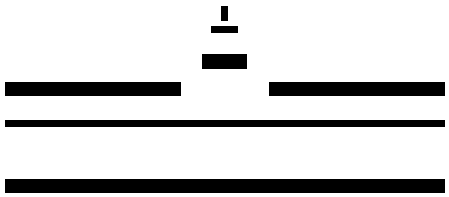


Westfälische
Wilhelms-Universität
Münster

Tree-Regular Analysis of Parallel Programs with Dynamic Thread Creation and Locks

Benedikt Nordhoff

Fachbereich Mathematik und Informatik

Arbeitsgruppe Softwareentwicklung und Verifikation

3. November 2012

What I am working on

- Static reachability analysis of recursive parallel programs.
  - Utilizing DPNs (this talk)
  - Implementation for real programming languages (Java)
  - Applications to information flow control
- Static data flow analyses for sequential programs with applications to information flow control.
  - Utilizing/combining PDGs, path conditions and abstract interpretation
- ...

What is a Monitor-DPN?

- A DPN is a bunch of push down systems (PDS) which can dynamically *spawn* new PDS as a side effect of their transitions.
- The PDS in a Monitor-DPN may additionally synchronize via a finite set of reentrant locks which are bound to the stack.
- Allows to precisely model effects of:
  - Recursive procedures.
  - Dynamic thread creation.
  - Synchronization via a finite set of well nested locks.
- Allows for a finite abstraction of:
  - Method local state.
  - Thread local state.
- Abstracts from shared state.

What is a Monitor-DPN?

- Each process in a Monitor-DPN has a control state and a stack which contains stack symbols and possibly locks.
- There are five kinds of transitions each depends on the control state of the process and the topmost stack symbol:

Base Modifies the control state and top of stack.

Return Modifies the control state and removes the top of stack with a possibly underlying lock.

Call Modifies the control state, top of stack, and adds an additional stack symbol.

Spawn Like base but create a new process with a given control state and stack (without locks).

Use Like call but puts a lock under the new stack symbol. Can only be executed if the lock is not currently on the stack of any other process.

From Executions to Execution Trees

- Executions/traces interleave actions from different threads.
- Action Trees branch executions at spawns, this yields a tree with context free paths.
- Execution Trees additionally branch at procedure calls.
- The set of reachable execution trees is tree regular.

Trace of two proceses a and b

$$c_0 \xrightarrow{base^{\eta_0}}_a c_1 \xrightarrow{call^{\eta_1}}_a c_2 \xrightarrow{base^{\eta_2}}_a c_3 \xrightarrow{spawn^{\eta_3}}_a c_4 \xrightarrow{base^{\eta_4}}_b c_5 \xrightarrow{call^{\eta_5}}_b c_6$$
$$\xrightarrow{base^{\eta_6}}_a c_7 \xrightarrow{ret^{\eta_7}}_a c_8 \xrightarrow{base^{\eta_8}}_a c_9 \xrightarrow{base^{\eta_9}}_b c_{10}$$

Trace of two proceses a and b

$$c_0 \xrightarrow{base^{\eta_0}}_a c_1 \xrightarrow{call^{\eta_1}}_a c_2 \xrightarrow{base^{\eta_2}}_a c_3 \xrightarrow{spawn^{\eta_3}}_a c_4 \xrightarrow{base^{\eta_4}}_b c_5 \xrightarrow{call^{\eta_5}}_b c_6$$
$$\xrightarrow{base^{\eta_6}}_a c_7 \xrightarrow{ret^{\eta_7}}_a c_8 \xrightarrow{base^{\eta_8}}_a c_9 \xrightarrow{base^{\eta_9}}_b c_{10}$$

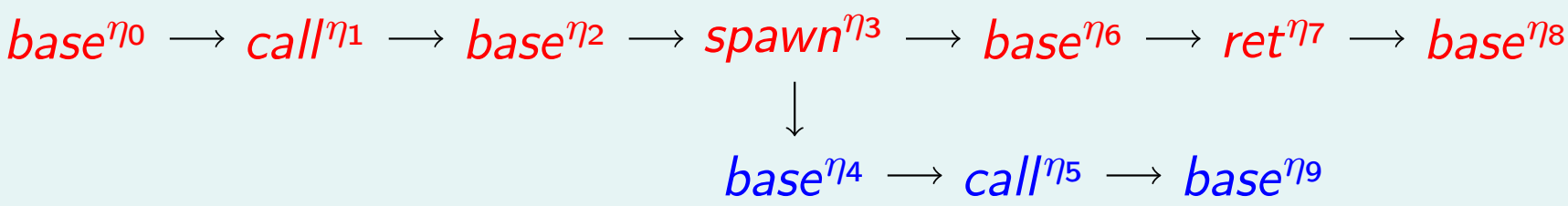
Action tree

$$base^{\eta_0} \longrightarrow call^{\eta_1} \longrightarrow base^{\eta_2} \longrightarrow spawn^{\eta_3} \longrightarrow base^{\eta_6} \longrightarrow ret^{\eta_7} \longrightarrow base^{\eta_8}$$
$$\downarrow$$
$$base^{\eta_4} \longrightarrow call^{\eta_5} \longrightarrow base^{\eta_9}$$

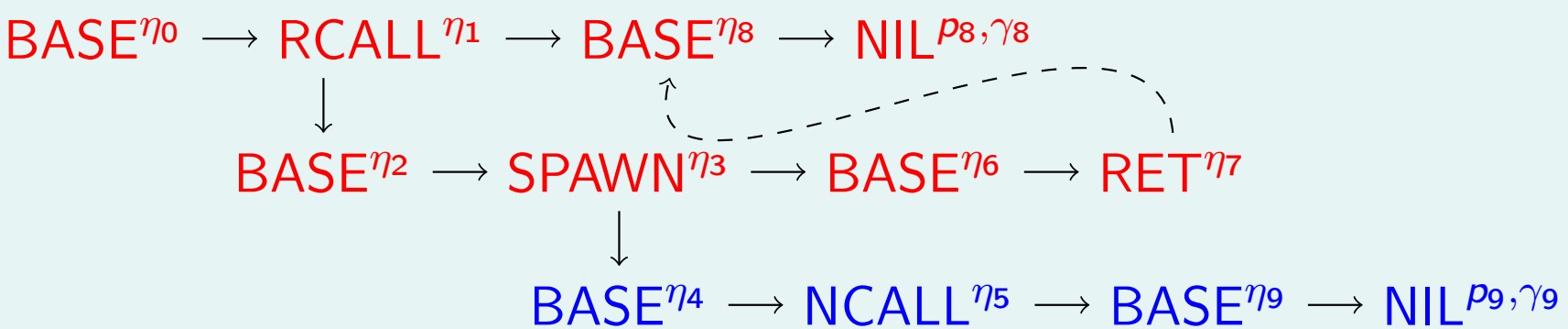
Trace of two proceses a and b

$$c_0 \xrightarrow{base^{\eta_0}}_a c_1 \xrightarrow{call^{\eta_1}}_a c_2 \xrightarrow{base^{\eta_2}}_a c_3 \xrightarrow{spawn^{\eta_3}}_a c_4 \xrightarrow{base^{\eta_4}}_b c_5 \xrightarrow{call^{\eta_5}}_b c_6$$
$$\xrightarrow{base^{\eta_6}}_a c_7 \xrightarrow{ret^{\eta_7}}_a c_8 \xrightarrow{base^{\eta_8}}_a c_9 \xrightarrow{base^{\eta_9}}_b c_{10}$$

Action tree



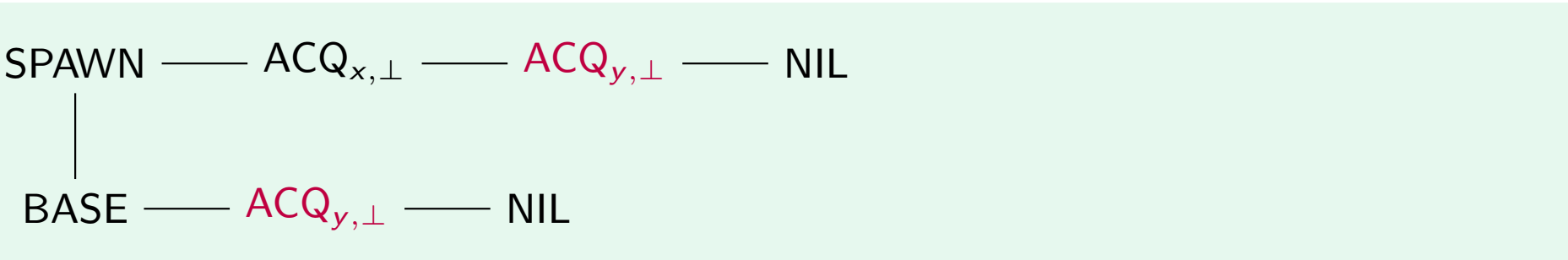
Execution tree



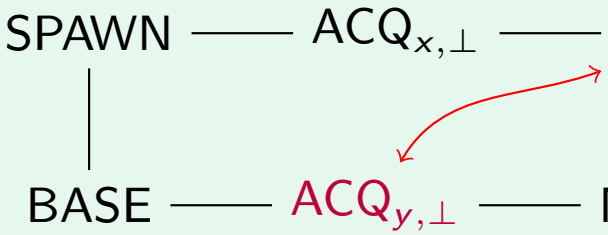
Analysis of DPNs with Execution Trees

- Use tree-regularity to decide reachability of configurations with tree regular properties.
  1. Build a tree automaton accepting all reachable configurations.
  2. Build a tree automaton accepting configurations with property of interest.
  3. Check intersection for emptiness.
- Can also check for reachability from those reachable configurations.
  - Allows to check for arbitrary gen/kill properties e.g. def/use dependencies between two threads over a shared variable.

Not lock-sensitively schedulable execution trees

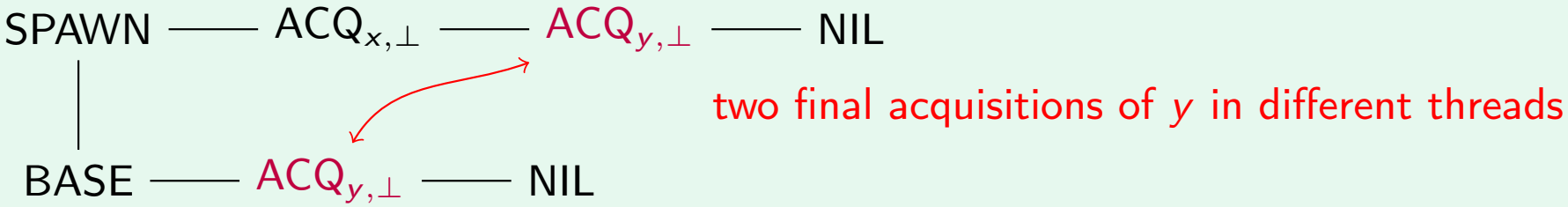


Not lock-sensitively schedulable execution trees

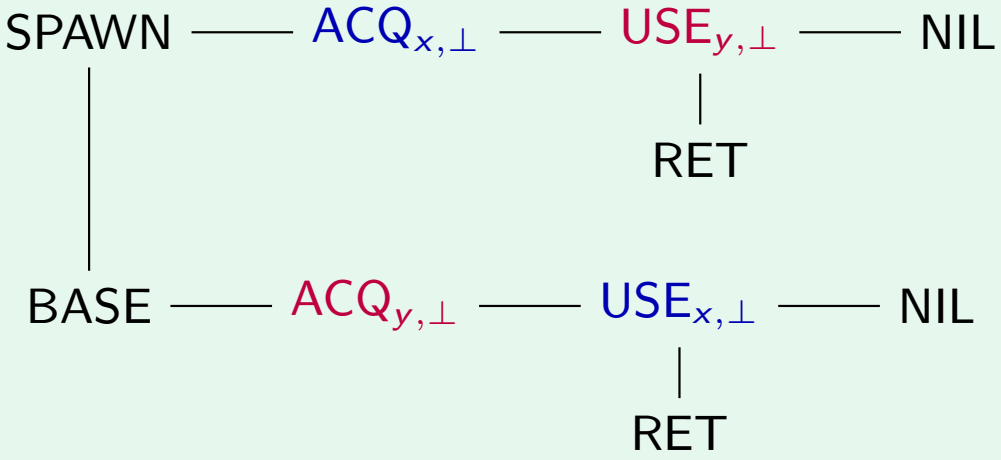


two final acquisitions of y in different threads

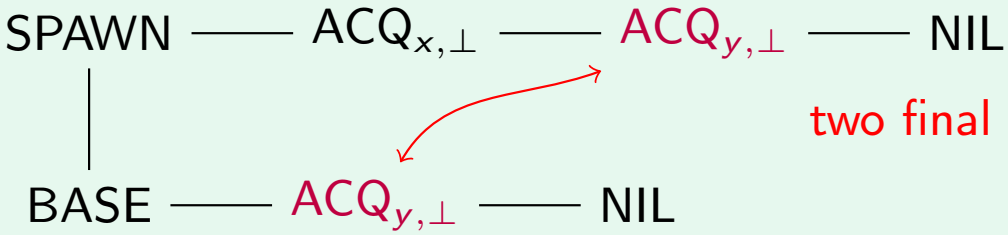
Not lock-sensitively schedulable execution trees



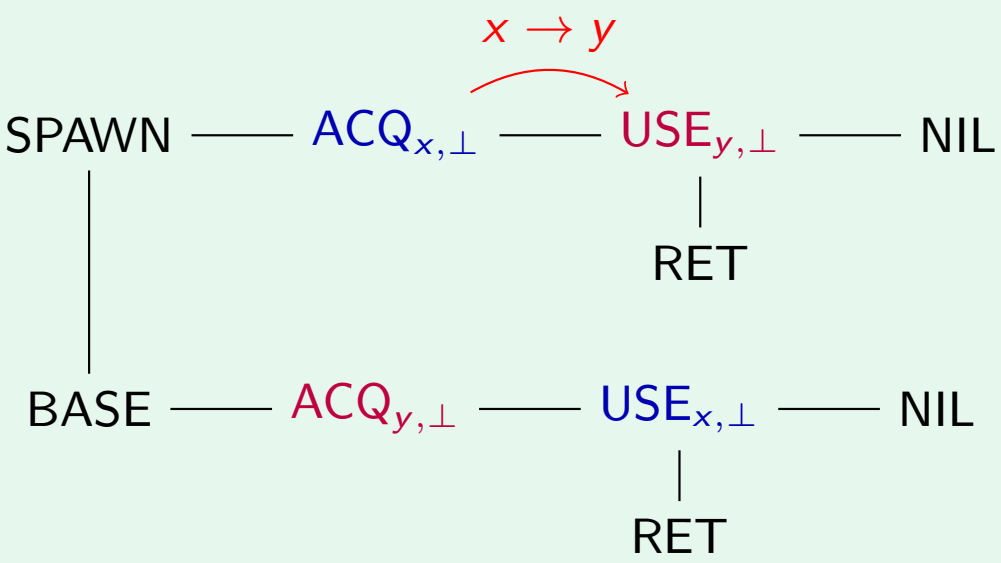
two final acquisitions of y in different threads



Not lock-sensitively schedulable execution trees

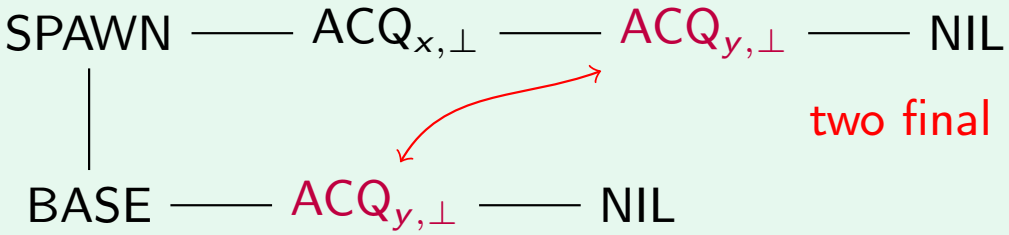


two final acquisitions of y in different threads

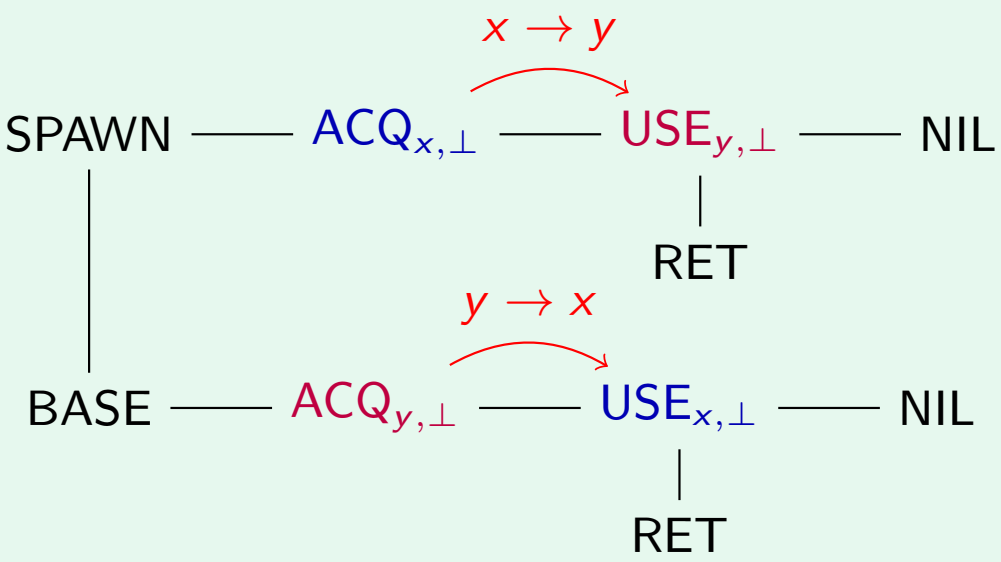


y needs to be used after
x has been finally acquired

Not lock-sensitively schedulable execution trees



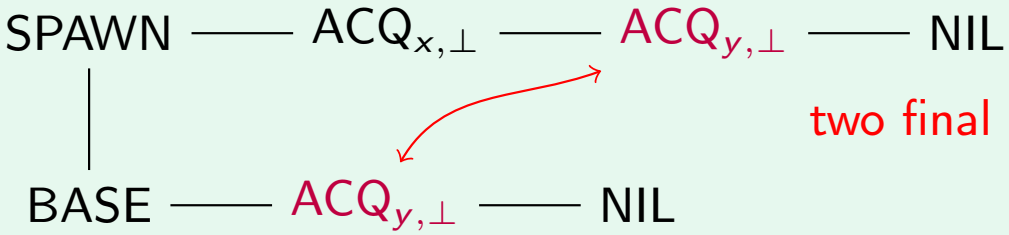
two final acquisitions of y in different threads



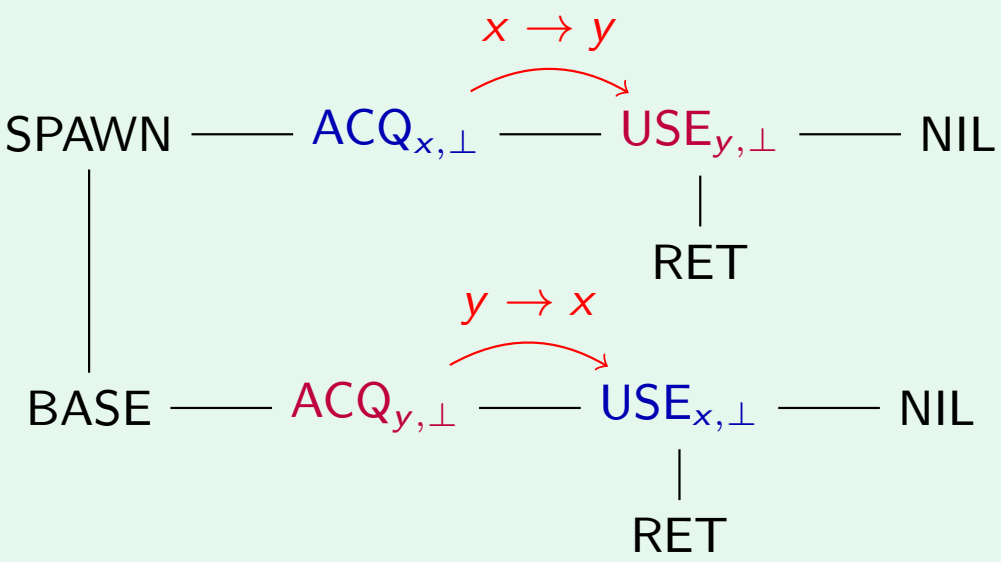
y needs to be used after
x has been finally acquired

x needs to be used after
y has been finally acquired

Not lock-sensitively schedulable execution trees



two final acquisitions of y in different threads



y needs to be used after
x has been finally acquired

x needs to be used after
y has been finally acquired

- These properties are neccessary, sufficient and tree-regular.

Tree Automata for Acquisition Histories

Lock sensitive schedulable execution trees.
Using a generalized version of Kahlon and Gupta's acquisition histories.

- State space: $\{(A, U, G) \mid A \subseteq U \subseteq X,\ G \subseteq X \times X\}$
- Accepting states: $\{(A, U, G) \in Q \mid G \text{ is acyclic}\}$

Interpretation: All operations only non reentrant.

A Locks finally acquired within the tree.

U Locks used or finally acquired within the tree.

G Acquisition graph, $x \to x' \in G \Leftrightarrow x'$ is used or finally acquired *after* x has been finally acquired. (Order constrain)

Tree Automata for Acquisition Histories

Transitions:

$$\text{NIL} \to (\emptyset, \emptyset, \emptyset) \qquad \text{RET} \to (\emptyset, \emptyset, \emptyset)$$

$$\text{BASE}\ \ \alpha \to \alpha \qquad \text{NCALL}\ \ \alpha \to \alpha$$

$$\text{ACQ}_{x,\top}\ \ \alpha \to \alpha$$

$$f\ (A, U, G)\ (A', U', G') \to (A \cup A',\ U \cup U',\ G \cup G') \quad A \cap A' = \emptyset,\ f \in \{\text{RCALL}, \text{USE}_{x,\top}, \text{SPAWN}\}$$

$$\text{USE}_{x,\bot}\ (A, U, G)\ (A', U', G') \to (A \cup A',\ U \cup U' \cup \{x\},\ G \cup G') \quad A \cap A' = \emptyset$$

$$\text{ACQ}_{x,\bot}\ (A, U, G) \to (A \cup \{x\},\ U \cup \{x\},\ G \cup \{(x, u) | u \in U\}) \quad x \notin U$$

The product automaton of these three automatons accepts all lock sensitive execution trees of the DPN.

A Simple Example: Conflict reachability ($T^{R,W}_{CFL}$)

Let R, W be two sets of stack symbols. E.g. reads and writes of some variable.

- State space: $2^{\{r,w\}}$
- Accepting states: $\{\{r, w\}\}$

$$\mathsf{NIL}_\gamma \to \{r\} \quad \gamma \in R \qquad \mathsf{NIL}_\gamma \to \{w\} \quad \gamma \in W$$
$$\mathsf{NIL}_\gamma \to \emptyset \quad \gamma \notin (R \cup W) \quad \mathsf{RET} \to \emptyset$$

$$\{\mathsf{ACQ}, \mathsf{BASE}, \mathsf{NCALL}\}\ \alpha \to \alpha$$

$$\{\mathsf{RCALL}, \mathsf{SPAWN}, \mathsf{USE}\}\ \alpha\ \beta \to \alpha \cup \beta$$

- This tree automaton accepts all trees in which both sets are reached simultaneously. E.g. there exists a datarace.
- $\mathcal{L}(\mathcal{T}_{\mathcal{M}}) \cap \mathcal{L}(\mathcal{T}_{ah}) \cap \mathcal{L}(\mathcal{T}^{R,W}_{CFL}) = \emptyset$ iff there exists no conflict.

Iterable Reachablility

- Have characterized (in some sense) $post^*_{\mathcal{M}}(\{\mathrm{NIL}^{p_0}_{\gamma_0}\})$
- For tree-regular $A \subseteq post^*_{\mathcal{M}}(\{\mathrm{NIL}^{p_0}_{\gamma_0}\})$ can characterize $post^*_{\hat{\mathcal{M}}}(A)$
  - A tree transducer *marks* an intermediate configuration from A in the execution trees.
  - Release structures ensure the locks held at the intermediate configuration can be released before they are needed.
  - $\hat{\mathcal{M}}$ can be a restriction of the DPN $\mathcal{M}$.

Iterable Reachablility

- Have characterized (in some sense) $post^*_{\mathcal{M}}(\{\mathrm{NIL}^{p_0}_{\gamma_0}\})$
- For tree-regular $A \subseteq post^*_{\mathcal{M}}(\{\mathrm{NIL}^{p_0}_{\gamma_0}\})$ can characterize $post^*_{\hat{\mathcal{M}}}(A)$
  - A tree transducer *marks* an intermediate configuration from A in the execution trees.
  - Release structures ensure the locks held at the intermediate configuration can be released before they are needed.
  - $\hat{\mathcal{M}}$ can be a restriction of the DPN $\mathcal{M}$.
- To calculate def/use chains over a shared variable let

  W be the set of execution trees which reached a configuration writing the variable at a given point (possibly in some context).

  $\hat{\mathcal{M}}$ the DPN without killing transitions on the variable.

  R the set of execution trees which reached a configuration reading the variable at a given point (possibly in some context).

  Check: $post^*_{\hat{\mathcal{M}}}(post^*_{\mathcal{M}}(\{\mathrm{NIL}^{p_0}_{\gamma_0}\}) \cap W) \cap R \stackrel{?}{=} \emptyset$

Iterable Reachability How To

- Use unary CUT-nodes to mark an arbitrary intermediate configuration in execution trees.
- Can use a tree transducer to obtain the intermediate configuration.
- A tree transducer is a tree automata with output.
- The inverse image of regular sets under tree transducers is regular.
  - Obtains trees where the marked intermediate configuration is accepted by a given tree automata.
- Check for lock sensitive schedulability using acquisition and release structures.

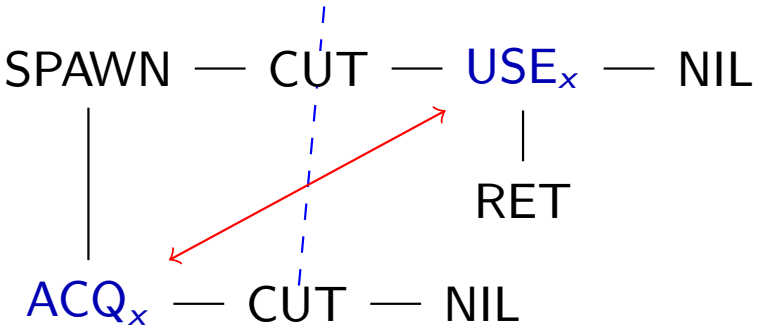
Cut-wellformed Execution Trees

- A tree-automata checks that the cut-nodes actually mark an intermediate configuration by ensuring that:
  - Every process spawned before the intermediate configuration contains exactly one cut-node.
  - No process spawned after the intermediate configuration contains a cut-node.

Cut-wellformed Execution Trees

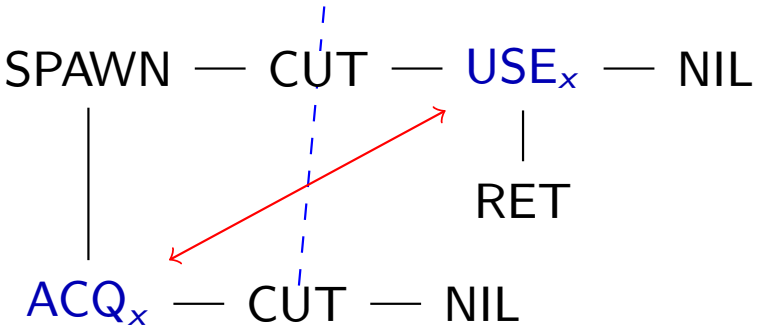
- A tree-automata checks that the cut-nodes actually mark an intermediate configuration by ensuring that:
  - Every process spawned before the intermediate configuration contains exactly one cut-node.
  - No process spawned after the intermediate configuration contains a cut-node.
- Still need to check for lock-sensitive schedulability.

Lock-sensitive Schedulability with Cuts

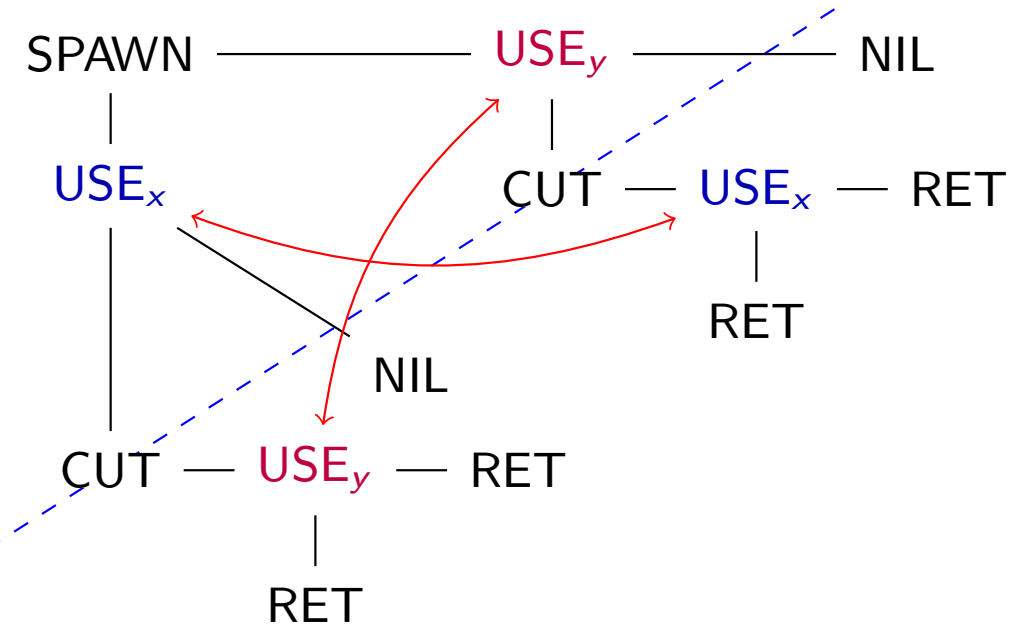


x is *held throughout* the second phase by the spawned thread but needed by the first.

Lock-sensitive Schedulability with Cuts

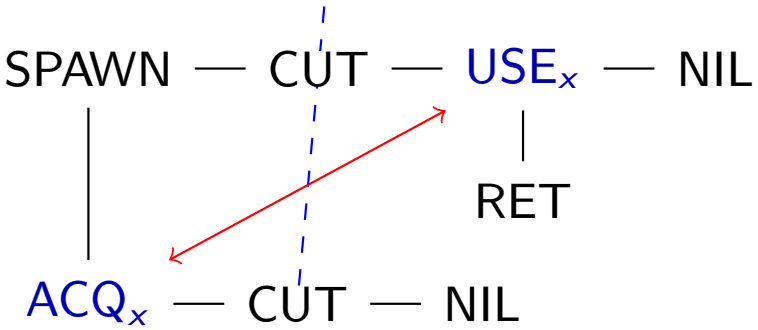


x is *held throughout* the second phase by the spawned thread but needed by the first.

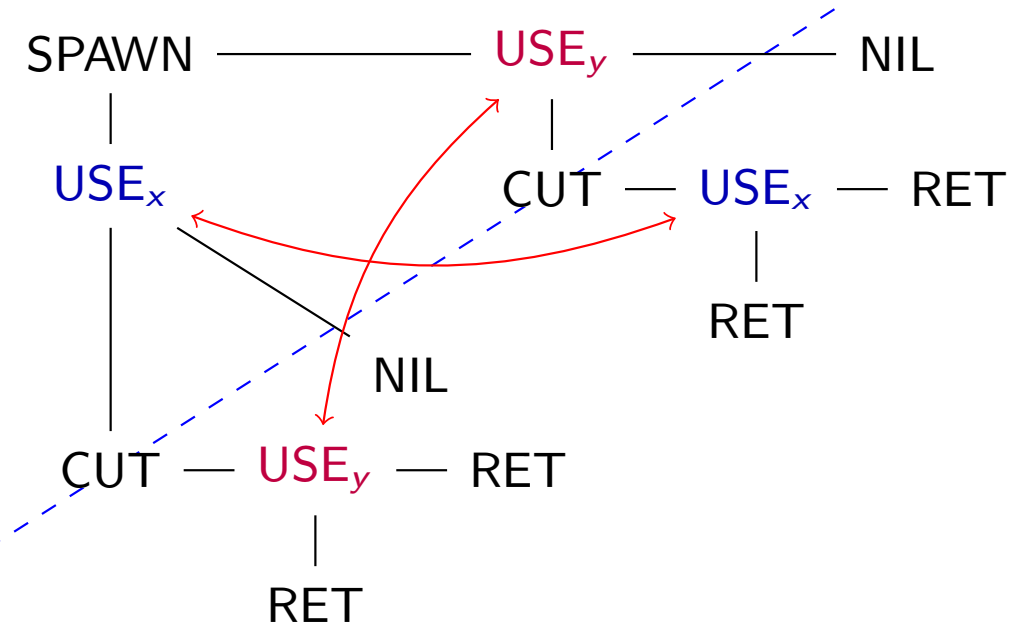


Deadlock each thread needs the a lock held by the other thread to release the lock it holds.

Lock-sensitive Schedulability with Cuts



x is *held throughout* the second phase by the spawned thread but needed by the first.



Deadlock each thread needs the a lock held by the other thread to release the lock it holds.

- Combined with the previous acquisition histories these properties are necessary, sufficient and tree-regular.

Locks Held Throughout

Ensures no lock finally acquired at the cut is used afterwards.
State space: $\mathbb{B} \times 2^X \times 2^X$
Accepting states: $\{(C, A, U) \in \mathbb{B} \times 2^X \times 2^X \mid A \cap U = \emptyset\}$

$$\begin{aligned}
\text{NIL} &\to (\bot, \emptyset, \emptyset) \\
\text{RET} &\to (\bot, \emptyset, \emptyset) \\
\text{BASE}\ \ p &\to p \\
\text{NCALL}\ \ p &\to p \\
\text{ACQ}_{x,\top}\ \ p &\to p \\
\text{CUT}\ (\bot, A, U) &\to (\top, A, U)
\end{aligned}$$

$$\begin{aligned}
\text{RCALL}\ (\bot, A, U)\ (\bot, A', U') &\to (\bot,\ A \cup A',\ U \cup U') \\
\text{RCALL}\ (\top, A, U)\ (\bot, _, U') &\to (\top,\ A,\ U \cup U') \\
\text{RCALL}\ (\bot, A, _)\ (\top, A', U') &\to (\top,\ A \cup A',\ U') \\
\text{USE}_{x,\bot}\ (\bot, A, U)\ (\bot, A', U') &\to (\bot,\ A \cup A',\ U \cup U' \cup \{x\}) \\
\text{USE}_{x,\bot}\ (\top, A, U)\ (\bot, _, U') &\to (\top,\ A,\ U \cup U') \\
\text{USE}_{x,\bot}\ (\bot, A, _)\ (\top, A', U') &\to (\top,\ A \cup A',\ U') \\
\text{ACQ}_{x,\bot}\ (\bot, A, U) &\to (\bot,\ A,\ U \cup \{x\}) \\
\text{ACQ}_{x,\bot}\ (\top, A, U) &\to (\top,\ A \cup \{x\},\ U) \\
\text{SPAWN}\ (_, A, U)\ (C, A', U') &\to (C,\ A \cup A',\ U \cup U')
\end{aligned}$$

(Reentrant use nodes are handled like returning calls.)

Release of Locks

Ensures that all locks used at the cut can be released.

State space $\mathbb{B} \times 2^X \times 2^{X \times X}$

Accepting states: $\mathbb{B} \times 2^X \times \{G \in 2^{X \times X} \mid G \text{ is acyclic}\}$

$$\text{NIL} \to (\bot, \emptyset, \emptyset) \qquad \text{RET} \to (\bot, \emptyset, \emptyset)$$

$$\text{BASE}\ \ p \to p \qquad \text{NCALL}\ \ p \to p$$

$$\text{ACQ}_{x,\_}\ \ p \to p \qquad \text{CUT}\ (\bot, U, G) \to (\top, U, G)$$

$$\text{RCALL}\ (C, U_c, G_c)\ (\bot, U_r, G_r) \to (C, U_c \cup U_r, G_c \cup G_r)$$

$$\text{RCALL}\ (\bot, \_, G_c)\ (\top, U_r, G_r) \to (\top, U_r, G_c \cup G_r)$$

$$\text{USE}_{x,\top}\ (C, U_c, G_c)\ (\bot, U_r, G_r) \to (C, U_c \cup U_r, G_c \cup G_r)$$

$$\text{USE}_{x,\_}\ (\bot, \_, G_c)\ (\top, U_r, G_r) \to (\top, U_r, G_c \cup G_r)$$

$$\text{USE}_{x,\bot}\ (\bot, U_c, G_c)\ (\bot, U_r, G_r) \to (\bot, U_c \cup U_r \cup \{x\}, G_c \cup G_r)$$

$$\text{USE}_{x,\bot}\ (\top, U_c, G_c)\ (\bot, U_r, G_r) \to (\top, U_c \cup U_r, G_c \cup G_r \cup \{(u, x) | u \in U_c\})$$

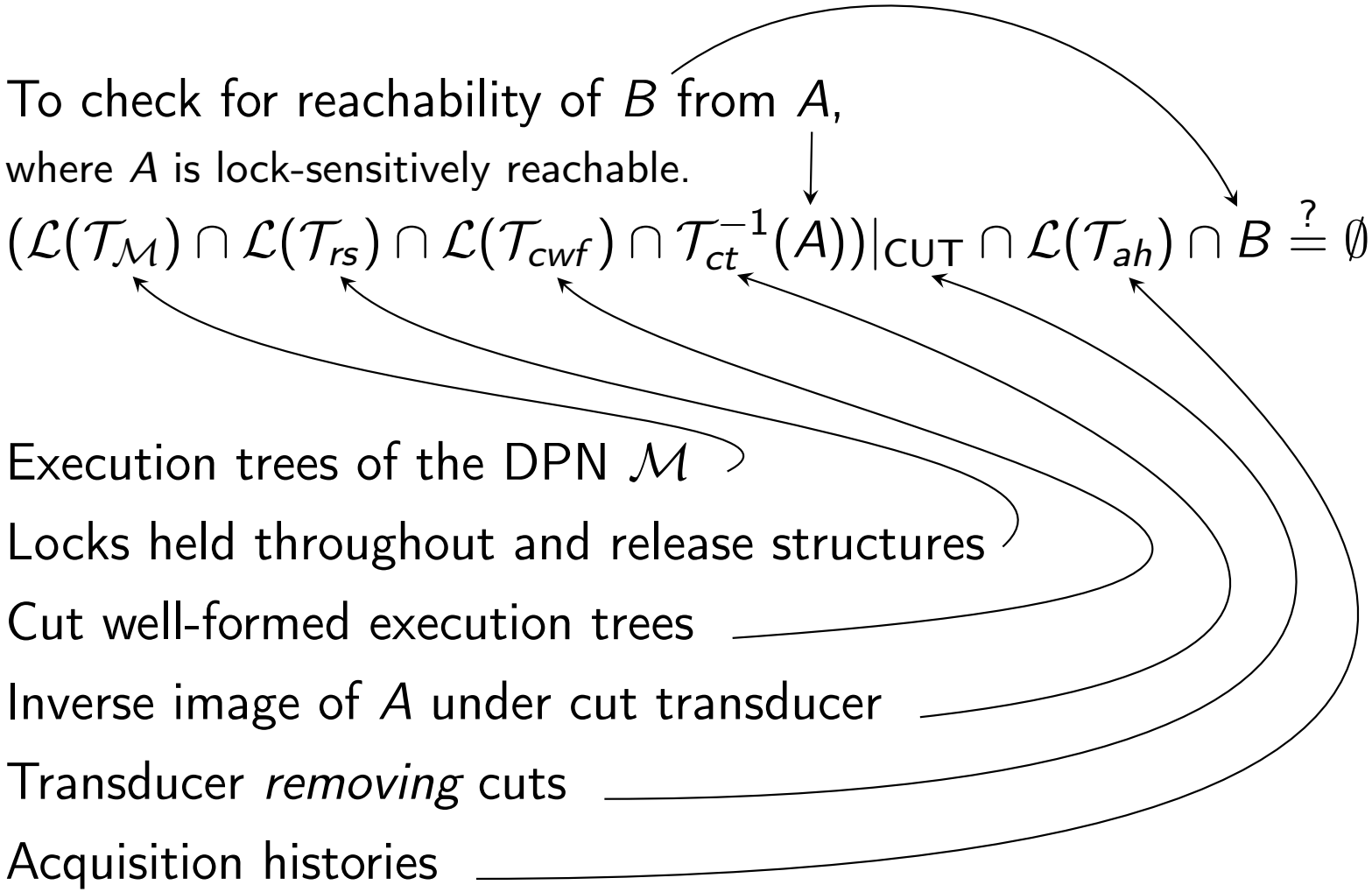
$$\text{SPAWN}\ (\bot, \_, \emptyset)\ (\bot, U_r, \emptyset) \to (\bot, U_r, \emptyset)$$

$$\text{SPAWN}\ (\top, \_, G_s)\ (B, U_r, G_r) \to (B, U_r, G_s \cup G_r)$$

Iterable Reachability with Tree Automata

- To check for reachability of B from A,
  where A is lock-sensitively reachable.
  $(\mathcal{L}(\mathcal{T}_{\mathcal{M}}) \cap \mathcal{L}(\mathcal{T}_{rs}) \cap \mathcal{L}(\mathcal{T}_{cwf}) \cap \mathcal{T}_{ct}^{-1}(A))|_{\mathsf{CUT}} \cap \mathcal{L}(\mathcal{T}_{ah}) \cap B \stackrel{?}{=} \emptyset$

Iterable Reachability with Tree Automata

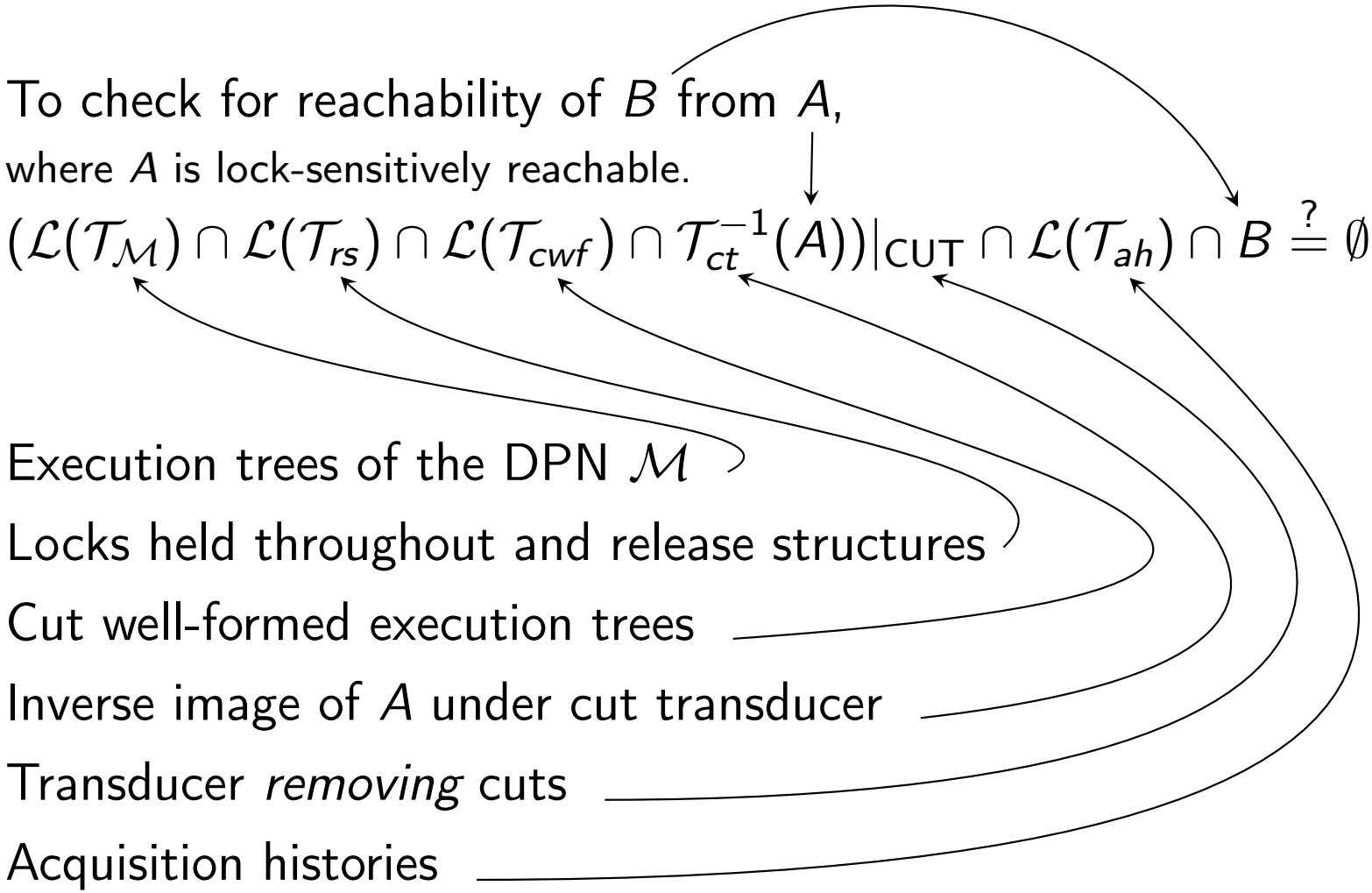


- To check for reachability of B from A,
  where A is lock-sensitively reachable.

$$(\mathcal{L}(\mathcal{T}_{\mathcal{M}}) \cap \mathcal{L}(\mathcal{T}_{rs}) \cap \mathcal{L}(\mathcal{T}_{cwf}) \cap \mathcal{T}_{ct}^{-1}(A))|_{\mathsf{CUT}} \cap \mathcal{L}(\mathcal{T}_{ah}) \cap B \stackrel{?}{=} \emptyset$$

- Execution trees of the DPN $\mathcal{M}$
- Locks held throughout and release structures
- Cut well-formed execution trees
- Inverse image of A under cut transducer
- Transducer *removing* cuts
- Acquisition histories

Iterable Reachability with Tree Automata



- To check for reachability of B from A,
  where A is lock-sensitively reachable.

$$(\mathcal{L}(\mathcal{T}_{\mathcal{M}}) \cap \mathcal{L}(\mathcal{T}_{rs}) \cap \mathcal{L}(\mathcal{T}_{cwf}) \cap \mathcal{T}_{ct}^{-1}(A))|_{\mathsf{CUT}} \cap \mathcal{L}(\mathcal{T}_{ah}) \cap B \stackrel{?}{=} \emptyset$$

- Execution trees of the DPN $\mathcal{M}$
- Locks held throughout and release structures
- Cut well-formed execution trees
- Inverse image of A under cut transducer
- Transducer *removing* cuts
- Acquisition histories
- Before projecting out the cut-nodes, one can also check that some transitions don't occur after/before the cut to *switch* the DPN.

Implemented for Analysis of Java Programs

- Utilizing the T.J. Watson Libraries for Analysis (WALA) from IBM.
- Using XSB a tabulating Prolog system to evaluate tree automata.
- Some automata are extremely nondeterministic when evaluated top down and other when evaluated bottom up.
  - Emptiness check for product automata explores some automata bottom up and other top down.
  - Avoids exploration of unfeasible states.
- Witnesses can be generated.
- Integrated with *Joana* an information flow control tool for Java.
  - Currently checking individual def/use dependencies between threads for feasibility.
  - Want to check additionally:
    - Interferences in given contexts during the slicing phase.
    - Multiple interferences on critical paths.

Thanks for listening!


Ahmed Bouajjani, Markus Müller-Olm, and Tayssir Touili.
Regular symbolic analysis of dynamic networks of pushdown systems.
In *CONCUR 2005 – Concurrency Theory*, 2005.


Peter Lammich, Markus Müller-Olm, and Alexander Wenner.
Predecessor sets of dynamic pushdown networks with tree-regular constraints.
In *Computer Aided Verification*, 2009.


Thomas Gawlitza, Peter Lammich, Markus Müller-Olm, Helmut Seidl, and Alexander Wenner.
Join-lock-sensitive forward reachability analysis for concurrent programs with dynamic process creation.
In *Verification, Model Checking, and Abstract Interpretation*, 2011.


Vineet Kahlon, Franjo Ivančić, and Aarti Gupta.
Reasoning about threads communicating via locks.
In *In CAV*. Springer, 2005.


Benedikt Nordhoff, Peter Lammich, and Markus Müller-Olm.
Iterable forward reachability analysis of Monitor-DPNs.
Submitted for publication, 2012.