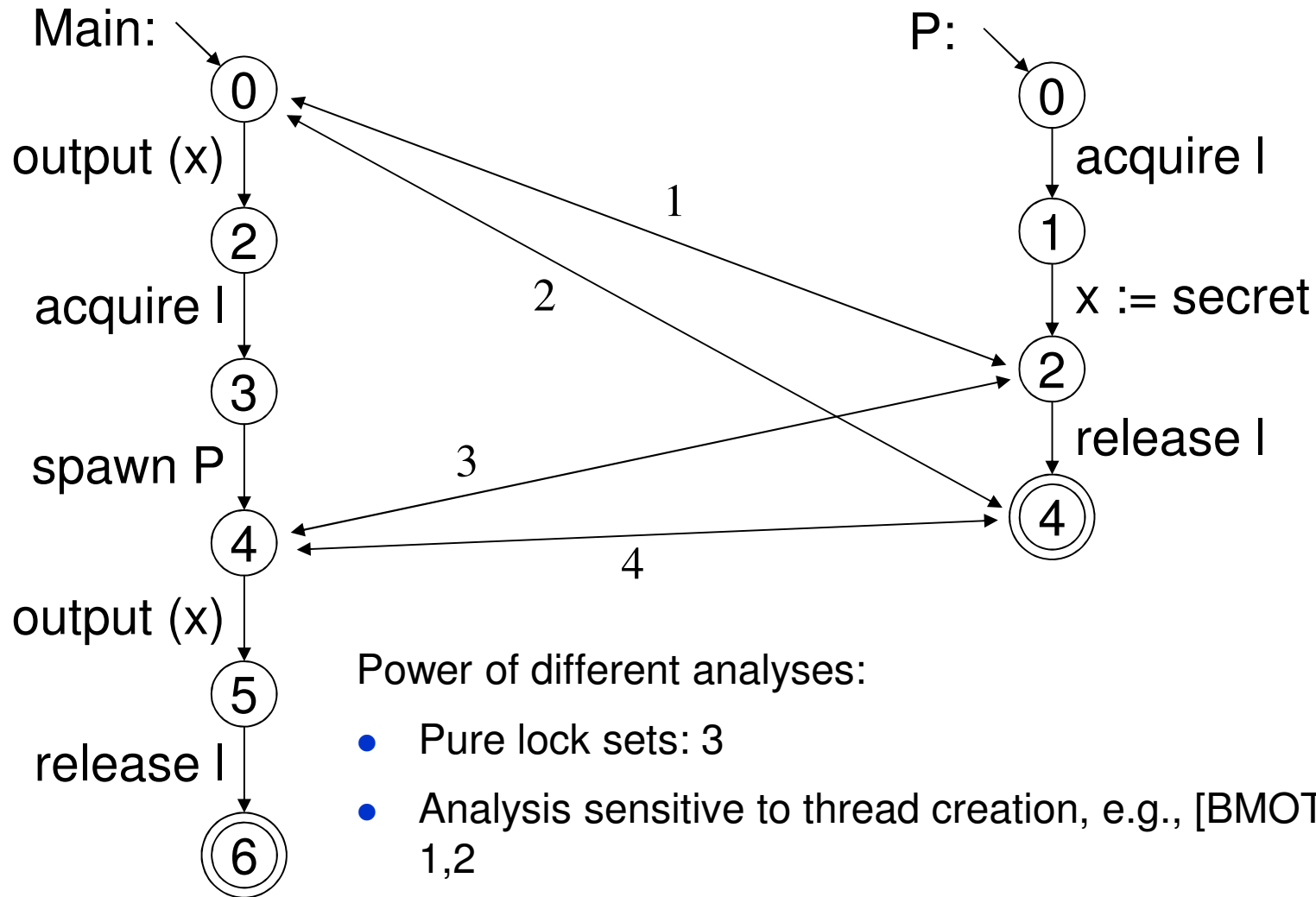
- Assume finite set of locks (or monitors)
- Have acquire- and release actions
 - $\text{acq } L, \text{rel } L \in \text{Act}$ f.a. locks L
- Intuition: At any time a lock can be hold by at most one thread
- The Goal: lock-sensitive analysis

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



Of course, also treat branching, loops and 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**.

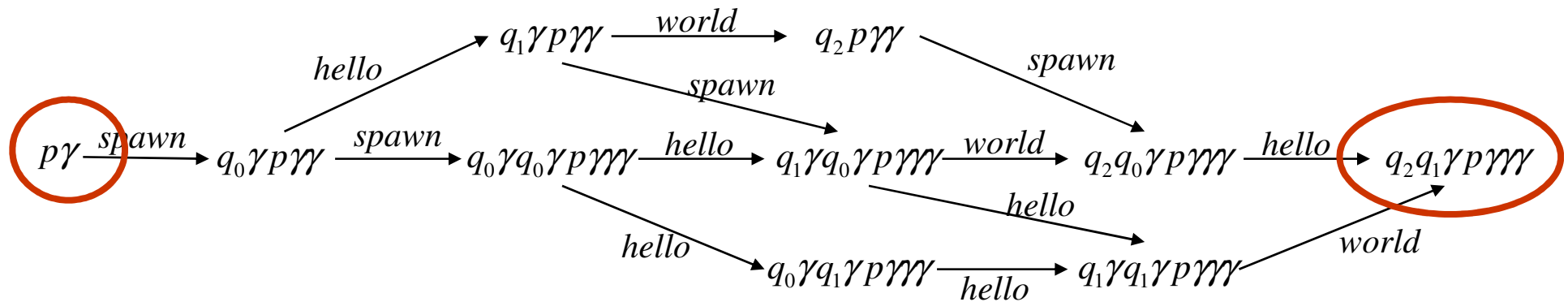
Idea: Collect information about lock usage of each process in „**acquisition histories**“ and check mutual consistency of the collected histories.

Our goal: Lock-sensitive analysis for **systems with thread creation**

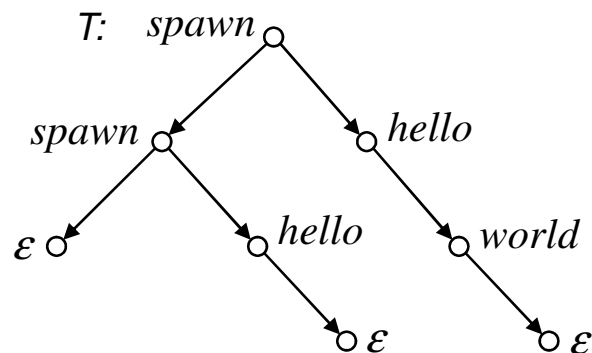
A Tree-Based View of Executions: Action Trees

A DPN: $p\gamma \xrightarrow{\text{spawn}} p\mathcal{W} \triangleright q_0\gamma$ $q_0\gamma \xrightarrow{\text{hello}} q_1\gamma$
 $q_1\gamma \xrightarrow{\text{world}} q_2$

Execution sequences:



Action tree:



We write: $p\gamma \xrightarrow{T}^* q_2q_1\mathcal{W}$

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 1 [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...

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: $p\gamma \xrightarrow{\text{call}} p'\mathcal{G}$

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

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

NoMore: $p\gamma \xrightarrow{\text{no_more}} p''\gamma$

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

An Even More Regular View to Executions

The DPN:

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

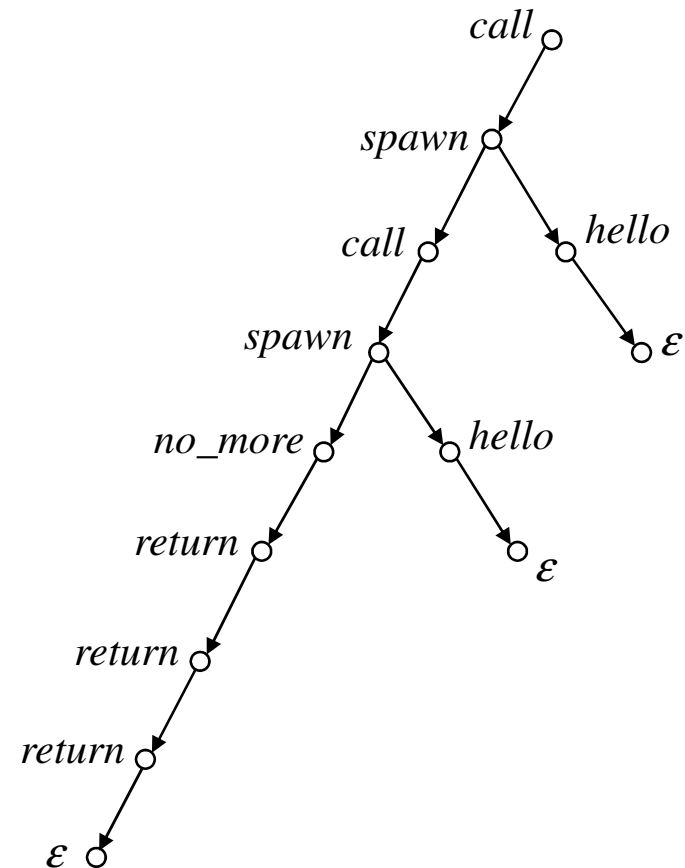
Spawn: $p'\gamma \xrightarrow{\text{spawn}} p\gamma \triangleright q\gamma$

NoMore: $p\gamma \xrightarrow{\text{no_more}} p''\gamma$

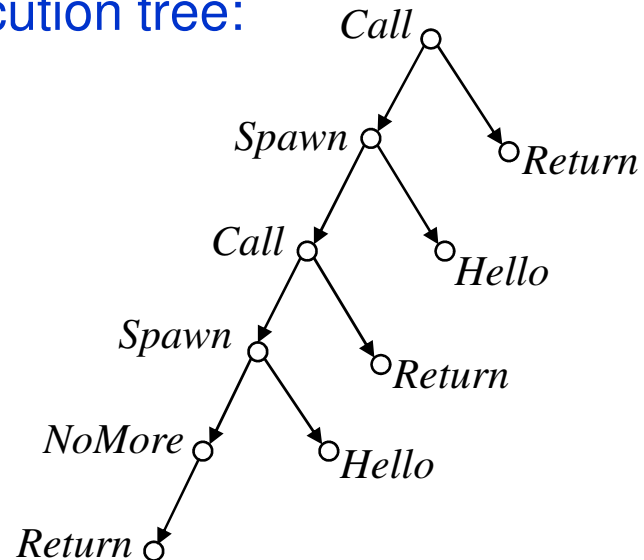
Return: $p''\gamma \xrightarrow{\text{return}} p''$

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

Action tree:



Execution tree:



Execution Trees

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

$\text{postE}^*(p_0\gamma_0) \cap \text{LockSensTrees} \cap \text{ExecTrees}(C)$ 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

The screenshot shows the Eclipse IDE interface. The main editor displays the following Java code for `BSP03.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();
}
```

A dialog box titled "Race found!" is displayed over the code, containing a warning icon and the text: "There might be a race in your program See result view." with an "OK" button.

The bottom of the IDE shows the "Data race result" view with the following information:

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

The status bar at the bottom indicates "Writable", "Smart Insert", and the time "11 : 20".

Java Data-Race Finder: Screenshot 2

The screenshot displays the Eclipse IDE interface. The main editor shows the source code for `BSP03.java`. The code is as follows:

```
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();
}
```

The `Witness View` window is open, showing a call graph. The root node is `Call1 in FakeRootClass.fakeRootMethod`. It leads to `Acq in BSP03.main`, which then leads to `Call2 in BSP03.main`. From `Call2`, there are two outgoing edges: `call` to `Spawn in Thread.start` and `return` to `Nil in BSP03.main`. From `Spawn`, there are two outgoing edges: `spawn` to `Nil in BSP03.run` and `return` to `Ret in Thread.start`.

The `Problems` window at the bottom shows the `Data race result` tab. The overall result is: `Race free: 2/3 Possible race: 1/3`. The results are:

- 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

The screenshot displays the Eclipse IDE interface. The main editor window shows the source code for `BSP03.java`. The code defines a `public class BSP03` that extends `Thread`. It includes a `main` method with a `synchronized` block that starts a thread and sets `x = 42`. The `run` method also contains a `synchronized` block that sets `x = 17`. A `lock1` of type `Lock` is declared at the bottom.

```
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();
}
```

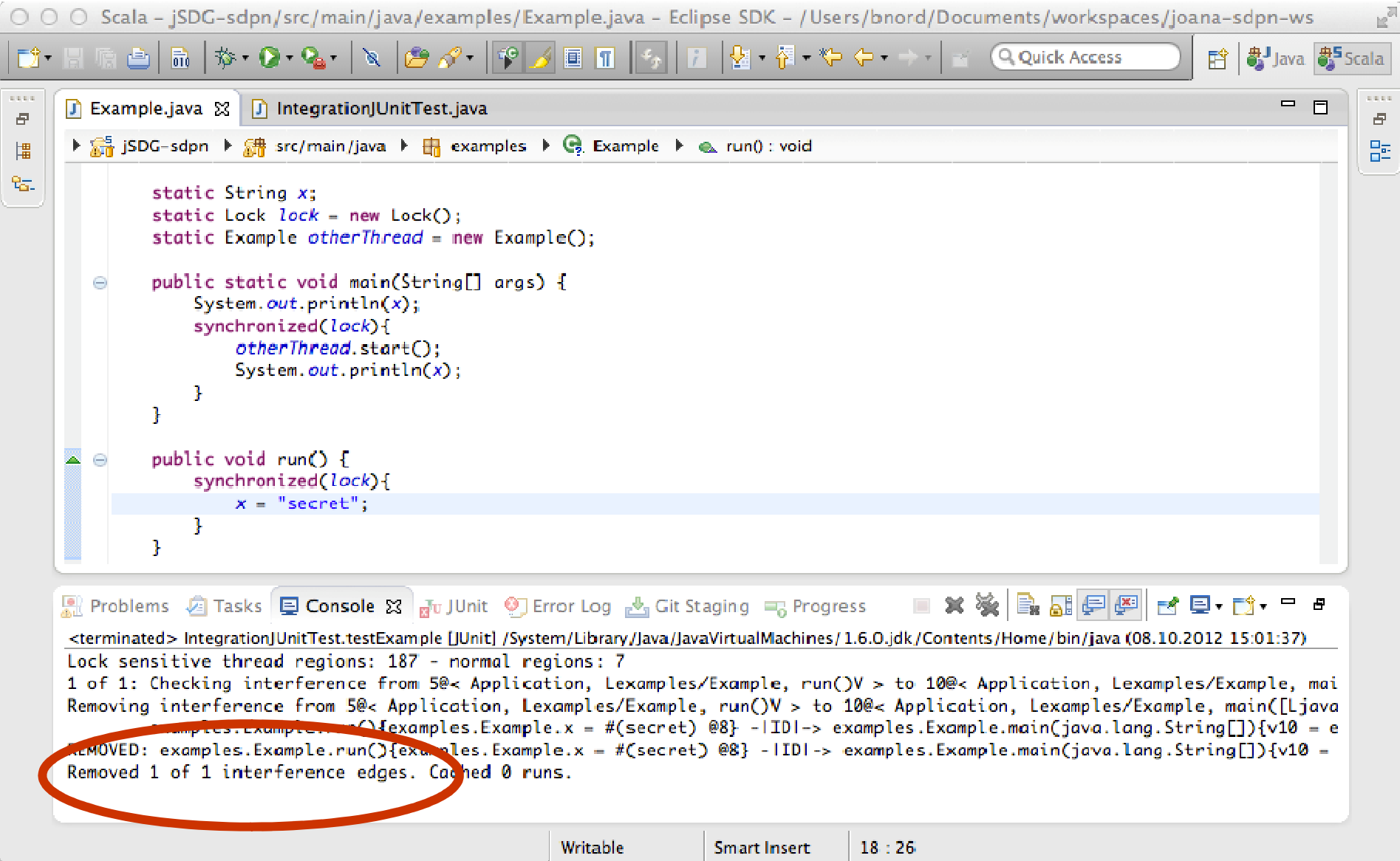
A dialog box titled "No race found" is overlaid on the code. It contains an exclamation mark icon and the text "There is no race in your program". An "OK" button is located at the bottom right of the dialog.

The bottom of the IDE shows the "Problems" view with a tab for "Data race result". The overall result is "Race free: 3/3 Possible race: 0/3". The list of fields includes:

- 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`

The status bar at the bottom indicates "Writable", "Smart Insert", and "16 : 31".

Experimental Integration with Joana: Screenshot



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

Thank you !