Iterable Forward Reachability Analysis of Monitor-DPNs

Benedikt Nordhoff, Markus Müller-Olm
Institut für Informatik, Westfälische Wilhelms-Universität Münster*
b.n@wwu.de, markus.mueller-olm@wwu.de

Peter Lammich
Fakultät für Informatik, TU München
lammich@in.tum.de

There is a close connection between data-flow analysis and model checking as observed and studied in the nineties by Steffen and Schmidt. This indicates that automata-based analysis techniques developed in the realm of infinite-state model checking can be applied as data-flow analyzers that interpret complex control structures, which motivates the development of such analysis techniques for ever more complex models. One approach proposed by Esparza and Knoop is based on computation of predecessor or successor sets for sets of automata configurations. Our goal is to adapt and exploit this approach for analysis of multi-threaded Java programs. Specifically, we consider the model of Monitor-DPNs for concurrent programs. Monitor-DPNs precisely model unbounded recursion, dynamic thread creation, and synchronization via well-nested locks with finite abstractions of procedure- and thread-local state. Previous work on this model showed how to compute regular predecessor sets of regular configurations and tree-regular successor sets of a fixed initial configuration. By combining and extending different previously developed techniques we show how to compute tree-regular successor sets of tree-regular sets. Thereby we obtain an iterable, lock-sensitive forward reachability analysis. We implemented the analysis for Java programs and applied it to information flow control and data race detection.

1 Introduction

Developing, debugging and analysing parallel programs is notoriously hard because the actual scheduling of different threads is mostly unspecified. For the developer it requires a lot of experience and skill to foresee all possible relevant impacts of the unknown scheduling. Debugging is difficult as critical bugs may only manifest under scheduling conditions that are hard to provoke during testing, but might be typical in a particular environment. Moreover, scheduling-dependent bugs are often hard to reproduce. From the program analysis side, parallel programs are tough as one quickly reaches the decidability barrier. Of particular relevance is the result by Ramalingam [19] that full context and synchronization sensitivity in the presence of rendezvous-style synchronization and recursion renders even reachability undecidable. Kahlon, Ivancic and Gupta [13] demonstrated how rendezvous-style synchronization can be modeled by non-well-nested locks. On another track Müller-Olm and Seidl [18] showed that optimal slicing for recursive parallel programs is undecidable. The goal therefore is to find reasonable abstractions where the problem of interest is effectively decidable, but sufficient precision is retained.

Steffen observed in the early nineties that data-flow analysis problems can be solved via model checking [22], an observation elaborated later by Schmidt [20] and Schmidt and Steffen [21]. This observation indicates that automata-based analysis procedures developed in the realm of infinite-state model checking can be applied as data-flow analyzers that interpret complex control structures. One approach proposed by Esparza and Knoop [7] is based on computation of predecessor or successor sets for sets of automata configurations. Our goal is to adapt and exploit this approach for analysis of multi-threaded Java programs. More specifically, we utilize an automata-theoretic approach which handles full recursion, thread

*This work was partially funded by DFG projects IFC for Mobile Components (MU 1508/2) within priority program RS3 (SPP 1496), and OpiAT (MU 1508/1).

A. Banerjee, O. Danvy, K.-G. Doh, J. Hatcliff (Eds):
David A. Schmidt's 60th Birthday Festschrift
EPTCS 129, 2013, pp. 384–403, doi:10.4204/EPTCS.129.24

creation and synchronization via reentrant well-nested locks. The underlying model is that of Dynamic Pushdown Networks (DPN) which was introduced (without locks) by Bouajjani *et al.* [3] in 2005. They showed how the pre^* operator for calculating predecessor sets of configurations – words over an alphabet of control and stack symbols – preserves regularity in the same way as done earlier in the sequential case [2]. Later, the DPN-model was enriched to allow for scheduling restrictions based on well-nested locks [15]. This was done by extending the *acquisition history* technique of Kahlon *et al.* [13] and thereby obtaining a tree-regular characterization of lock-sensitive schedulability, which was encoded into the control states of the DPN. The converse set of configurations reachable from a given configuration ($post^*$) was shown to be non-regular in the word semantics [3]. Gawlitza *et al.* [9] then switched from configuration words to *execution trees*. An execution tree describes the steps of an execution as well as the reached configuration. The tree structure makes visible both the parallel execution of steps in different threads and the nesting structure of procedure calls and returns. This allowed them to obtain a tree-regular representation of the set of execution trees reachable from an initial single thread configuration. Note that the tree-based semantics is strictly more expressive than the word-based semantics: By a suitable traversal of the execution tree one can obtain the configuration word and every regular property over configuration words can be translated to an equivalent tree-regular property over execution trees. Conversely, the execution tree cannot be obtained from the configuration word as it contains information of the history about the execution. The applicability of these techniques for the analysis of programs in real programming languages like Java was postulated by different authors but never implemented.

We extend this research by the following three main contributions:

1. In a first step we adapt the forward analysis approach of Gawlitza *et al.* [9] to fit the setting of well-nested reentrant locks as found in Java. The resulting technique can treat reachability problems like checking a program for data races.
2. In order to solve data-flow problems, like calculating def-use dependencies, we extend our technique to calculate $post^*$ for arbitrary tree-regular sets of reachable configurations. The extension is based on the insight that each execution tree that is reached in the course of an execution can be obtained from any later reached tree. In order to check for lock sensitive schedulability we adapt techniques based on so called *acquisition* and *release structures* [12, 15]. We do this by identifying previously defined criteria for schedulability and checking those in a modular way by defining appropriate tree automata.
3. We report on our implementation of the developed techniques for the analysis of Java programs. Our application includes an Eclipse IDE plugin for data race detection and an inter-thread def-use dependency checker for an information flow control analysis tool.

As we head for an actual implementation, we describe all constructions more explicitly than done in previous work. We will explain the key insights from the original correctness proofs of the used techniques. Afterwards, we report on obstacles, solutions and results from our implementation for the analysis of parallel Java programs. We also provide an undecidability result in the presence of `wait`-calls as found in Java.

1.1 Related Work / History

This work can be related to several lines of previous research. There is a vast amount of literature on analyzing infinite-state systems with automata-based techniques. We only cover a few papers here that are closely related to our work. Dating back till 1964 is research concerning the preservation of regularity under various term rewrite systems. Büchi's work on *Regular Canonical Systems* [4], which,

among other things, implies that the set of reachable configurations of a push-down automata is regular, is probably one of the first. In 1997, Bouajjani, Esparza and Maler [2] showed how predecessor sets of regular sets of push-down automata configurations are effectively regular. They used their results to perform model checking of linear and branching time logic of push-down automata. This work was the basis for the transition to multi-threaded programs in the previously mentioned work by Bouajjani *et al.* [3] from 2005 where Dynamic Pushdown Networks were introduced. Lugiez and Schnoebelen in 1998 [16, 17] applied tree automata-based analysis techniques to another class of parallel systems, so called PA-processes and showed that tree-regularity of sets of PA-process terms is preserved by pre^* and $post^*$ operations. Then in 2000 Esparza and Podelski [8] showed how these sets can effectively be calculated as the least model of a set of Horn-clauses, similar to the encoding in logical programs that we use in our implementation. This could then be used to solve bitvector data-flow analysis problems. Also in 2000, Bouajjani *et al.* [1] interpreted context free-grammars as term rewriting systems over regular sets of words and gave an efficient algorithm to compute pre^* images. This could be used to efficiently answer various problems concerning context free-grammars. They also stated an equivalent encoding as a set of Horn-clauses which led to an algorithm of equal time complexity.

Work on how to use automata theoretic and model checking approaches to solve data-flow analysis problems includes work by Steffen [22] from 1991 who showed how data-flow analysis problems can be expressed as model checking problems of modal mu-calculus formulas against a finite program model. Further research in this direction was done in 1998 by Schmidt partially in collaboration with Steffen [20, 21] who incorporated abstract interpretation in this approach. Work on solving data-flow analysis problems via pre^* and $post^*$ computations of regular sets was done by Esparza and Knoop in 1999 [7].

1.2 Motivation

This work was motivated by the goal to improve the precision of an information flow control analysis for parallel Java programs based on system dependence graphs (SDG) as described for example by Hammer and Snelting [11] or Giffhorn [10]. The specific aim was to improve the handling of data flows between different threads in the SDG-based analysis. Besides, we obtain the first implementation of the previously developed techniques for real world applications.

In order to illustrate the effects of locking and thread creation, consider the snippets of Java code in Table 1. Assuming an adequate context where x and y are initially 0 and t2 is an object of the Java class Thread: Can the programs print the value 42? The answer is no for all programs, but how can an analysis automatically infer this? An analysis abstracting from thread creation must assume that 42 can be printed by the first program. The second example shows the interplay between thread creation and locking. The main thread holds a lock while spawning the second thread and only prints the variable before releasing the lock. However, the second thread needs to use the lock before writing 42 to the variable. In the third example the write of 42 would need to be scheduled between the assignment x = 17 and the print which is imposible due to the shared lock. The fourth example is similar from a scheduling point of view but there are two transfers involved. While both transfers are feasible, there is no run exhibiting both. In the fifth example, analogously to the third example, the assignment of 42 to x must be scheduled right before printing x, but there is no shared lock held by both processes at these points. By exhaustive inspection of the different possibilities in which the two threads can acquire and release the locks a and b one can see that it is not possible to schedule the assignment as required. The last example is similar but a little more involved. Either there is an intervening kill or the program reaches a deadlock. A lock-sensitive DPN-based analysis as described in the remainder of this paper can treat all these effects precisely and automatically infers that none of the example programs can print the value 42.

#	main method	t2's run method
1	`print(x);` `t2.start();`	`x = 42;`

2	`synchronized(a){`
	`  t2.start();`
	`  print(x);  }`
		`synchronized(a){...}`
		`x = 42;`

3	`t2.start();`
	`synchronized(a){`
	`  x = 17;`
	`  print(x);  }`
		`synchronized(a){`
		`  x = 42;`
		`}`

4	`t2.start();`
	`synchronized(a){`
	`  y = 42;`
	`  print(x);  }`
		`synchronized(a){`
		`  x = y;`
		`}`

5	`t2.start();`
	`synchronized(a){`
	`  synchronized(b) {...}`
	`  x = 17;`
	`  print(x);  }`
		`synchronized(b){`
		`  synchronized(a) {...}`
		`  x = 42;`
		`}`

6	`t2.start()`
	`synchronized(a){`
	`  synchronized(b){`
	`  x = 42;  }`
	`  x = 23;  }`
		`synchronized(b){`
		`  if (...) {`
		`  synchronized(a){...}`
		`  } else {`
		`  x = 17; }`
		`  print(x);  }`

Table 1: Spurious examples of data flows between threads

2 The Monitor-DPN Model

We now define the underlying model for our analyses. DPNs precisely model unbounded recursion and thread creation with finite abstractions of method- and thread-local state but abstract from global state. Intuitively, a DPN is a set of push-down systems which are able to add new push-down systems as a side effect of their transitions. The latter corresponds to thread creation. In a Monitor-DPN we also allow the threads to communicate via a finite set of reentrant locks, which are used in a well-nested fashion. Reentrance means that a thread may acquire the same lock multiple times and releases it only after a matching number of release-operations. We enforce well-nestedness syntactically by only allowing a lock to be acquired when pushing a local state onto the stack and releasing it implicitly when the old stack level is reached again. That is, lock acquisition and release is bound to procedure calls. Note that if locks are used in a syntactically well-nested fashion – like in synchronized blocks in Java – they can be transformed to procedure calls with attached locks. We call locks used in this way *monitors*.

Definition 1. A *Monitor-DPN* $\mathscr{M}$ is a tuple $(\mathrm{Act}, P, \Gamma, X, \Delta, (p_0, \gamma_0))$ consisting of an initial configuration $(p_0, \gamma_0) \in P \times \Gamma$ and finite sets of: actions Act, control states P, stack symbols Γ, locks X and a set Δ of transition rules of the form:

$$\text{(Base) } p\gamma \overset{a}{\hookrightarrow} p'\gamma' \qquad \text{(Call) } p\gamma \overset{a}{\hookrightarrow} p'\gamma'\gamma_r \qquad \text{(Return) } p\gamma \overset{a}{\hookrightarrow} p'$$

$$\text{(Spawn) } p\gamma \overset{a}{\hookrightarrow} p_s\gamma_s p'\gamma' \qquad \text{(Monitor) } p\gamma \overset{a,x}{\hookrightarrow} p'\gamma'\gamma_r$$

where $p, p', p_s \in P$, $a \in \mathrm{Act}$, $x \in X$, $\gamma, \gamma', \gamma_r, \gamma_s \in \Gamma$. An *ordinary DPN* is a Monitor-DPN without monitor rules.

For the rest of this paper, we fix a Monitor-DPN $\mathscr{M} = (\mathrm{Act}, P, \Gamma, X, \Delta, (p_0, \gamma_0))$ and refer to Monitor-DPNs generally as DPNs. We define two semantics for DPNs, a lock-sensitive and a lock-insensitive one. To distinguish the threads in a configuration, we use thread identifiers consisting of sequences of natural numbers, which represent the way a thread was created. We denote by $\tau \in \mathbb{N}^*$ the empty sequence and by $t\,n$ the sequence t extended by n. Moreover, we extend the set of locks by a special *no lock* symbol to $\bar{X} = X \uplus \{\bullet\}$, where we write $\uplus$ for disjoint union. We assume that the sets P, Γ, X, Δ, and $\mathbb{N}^*$ are pairwise disjoint.

Definition 2. A thread configuration is a word from $P(\Gamma\bar{X})^*$. A DPN configuration $c \in 2^{\mathbb{N}^* \times P(\Gamma\bar{X})^*}$ is a finite set of pairs of thread identifiers and thread configurations. The initial configuration is $\{(\tau, p_0\gamma_0\bullet)\}$. The set of locks *held* by a DPN configuration c is $l(c) = \{x \in X \mid \exists t, w, w'.\ (t, wxw') \in c\}$ and the set of *active* threads is $c|_1 = \{t \in \mathbb{N}^* \mid \exists w.\ (t,w) \in c\}$. The lock-sensitive semantics of DPN is given by the following transition rules:

$$
\begin{array}{ll}
c \uplus \{(t, p\gamma yw)\} \xrightarrow{p\gamma \overset{a}{\hookrightarrow} p'\gamma'}_t & c \uplus \{(t, p'\gamma' yw)\} \\
c \uplus \{(t, p\gamma yw)\} \xrightarrow{p\gamma \overset{a}{\hookrightarrow} p'}_t & c \uplus \{(t, p'w)\} \\
c \uplus \{(t, p\gamma yw)\} \xrightarrow{p\gamma \overset{a}{\hookrightarrow} p'\gamma'\gamma_r}_t & c \uplus \{(t, p'\gamma' \bullet \gamma_r yw)\} \\
c \uplus \{(t, p\gamma yw)\} \xrightarrow{p\gamma \overset{a}{\hookrightarrow} p_s\gamma_s p'\gamma'}_t & c \uplus \{(t, p'\gamma' yw), (t\,n, p_s\gamma_s\bullet)\} \quad t\,n \notin c|_1 \wedge (n = 1 \vee t\,(n-1) \in c|_1) \\
c \uplus \{(t, p\gamma yw)\} \xrightarrow{p\gamma \overset{a,x}{\hookrightarrow} p'\gamma'\gamma_r}_t & c \uplus \{(t, p\gamma' x\gamma_r yw)\} \quad \text{if } x \notin l(c)\ .
\end{array}
$$

The lock-insensitive semantics for DPN is obtained by dropping the constraint $x \notin l(c)$ in the last rule. The set $\Pi(\mathscr{M})$ of executions of a DPN consists of all sequences π of the form $\pi = c_0 \xrightarrow{\eta_1}_{t_1} c_1 \ldots \xrightarrow{\eta_n}_{t_n} c_n$ where $c_0 = \{(\tau, p_0\gamma_0\bullet)\}$ and $\eta_i \in \Delta$. We denote by $\delta(\pi) = \{\eta_i \mid 1 \le i \le n\}$ the set of transitions used within execution π.

For a set C of DPN configurations and a set $\Delta' \subseteq \Delta$ of transition rules, the set of configurations that are lock sensitively reachable from C via Δ' is $\mathrm{lspost}^*_{\Delta'}(C) = \{c \mid \exists c_0 \in C, \pi.\ \pi = c_0 \xrightarrow{\eta_1}_{t_1} c_1 \ldots \xrightarrow{\eta_n}_{t_n} c \wedge \delta(\pi) \subseteq \Delta'\}$. We also define the lock-insensitive version $\mathrm{post}^*_{\Delta'}(C)$, which corresponds to the lock-insensitive semantics. The goal is to find a reasonable class of representations for sets of configurations such that $\mathrm{lspost}^*_{\Delta'}$ is a computable endomorphism on this class.

2.1 Execution Trees

Note that for all configurations c that are reached by executions from the initial configuration, the set of active threads $c|_1$ forms a tree and thread $t\,n$ is the thread spawned by the n-th spawn transition performed by thread t. If one denotes by π^t the sequence of transitions performed by t in π and builds a tree by making $\pi^{t\,n}$ the left branch of the n-th spawn appearing in π^t one obtains (up to projection to the annotated actions) what is called an action tree [9, 15]. An action tree reflects the constraints on the ordering of transitions to form a valid execution: A transition must be executed after its predecessors in the tree. In the lock-insensitive setting, these constraints completely characterize the valid executions, i.e., all topological orderings of the transitions according to the action tree are valid executions. In

the lock-sensitive setting, there are additional constraints imposed by the locks, which are explained in Section 2.2.

As the π^t correspond to traces of ordinary push-down systems, they are essentially equivalent to context free languages. One can regain (tree-)regularity by adding additional structure which makes the push-down behavior visible. This is done by matching return transitions to corresponding calls in the following way: First, we add for convenience a special node $p\gamma$ to the end of π^t, if the thread configuration reached by t has a non-empty stack, where p is the reached control state and γ is the top of stack. Then let $\pi^t = \eta_0 \dots \eta_i \eta_{i+1} \dots \eta_j \eta_{j+1} \dots \eta_k$ (here η_k may be the newly introduced node) and let η_i be the first call- or monitor-rule in π^t matched by a return-rule, say η_j. We transform π^t to the tree $\eta_0 \dots \eta_i(\eta_{i+1} \dots \eta_j , \eta_{j+1} \dots \eta_k)$ and continue recursively with the new subtraces $\eta_{i+1} \dots \eta_j$ and $\eta_{j+1} \dots \eta_k$. Here we represent trees by terms. If we again hook in the obtained trees as left branches of the corresponding spawn operations, we obtain the execution trees introduced by Gawlitza *et al.* [9]. The execution tree of an execution π is denoted by $人(\pi)$. Moreover, we define the execution tree corresponding to the empty execution to be the initial node $p_0\gamma_0$.

Figure 1 depicts an example how a Java program can be represented as a DPN and how, for a fixed execution, the execution tree is build. The flow graph model already models the synchronized blocks as procedures R and S. The lock x is bound to the call of those procedures. We assume that the call q.**start**() spawns a thread that executes procedure Q. On the bottom left an example execution which consists of the transition $\eta_{p_1}\eta_{q_1}\eta_{s_1}\eta_{p_2}\eta_{s_2}\eta_{p_3}\eta_{r_1}\eta_{r_2}$ is shown. On the right the construction of its execution tree via the action tree is illustrated. On the bottom right the execution tree is depicted in an extended notation which will be introduced a little later.

We also overload post* and lspost* to sets of execution trees:

Definition 3. The lock-sensitive successors of a set A of execution trees via $\Delta' \subseteq \Delta$ are

$$\text{lspost}^*_{\Delta'}(A) = \left\{ 人(\pi\pi') \mid \pi, \pi\pi' \in \Pi(\mathscr{M}),\ 人(\pi) \in A,\ \delta(\pi') \subseteq \Delta' \right\}$$

The lock-insensitive version $\text{post}^*_{\Delta'}$ is the one corresponding to the lock-insensitive semantics.

Note that different executions may produce the same execution tree, because the tree does not capture the relative order of the transitions of different threads completely. In Figure 1 the execution tree also corresponds to the four other lock-sensitive executions which reach the same configuration. An execution that produces a given execution tree is called a *schedule* of that tree. An execution tree which possesses a schedule is called *schedulable*. If it possesses a schedule in the lock-sensitive version of the semantics it is called lock-sensitively schedulable. The configuration reached by an execution can easily be reconstructed from the execution tree. We call the corresponding function conf and overload it to sets of execution trees. This allows us to state the following adequacy result for our definition of lspost* for execution trees (the analogous holds for post*):

Proposition 1. *For a reachable set of execution trees* $A \subseteq \text{lspost}^*_\Delta(\{p_0\gamma_0\})$ *and* $\Delta' \subseteq \Delta$ *it holds that*

$$\text{lspost}^*_{\Delta'}(\text{conf}(A)) = \text{conf}(\text{lspost}^*_{\Delta'}(A)) \,.$$

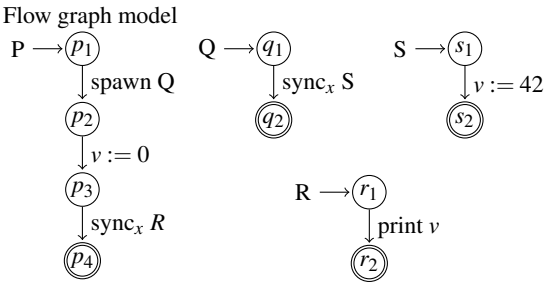
We introduce additional notations for the nodes of an execution tree to enhance readability and include some additional information. As noted earlier every node in an execution tree – except the added $p\gamma$ leafs – corresponds to a transition of the DPN. We denote a node which corresponds to a transition η of type *base* as BASE^η. A node corresponding to a *call* transition η with one successor (i.e. a call which has no matching return within the execution) is denoted as NCALL^η. If it has two successors (i.e. the call returns) it is denoted as RCALL^η. We denote by RET^η a node corresponding to a *return*

Java Program

```
void P() {
  q.start();
  v = 0;
  synchronized(x){
    print(v);
  }
}
```

```
void Q() {
  synchronized(x){
    v = 42;
  }
}
```

Flow graph model



DPN-model $\mathscr{M}' = (\{\#\}, \{p_i, q_j, s_j, r_j | i \in \{1,2,3,4\}, j \in \{1,2\}\}, \text{Act}, \{x\}, \Delta', (\#, p_1))$
$\text{Act} = \{spawn\ Q,\ v := 0,\ sync_xR,\ ret\ P,\ sync_xS,\ ret\ Q,\ print\ v,\ ret\ R,\ v := 42,\ ret\ S\}$

DPN execution π

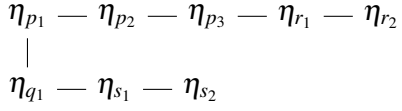
$\{(\tau, \#p_1\bullet)\}$
$\tau \downarrow \#p_1 \xrightarrow{spawn\ Q} \#q_1\#p_2$
$\{(\tau, \#p_2\bullet), (1, \#q_1\bullet)\}$
$1 \downarrow \#q_1 \xrightarrow{sync_xS,\ x} \#s_1q_2$
$\{(\tau, \#p_2\bullet), (1, \#s_1xq_2\bullet)\}$
$1 \downarrow \#s_1 \xrightarrow{v:=42} \#s_2$
$\{(\tau, \#p_2\bullet), (1, \#s_2xq_2\bullet)\}$
$\tau \downarrow \#p_2 \xrightarrow{v:=0} \#p_3$
$\{(\tau, \#p_3\bullet), (1, \#s_2xq_2\bullet)\}$
$1 \downarrow \#s_2 \xrightarrow{ret\ S} \#$
$\{(\tau, \#p_3\bullet), (1, \#q_2\bullet)\}$
$\tau \downarrow \#p_3 \xrightarrow{sync_xR,\ x} \#r_1p_4$
$\{(\tau, \#r_1xp_4\bullet), (1, \#q_2\bullet)\}$
$\tau \downarrow \#r_1 \xrightarrow{print\ v} \#r_2$
$\{(\tau, \#r_2xp_4\bullet), (1, \#q_2\bullet)\}$
$\tau \downarrow \#r_2 \xrightarrow{ret\ R} \#$
$\{(\tau, \#p_4\bullet), (1, \#q_2\bullet)\}$

DPN transitions Δ'

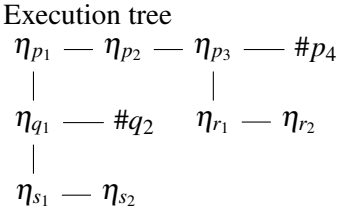
$\eta_{p_1} = \#p_1 \xrightarrow{spawn\ Q} \#q_1\#p_2$
$\eta_{p_2} = \#p_2 \xrightarrow{v:=0} \#p_3$
$\eta_{p_3} = \#p_3 \xrightarrow{sync_xR,\ x} \#r_1p_4$
$\eta_{p_4} = \#p_4 \xrightarrow{ret\ P} \#$
$\eta_{q_1} = \#q_1 \xrightarrow{sync_xS,\ x} \#s_1q_2$
$\eta_{q_2} = \#q_2 \xrightarrow{ret\ Q} \#$
$\eta_{r_1} = \#r_1 \xrightarrow{print\ v} \#r_2$
$\eta_{r_2} = \#r_2 \xrightarrow{ret\ R} \#$
$\eta_{s_1} = \#s_1 \xrightarrow{v:=42} \#s_2$
$\eta_{s_2} = \#s_2 \xrightarrow{ret\ S} \#$

$\pi^\tau = \eta_{p_1}\eta_{p_2}\eta_{p_3}\eta_{r_1}\eta_{r_2}$
$\pi^1 = \eta_{q_1}\eta_{s_1}\eta_{s_2}$

Action tree



Execution tree



Execution tree in extended notation

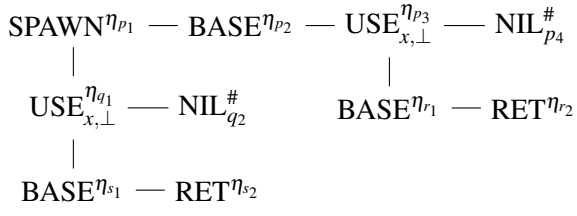


Figure 1: Construction of an Execution Tree

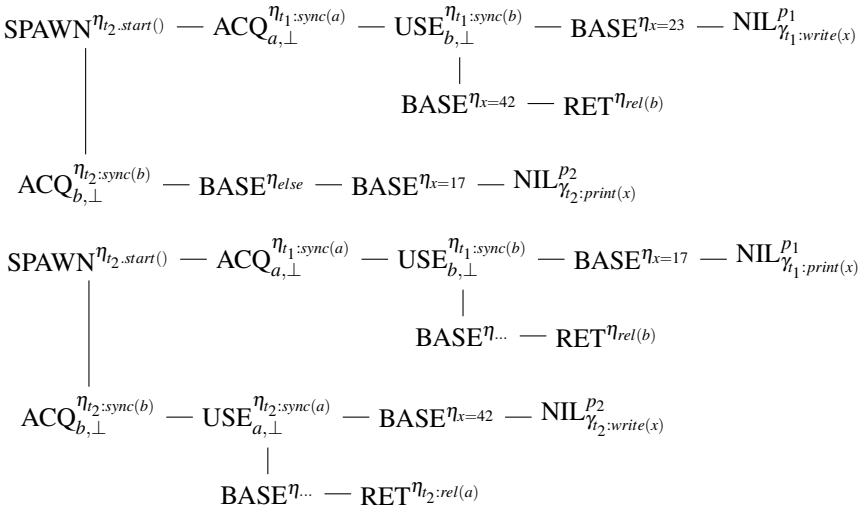


Figure 2: Two execution trees corresponding to the sixth and fifth example from Table 1.

transition η and by NIL^p_γ an inserted $p\gamma$ leaf. Nodes corresponding to *monitor* transitions η are denoted analogously to calls as $\mathrm{ACQ}^\eta_{x,r}$ (the lock is *finally acquired*, i.e., it remains acquired for the rest of the execution) or $\mathrm{USE}^\eta_{x,r}$ (the lock is *used*, i.e., it is released at some later point in the execution). The additional annotations x and r at monitor nodes denote the lock x and a bit $r \in \mathbb{B} = \{\bot, \top\}$ (interpreted as *false* and *true*) indicating whether the operation is reentrant, i.e., whether this lock is already acquired by the process when executing the transition. Nodes corresponding to *spawn* transitions η are always binary and denoted as SPAWN^η. The left branch corresponds to the execution of the newly spawned process and the right branch to the execution of the spawning one.

Figure 2 contains two examples of execution trees which correspond to an execution of some models of the sixth and fifth example from Table 1. The left child of each binary node grows downwards. The first execution tree is schedulable lock-sensitively but the use of lock b by the first thread must be scheduled before the other thread acquires lock b. The second execution tree is reachable only in the lock-insensitive version of the semantics, i.e., it is from the set $\mathrm{post}^*_\Delta(\{\mathrm{NIL}^{p_0}_{\gamma_0}\}) \setminus \mathrm{lspost}^*_\Delta(\{\mathrm{NIL}^{p_0}_{\gamma_0}\})$. Here the main thread wants to acquire a and then use b. The spawned thread wants to acquire b and then use a. Obviously this is not possible as one would unavoidably reach a deadlock.

Gawlitza *et al.* [9] showed that the set of reachable execution trees for DPN without locks is effectively tree-regular. We recap their construction, slightly modified by adding reentrance annotations, in Definition 4. Moreover, they used acquisition structures to obtain a regular characterization of reachable execution trees with (non-reentrant) well-nested locks and extended this even further to *joins* with spawned threads. We pick up this approach and extend it to reentrant locks in Proposition 4. After introducing the concept of a *cut* of an execution tree, we then use acquisition and release structures [15] to also obtain an iterable tree-regular characterization of reachable configurations. All this is done by defining appropriate finite tree automata.

A finite (bottom-up) tree automaton $\mathscr{T}$ over a ranked alphabet Σ is a triple (Q, Q_f, δ) consisting of finite sets of states Q, accepting states $Q_f \subseteq Q$, and rules δ of the form $f\ q_1 \dots q_n \to q$, where $f \in \Sigma$ is a symbol of rank n and $q, q_1, \dots, q_n \in Q$. If a tree automaton can recognize the trees $t_1, \dots, t_n$ with states $q_1, \dots, q_n$ respectively and it contains the above rule, it can recognize the tree $f(t_1, \dots, t_n)$ with state q. A tree is accepted if it can be recognized with an accepting state. The language $\mathscr{L}(\mathscr{T})$ is the

set of all trees accepted by the automaton. A set of trees is called tree-regular if it is the language of a finite tree automaton. The intersection of tree-regular sets is again tree-regular (accepted by the product automaton) and emptiness of the language of a given tree automaton is decidable in linear time. For an introduction to tree automata we refer the reader to Comon *et al.* [5]. When defining concrete automata we leave out sub- and superscripts of nodes which do not influence the automata and write an underscore to denote arbitrary values, e.g., NIL $\to$ _ denotes $\mathrm{NIL}^p_\gamma \to U$ for all p, γ and U.

Definition 4. For a Monitor-DPN $\mathscr{M}$ we define the tree automaton $\mathscr{T}_\mathscr{M}$, which is intended to accept the *lock-insensitive execution trees* of $\mathscr{M}$. The state space is $P \times \Gamma \times P \times \{\bot,\top\} \times 2^X$ and the accepting states are $\{p_0\} \times \{\gamma_0\} \times P \times \{\bot,\top\} \times \{\emptyset\}$. For all $\eta \in \Delta$, $t \in \{\bot,\top\}$, $p, p', p''.p''' \in P$, $\gamma, \gamma', \gamma'' \in \Gamma$ and $ls \subseteq X$ the automaton contains the following rules. We write $\frac{f\ q_1 \ldots q_n}{q}$ to denote the rule $f\ q_1 \ldots q_n \to q$.

$$\frac{}{(p,\gamma,p,\bot,ls)}\mathrm{NIL}^p_\gamma,$$

$$\frac{}{(p,\gamma,p',\top,ls)}\mathrm{RET}^\eta \quad \text{if } \eta = p\gamma \overset{a}{\hookrightarrow} p',$$

$$\frac{(p',\gamma',p'',t,ls)}{(p,\gamma,p'',t,ls)}\mathrm{BASE}^\eta \quad \text{if } \eta = p\gamma \overset{a}{\hookrightarrow} p'\gamma',$$

$$\frac{(p',\gamma',p'',\bot,ls)}{(p,\gamma,p'',\bot,ls)}\mathrm{NCALL}^\eta \quad \text{if } \eta = p\gamma \overset{a}{\hookrightarrow} p'\gamma'\gamma'',$$

$$\frac{(p',\gamma',p'',\top,ls)\ (p'',\gamma'',p''',t,ls)}{(p,\gamma,p''',t,ls)}\mathrm{RCALL}^\eta \quad \text{if } \eta = p\gamma \overset{a}{\hookrightarrow} p'\gamma'\gamma'',$$

$$\frac{(p',\gamma',\_,\_,\emptyset)\ (p'',\gamma'',p''',t,ls)}{(p,\gamma,p''',t,ls)}\mathrm{SPAWN}^\eta \quad \text{if } \eta = p\gamma \overset{a}{\hookrightarrow} p'\gamma' p''\gamma',$$

$$\frac{(p',\gamma',p'',\bot,ls\cup\{x\})}{(p,\gamma,p'',\bot,ls)}\mathrm{ACQ}^\eta_{x,r} \quad \text{if } \eta = p\gamma \overset{a,x}{\hookrightarrow} p'\gamma'\gamma'',\ r = \top \Leftrightarrow x \in ls,$$

$$\frac{(p',\gamma',p'',\top,ls\cup\{x\})\ (p'',\gamma'',p''',t,ls)}{(p,\gamma,p''',t,ls)}\mathrm{USE}^\eta_{x,r} \quad \text{if } \eta = p\gamma \overset{a,x}{\hookrightarrow} p'\gamma'\gamma'',\ r = \top \Leftrightarrow x \in ls.$$

For a tree recognized by this automaton the corresponding state records in the first two places the control state and top of stack with which the execution started. That is the head of the rule corresponding to the root node. The third place records the control state that has been reached by the main thread with its last action, that is the control state annotated at the right-most NIL or RET node. Whether this last action of the main thread is a RET node or not is marked in the forth place by $\top$ or $\bot$ respectively. Moreover, it contains a set of locks in the fifth place, which is used to check the marking of reentrant lock operations. That is, it calculates for each node which locks the thread holds upon the execution of the corresponding transition. In the lock-insensitive setting this tree automaton already characterizes the set of reachable execution trees:

Proposition 2. $\mathscr{L}(\mathscr{T}_\mathscr{M}) = \mathrm{post}^*_\Delta(\{\mathrm{NIL}^{p_0}_{\gamma_0}\})$.

To obtain $\mathrm{post}^*_{\Delta'}$ for arbitrary $\Delta' \subseteq \Delta$ one can utilize the following simple tree automaton with a single state to check that only transition from Δ' are used:

Definition 5. The tree automaton $\mathscr{T}_{\Delta'}$ has the single state $\bullet$ (which is accepting) and the following rules for all $\eta \in \Delta'$:

$$\mathrm{NIL} \to \bullet \qquad \mathrm{RET}^\eta \to \bullet \qquad \mathrm{NCALL}^\eta\ \bullet \to \bullet \qquad \mathrm{ACQ}^\eta\ \bullet \to \bullet$$
$$\mathrm{RCALL}^\eta\ \bullet\ \bullet \to \bullet \qquad \mathrm{USE}^\eta\ \bullet\ \bullet \to \bullet \qquad \mathrm{SPAWN}^\eta\ \bullet\ \bullet \to \bullet$$

Proposition 3. $\mathscr{L}(\mathscr{T}_\mathscr{M}) \cap \mathscr{L}(\mathscr{T}_{\Delta'}) = \mathrm{post}^*_{\Delta'}(\{\mathrm{NIL}^{p_0}_{\gamma_0}\})$.

2.2 Lock-Sensitive Analysis

As $\mathrm{lspost}^*_{\Delta'}(\{\mathrm{NIL}^{p_0}_{\gamma_0}\}) \subseteq \mathrm{post}^*_{\Delta'}(\{\mathrm{NIL}^{p_0}_{\gamma_0}\})$ one can obtain the set of lock-sensitively reachable execution trees by filtering out those which do not have a lock-sensitive schedule. This is done by the acquisition structure, which precisely models the restrictions to a possible schedule imposed by the final acquisitions and uses of locks. Only non-reentrant actions need to be considered: Reentrant actions can always be executed, and their execution cannot inhibit execution of other threads.

Definition 6. The tree automaton $\mathscr{T}_{ah}$ accepts from the set of lock-insensitively reachable execution trees those which possess a lock-sensitive schedule. The state space is $2^X \times 2^X \times 2^{X\times X}$ with accepting states $2^X \times 2^X \times \{G \in 2^{X\times X} \mid G \text{ is acyclic}\}$. We write sets of nodes to denote a rule for each node of the given set. The rules are as follows for all $x \in X$:

$$\mathrm{NIL} \to (\emptyset,\emptyset,\emptyset) \quad \mathrm{RET} \to (\emptyset,\emptyset,\emptyset)$$

$$\mathrm{BASE}\ \alpha \to \alpha \quad \mathrm{NCALL}\ \alpha \to \alpha \quad \mathrm{ACQ}_{x,\top}\ \alpha \to \alpha$$

$$\frac{\{\mathrm{RCALL}, \mathrm{USE}_{x,\top}, \mathrm{SPAWN}\}\ (A,U,G)\ (A',U',G')}{(A\cup A',\ U\cup U',\ G\cup G')}\ \text{if } A\cap A' = \emptyset$$

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

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

The first component of the state of this tree automaton represents the set of locks which are finally acquired non-reentrantly within the subtree, that means, there is a corresponding ACQ-node. It is used to assert that (1) no lock is finally acquired twice, which is achieved by the side conditions of the above rules. The second component represents all locks which are used (non-reentrantly, including final acquisitions) in the subtree. The third component is the *acquisition graph*, which represents order constraints imposed by the program order. It contains an edge $x \to y$ if in any schedule the lock y must be used or acquired non-reentrantly after the lock x has been finally acquired non-reentrantly. The acceptance condition ensures that (2) the acquisition graph is acyclic. Criteria (1) and (2) suffice to precisely characterize the set of lock-sensitively schedulable execution trees [15]:

Proposition 4. $\mathscr{L}(\mathscr{T}_{\mathscr{M}}) \cap \mathscr{L}(\mathscr{T}_{ah}) = \mathrm{lspost}^*_{\Delta}(\{\mathrm{NIL}^{p_0}_{\gamma_0}\})$.

The corresponding version restricted to a set $\Delta' \subseteq \Delta$ of allowed transition is:

Proposition 5. $\mathscr{L}(\mathscr{T}_{\mathscr{M}}) \cap \mathscr{L}(\mathscr{T}_{ah}) \cap \mathscr{L}(\mathscr{T}_{\Delta'}) = \mathrm{lspost}^*_{\Delta'}(\{\mathrm{NIL}^{p_0}_{\gamma_0}\})$.

One easily sees that if a tree is not accepted by the tree automaton $\mathscr{T}_{ah}$ it cannot be scheduled: Either some lock is finally acquired twice by different threads or there exists a cycle in the acquisition graph corresponding to an unavoidable deadlock. On the other hand, if a tree is accepted one can show that it has a schedule. In this schedule the final acquisitions are executed as late as possible in a topological ordering of the acyclic acquisition graph and the uses of locks are scheduled atomically between the final acquisitions. The latter is possible due to the requirements captured in the acquisition graph and as a thread holds the same locks before and after a use due to well-nestedness [9].

The language $\mathscr{L}(\mathscr{T}_{\mathscr{M}}) \cap \mathscr{L}(\mathscr{T}_{ah})$ is the language of the product automaton of $\mathscr{T}_{\mathscr{M}}$ and $\mathscr{T}_{ah}$. This can be used to answer reachability questions for tree-regular sets of configurations by checking their intersection for emptiness.

One may ask why we used reentrance annotations instead of calculating the required information directly in the last automaton. Note that the first tree automaton ($\mathscr{T}_{\mathscr{M}}$) is deterministic when evaluated top down and the second one ($\mathscr{T}_{ah}$) when evaluated bottom up, but both are extremely nondeterministic

when evaluated in the opposite direction. We exploit this with a specialized emptiness check which evaluates one automaton only bottom up and the other top down. We will report on this in Section 3.2.2. This determinism of $\mathscr{T}_{ah}$ would be lost if it would calculate reentrance information.

Example 1. A simple application is calculation of may-happen-in-parallel information. That is, given two sets R and W of stack symbols, is it possible that two threads simultaneously reach a topmost stack symbol of the corresponding sets? The relevant execution trees are accepted by the tree automaton $\mathscr{T}_{rw}$ with state space $2^{\{r,w\}}$, single accepting state $\{r,w\}$ and the following rules:

$$\mathrm{NIL}_\gamma \to \{r\} \text{ for } \gamma \in R \qquad \mathrm{NIL}_\gamma \to \{w\} \text{ for } \gamma \in W \qquad \mathrm{NIL}_\gamma \to \emptyset \text{ for } \gamma \notin (R \cup W)$$
$$\mathrm{RET} \to \emptyset \quad \{\mathrm{ACQ}, \mathrm{BASE}, \mathrm{NCALL}\}\ \alpha \to \alpha$$
$$\{\mathrm{RCALL}, \mathrm{SPAWN}, \mathrm{USE}\}\ \alpha\ \beta \to \alpha \cup \beta$$

By checking whether $\mathscr{L}(\mathscr{T}_{\mathscr{M}}) \cap \mathscr{L}(\mathscr{T}_{ah}) \cap \mathscr{L}(\mathscr{T}_{rw}) \stackrel{?}{=} \emptyset$ one can prove the absence of a data race on some variable if R represents the set of stack symbols where the variable is read or written and W the set of stack symbols where the variable is written. This rules out all the spurious data races in the examples in Table 1, i.e., all except the two actual races between the assignment x = 23 by the first and the accesses to x by the other thread in the sixth example. A schedulable execution tree witness is depicted in the upper part of Figure 2.

2.3 Iterable Analysis

The hitherto presented techniques allow only to check for reachability from the initial configuration of the Monitor-DPN. We now extend this technique to answer reachability queries starting from arbitrary tree-regular sets of reachable execution trees. We obtain a tree-regular representation of $\mathrm{lspost}^*_{\Delta'}(A)$ for arbitrary tree-regular $A \subseteq \mathrm{lspost}^*_{\Delta}(\{\mathrm{NIL}^{p_0}_{\gamma_0}\})$ and $\Delta' \subseteq \Delta$. This in particular allows us to answer iterated reachability questions: Given a sequence $A_1, \ldots, A_n$ of regular sets of execution trees and sets of DPN transitions $\Delta_1, \ldots, \Delta_n \subseteq \Delta$, we compute $A_n \cap \mathrm{lspost}^*_{\Delta_n}(A_{n-1} \cap \mathrm{lspost}^*_{\Delta_{n-1}}(\ldots A_1 \cap \mathrm{lspost}^*_{\Delta_1}(\{\mathrm{NIL}^{p_0}_{\gamma_0}\})\ldots))$ and check it for emptiness, in order to decide whether there is an execution visiting the set $A_1, \ldots, A_n$ in the given order using the given transitions.

In order to compute $\mathrm{lspost}^*_{\Delta'}(A)$, we exploit the fact that all execution trees of prefixes of an execution are closely related to prefixes of the final tree: They are obtained by cutting the final tree, replacing subtrees by NIL-nodes and changing some returning calls or uses to non-returning calls or acquisitions as necessary. Note however, that in the presence of locking not all prefixes of an execution tree correspond to prefixes of its schedules. We use an additional type of node (a CUT-node) to *mark* an intermediate configuration in the tree. For each thread that existed at the intermediate configuration, a cut node marks the position after its last step before reaching the intermediate configuration.

In a first step we characterize the execution-trees which are wellformed with respect to the placement of cut nodes. In order to avoid several additional case distinctions we make the assumption that no thread may empty its stack, that is the corresponding procedure with which its execution started may not perform a return action. With this assumption, in the lock-insensitive setting, it suffices to check that each thread spawned *before* the cut contains exactly one cut node and that no thread spawned afterwards contains any cuts. This can easily be checked by a tree automaton. To obtain the regular representation we first extend $\mathscr{T}_{\mathscr{M}}$ by the rule $\mathrm{CUT}^p_\gamma\ (p,\gamma,p',t,ls) \to (p,\gamma,p',t,ls)$. This allows arbitrary (well-annotated) cut-nodes to be inserted in the execution tree. The following tree automaton checks that the inserted cut nodes mark a legit intermediate configuration.

Definition 7. The cut wellformed trees are accepted by the tree automaton $\mathscr{T}_{cwf}$ which has the state space $\{\bot, \top, \overline{\$}, \underline{\$}\}$, accepting state $\top$ and the following rules. We write sets of states to denote a corresponding rule for every state of the given set. We leave out the rules for USE/ACQ nodes as these are the same as for call nodes.

$$\begin{array}{rrr}
\text{RCALL } \{\bot,\overline{\$}\}\ \top \to \top & \text{RCALL } \top\ \{\bot,\underline{\$}\} \to \top & \text{NIL} \to \bot \\
\text{RCALL } \overline{\$}\ \{\bot,\overline{\$}\} \to \overline{\$} & \text{RCALL } \underline{\$}\ \{\bot,\underline{\$}\} \to \underline{\$} & \text{RET} \to \bot \\
\text{RCALL } \bot\ \overline{\$} \to \overline{\$} & \text{RCALL } \bot\ \underline{\$} \to \underline{\$} & \text{BASE } q \to q \\
\text{RCALL } \bot\ \bot \to \bot & \text{SPAWN } \top\ \{\bot,\overline{\$}\} \to \overline{\$} & \text{NCALL } q \to q \\
\text{SPAWN } \{\bot,\underline{\$}\}\ \{\bot,\underline{\$}\} \to \underline{\$} & \text{SPAWN } \top\ \top \to \top & \text{CUT } \{\bot,\underline{\$}\} \to \top
\end{array}$$

The intuition behind the rules is as follows: A tree recognized with the accepting state $\top$ is cut-wellformed. A tree recognized with $\bot$ does not contain any cut or spawn nodes. A tree recognized with state $\overline{\$}$ contains some spawn nodes and the trees corresponding to the spawned threads are cut-wellformed, but the main thread of the tree is missing its cut node. In this case the tree must be in the left (returning) branch of a returning call/use and the corresponding cut node must be in the right branch. Finally $\underline{\$}$ corresponds to a tree which contains spawn nodes but no cut nodes. This requires that it is located *after* the cut. By careful inspection of the individual cases one can see that the above tree automaton calculates the correct information.

Based on a cut-wellformed execution tree we need to check whether the execution tree corresponding to the execution marked by the cut is from the given tree-regular set for which we want to calculate post*. We can do this by using a tree transducer to obtain the execution tree in question. A tree transducer can be seen as a tree automaton with output. The rules have the form $f\ q_1(t_1)\dots q_n(t_n) \to q(f'\ t_{i_1}\dots t_{i_k})$. The semantics is similar to the one for tree automata except that the states are augmented with an output which is inductively constructed from the output of the subtrees in each rule. It is known that the inverse image of a tree-regular set under a tree transducer is again tree-regular (Engelfriet [6]).

Definition 8. The tree transducer $\mathscr{T}_{ct}$ with state space $\{\bot, \top\}$, accepting state $\top$ and the following rules obtains the execution tree marked by the cut from a well-cut execution tree.

$$\begin{array}{rr}
\text{NIL}^p_\gamma \to \bot(\text{NIL}^p_\gamma) & \text{RET}^\eta \to \bot(\text{RET}^\eta) \\
\text{BASE}^\eta\ q(w) \to q(\text{BASE}^\eta\ w) & \text{NCALL}^\eta\ q(w) \to q(\text{NCALL}^\eta\ w) \\
\text{RCALL}^\eta\ \top(c)\ \bot(_) \to \top(\text{NCALL}^\eta\ c) & \text{RCALL}^\eta\ \bot(c)\ \bot(t) \to \bot(\text{RCALL}^\eta\ c\ t)) \\
\text{ACQ}^\eta_{x,r}\ q(w) \to q(\text{ACQ}^\eta_{x,r}\ w) & \text{USE}^\eta_{x,r}\ \top(c)\ \bot(_) \to \top(\text{ACQ}^\eta_{x,r}\ c) \\
\text{USE}^\eta_{x,r}\ \bot(c)\ \bot(t) \to \bot(\text{USE}^\eta_{x,r}\ c\ t)) & \text{SPAWN}^\eta\ _(c)\ q(t) \to q(\text{SPAWN}^\eta\ c\ t) \\
\text{CUT}^p_\gamma\ \bot(_) \to \top(\text{NIL}^p_\gamma) &
\end{array}$$

Proposition 6. *For tree-regular $A \subseteq \text{post}^*_\Delta(\{\text{NIL}^{p_0}_{\gamma_0}\})$ it holds that*

$$\text{post}^*_\Delta(A) = (\mathscr{L}(\mathscr{T}_\mathscr{M}) \cap \mathscr{L}(\mathscr{T}_{cwf}) \cap \mathscr{T}_{ct}^{-1}(A))|_{\text{CUT}}\ .$$

Here, $|_{\text{CUT}}$ denotes the projection which removes all cut nodes from the tree which can be done by a simple tree transducer.

In order to compute $\text{post}^*_{\Delta'}(A)$ for arbitrary $\Delta' \subseteq \Delta$ we again use a simple tree automaton on the set of execution trees with cut nodes, which checks that only rules from Δ' are used *after* the cut.

Definition 9. For a set $\Delta' \subseteq \Delta$ the tree automaton $\mathscr{T}^c_{\Delta'}$ checks that only rules from Δ' are used *after* the cut. The state space is $\{\bot, +, \top\}$ and $\top$ is the only accepting state. For the definition of the rules let η

range over Δ, η_g range over Δ', and η_b range over $\Delta \setminus \Delta'$. The rules are as follows:

$$\begin{array}{rrr}
\text{BASE}^{\eta_g}\ q \to q & \text{BASE}^{\eta_b}\ \{\bot,+\} \to + & \text{BASE}^{\eta_b}\ \top \to \top \\
\text{RCALL}^{\eta}\ \top\ \bot \to \top & \text{RCALL}^{\eta}\ \{\bot,+\}\ \top \to \top & \\
\{\text{SPAWN}^{\eta_b}, \text{RCALL}^{\eta_b}\}\ \bot\ \bot \to + & \{\text{SPAWN}^{\eta_g}, \text{RCALL}^{\eta_g}\}\ \bot\ \bot \to \bot & \\
\{\text{SPAWN}^{\eta}, \text{RCALL}^{\eta}\}\ +\ \{\bot,+\} \to + & \{\text{SPAWN}^{\eta}, \text{RCALL}^{\eta}\}\ \bot\ + \to + & \\
\text{SPAWN}^{\eta_g}\ \top\ \bot \to \bot & \text{SPAWN}^{\eta_b}\ \top\ \bot \to + & \text{SPAWN}^{\eta}\ \top\ \top \to \top \\
\text{SPAWN}^{\eta}\ \top\ + \to + & \text{CUT}\ \bot \to \top &
\end{array}$$

The rules for ACQ and NCALL-nodes are the same as for BASE-nodes and the rules for USE-nodes are the same as those for RCALL-nodes.

Here the state $\bot$ indicates that there is neither a cut node in the main thread nor a transition that is not from Δ' in any thread. The state $+$ indicates that there is a transition not from Δ' which may lay below the cut as there is no cut node in the main thread. Therefore at a cut node only $\bot$ is accepted and the state changes to $\top$ which indicates that there is a cut in the main thread and afterwards only actions from Δ' are used. As the left branch of a returning call is executed before the right branch, if there is a cut node in the left branch of such a call, then all transitions in the right branch must be from Δ'. Thus, only $\bot$ is accepted for the right branch.

Proposition 7. *For tree-regular $A \subseteq \text{post}^*_\Delta(\{\text{NIL}^{p_0}_{\gamma_0}\})$ and $\Delta' \subseteq \Delta$ it holds that*

$$\text{post}^*_{\Delta'}(A) = (\mathscr{L}(\mathscr{T}_{\mathscr{M}}) \cap \mathscr{L}(\mathscr{T}_{cwf}) \cap \mathscr{T}_{ct}^{-1}(A) \cap \mathscr{L}(\mathscr{T}^c_{\Delta'}))|_{\text{CUT}}\ .$$

2.4 Iterable Lock-Sensitive Analysis

In the lock-sensitive case, it remains to be checked whether the final configuration can be reached lock-sensitively from the intermediate configuration marked by the cut. This clearly requires that the final configuration is lock-sensitively reachable. Note that we assume that the intermediate configuration is already lock-sensitively reachable. Yet this is not sufficient, as reaching the cut may introduce additional constraints. Two prototypical examples are shown in Figure 3, where the dashed lines illustrate how cut nodes divide an execution tree into two parts. In both trees the intermediate configuration marked by the cut and the final configuration are lock-sensitively reachable, but the final configuration is not lock-sensitively reachable from the intermediate one. To obtain a sufficient criterion we additionally use *release structures*, which, analogously to acquisition structures for acquisitions, model restrictions for releasing held locks [15]. Firstly, one must ensure that all locks used at the cut can actually be released afterwards. This is not always possible as releasing one lock may require using another one first as depicted in the right part of Figure 3. Secondly, one must check that no lock finally acquired before the cut is used by another thread after the cut, as depicted in the left part of Figure 3. A difference to the earlier approach [15] is that there the acquisition graph is computed independently for each phase, while we check the acquisition graph of the whole execution tree. It is easy to see that this approach is equivalent under the assumption that the previous phases are reachable lock sensitively and that no lock finally acquired before the cut is used afterwards. The reason for this approach is that it is easier for a tree automaton to calculate this information because it does not need to make additional distinctions whether it is above or below the cut. Moreover we can reuse the above presented tree automaton $\mathscr{T}_{ah}$ by simply ignoring the cut nodes, that is we add the rule $\text{CUT}\, q \to q$. The additional constraints can be checked with the automaton given in Definition 10 and 11.

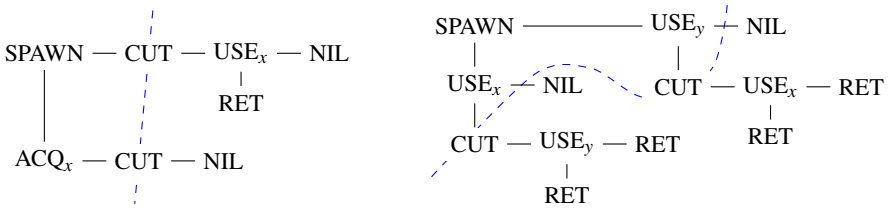


Figure 3: Not lock-sensitively schedulable execution trees with cut.

Definition 10. The release structure is computed by the following tree automaton $\mathscr{T}_{rh}$ with state space $\mathbb{B} \times 2^X \times 2^{X \times X}$, accepting states $\mathbb{B} \times 2^X \times \{G \in 2^{X \times X} \mid G \text{ is acyclic}\}$, and transitions:

$$\begin{aligned}
\text{NIL} &\to (\bot, \emptyset, \emptyset) & \text{RET} &\to (\bot, \emptyset, \emptyset) \\
\text{BASE } p &\to p & \text{NCALL } p &\to p \\
\text{ACQ}_{x,_}\ p &\to p & \text{CUT } (\bot, U, G) &\to (\top, U, G) \\
\text{RCALL } (C, U_c, G_c)\ (\bot, U_r, G_r) &\to (C, U_c \cup U_r, G_c \cup G_r) \\
\text{RCALL } (\bot, _, G_c)\ (\top, U_r, G_r) &\to (\top, U_r, G_c \cup G_r) \\
\text{USE}_{x,\top}\ (C, U_c, G_c)\ (\bot, U_r, G_r) &\to (C, U_c \cup U_r, G_c \cup G_r) \\
\text{USE}_{x,_}\ (\bot, _, G_c)\ (\top, U_r, G_r) &\to (\top, U_r, G_c \cup G_r) \\
\text{USE}_{x,\bot}\ (\bot, U_c, G_c)\ (\bot, U_r, G_r) &\to (\bot, U_c \cup U_r \cup \{x\}, G_c \cup G_r) \\
\text{USE}_{x,\bot}\ (\top, U_c, G_c)\ (\bot, U_r, G_r) &\to (\top, U_c \cup U_r, G_c \cup G_r \cup \{(y,x) \mid y \in U_c\}) \\
\text{SPAWN } (_, _, G_s)\ (C, U_r, G_r) &\to (C, U_r, G_s \cup G_r)
\end{aligned}$$

In this automaton the bit in the first place is set to top once the main thread of the tree contains a cut node. The lock-set in the second place contains the locks which are used by the main thread of the subtree after the cut. The lock-set is discarded if it becomes apparent that the subtree lies over the cut. The release-graph in the third place contains an edge from y to x if there is a use of x inside which there is the cut and, after the cut, a use of y. This means that y needs to be available before x can be released.

Definition 11. The tree automaton $\mathscr{T}_{ht}$ checks that no lock finally acquired before the cut is used afterwards. It has the state space $\mathbb{B} \times 2^X \times 2^X$. The accepting states are $\{(C, A, U) \in \mathbb{B} \times 2^X \times 2^X \mid A \cap U = \emptyset\}$. The rules are the following. Reentrant use nodes are handled in the same way as returning calls.

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

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

Again, the bit in the first position of a state indicates whether the main thread has seen a cut. The first lock-set represents the locks that are finally acquired before the cut. The second lock-set represents the locks used or finally acquired after the cut similar to those in the previous automaton, except that we include uses from spawned threads.

Using these automata we can characterize the set of execution trees that are lock-sensitively reachable from a regular set of lock-sensitively reachable execution trees:

Proposition 8. *For tree-regular* $A \subseteq \mathrm{lspost}^*_\Delta(\{\mathrm{NIL}^{p_0}_{\gamma_0}\})$ *and* $\Delta' \subseteq \Delta$ *it holds that*

$$\mathrm{lspost}^*_{\Delta'}(A) = (\mathscr{L}(\mathscr{T}_{\mathscr{M}}) \cap \mathscr{L}(\mathscr{T}_{rh}) \cap \mathscr{L}(\mathscr{T}_{ht}) \cap \mathscr{L}(\mathscr{T}_{ah}) \cap \mathscr{L}(\mathscr{T}_{cwf}) \cap \mathscr{T}_{ct}^{-1}(A) \cap \mathscr{L}(\mathscr{T}^c_{\Delta'}))|_{\mathrm{CUT}} .$$

After assuring oneself that the stated tree automata calculate the correct information, firstly one needs to understand that each accepted tree is schedulable in the desired way. Again we need to consider only non-reentrant operations. In a first phase, after reaching the intermediate configuration, the *open* uses which contain a cut are *closed*. This can be done based on a topological ordering of the acyclic release graph, because for an open use of lock x only the predecessors of x in the release graph are needed to close the use, and because the tree automaton $\mathscr{T}_{ht}$ assures that none of these locks are finally acquired before the cut. Again, the well-nestedness of lock operations allows to schedule uses atomically. Afterwards, in a second phase, we schedule the rest of the tree: We first schedule all uses, such that every thread is either in its final state, or before a final acquisition. This is possible, as none of the used locks has been finally acquired before the cut. Then, we schedule a final acquisition of a lock that is minimal in the acquisition graph. As the acquisition graph is acyclic, we always find such a lock. Moreover, the construction of the acquisition graph guarantees that this minimal lock is not required to schedule the rest of the tree. After the final acquisition, we schedule all subsequent uses. We then continue with the next final acquisition, until all threads have reached their final states.

Secondly, in order to show that the criteria checked by the tree automata are necessary for a schedule to exist (again, reentrant operations can be ignored), note that a schedulable tree obviously contains no two final acquisitions of the same lock, nor a final acquisition of a lock before the cut, that is used after the cut. Moreover, the acquisition graph captures ordering constraints between the final acquisitions and uses: An edge $x \to y$ means that the final acquisition of x must be scheduled before a use of y. Also, all uses of y must be scheduled before a potential final acquisition of y. Thus, a cycle in the acquisition graph indicates an unavoidable deadlock for any schedule of the tree. Symmetrically, an edge $x \to y$ in the release graph indicates that, after the intermediate configuration has been reached, lock x must be used before an open use of y can be closed. As an open use of a lock must be closed before the lock can again be used, a cycle in the release graph indicates an unavoidable deadlock after the intermediate configuration has been reached.

Example 2. An example application for this technique is solving forward bit vector problems bitwise. Assume there is a set $G \subseteq \Gamma$ of stack symbols corresponding to program points where the bit in question has just been generated. For simplicity we assume that generating the bit does not use any locks and does not spawn new threads. The tree automaton $\mathscr{T}_G$ checks that some thread has just reached a top of stack symbol from G. The state space is $\mathbb{B}$, $\top$ is the accepting state and the rules are:

$$\begin{aligned}
\mathrm{NIL}_\gamma &\to \top \quad \textbf{if } \gamma \in G & \qquad \mathrm{NIL}_\gamma &\to \bot \quad \textbf{if } \gamma \notin G \\
\mathrm{RET} &\to \bot & \mathrm{CUT}\ p &\to p \\
\mathrm{BASE}\ p &\to p & \mathrm{ACQ}\ p &\to p \\
\mathrm{NCALL}\ p &\to p & \mathrm{RCALL}\ p\ q &\to p \vee q \\
\mathrm{USE}\ p\ q &\to p \vee q & \mathrm{SPAWN}\ p\ q &\to p \vee q
\end{aligned}$$

Here we use $\vee$ to denote the disjunction of the states interpreted as $true$ and $false$. Let $\Delta' \subseteq \Delta$ be the set of transitions which do not kill the bit. Then, the bit reaches a program point corresponding to a set of

stack symbols P iff $\mathrm{lspost}^*_{\Delta'}(\mathrm{lspost}^*_{\Delta}(\{\mathrm{NIL}^{p_0}_{\gamma_0}\}) \cap \mathscr{L}(\mathscr{T}_G)) \cap \mathscr{L}(\mathscr{T}_P) = \emptyset$. Here $\mathscr{T}_P$ is defined analogously to $\mathscr{T}_G$ with P instead of G. This allows, for instance, to calculate lock-sensitive def-use dependencies, which we use in our application.

Note that instead of projecting out the cut-nodes it is convenient to leave them in the execution tree as they carry additional information which can be useful if one wants to directly design a tree automaton which checks for some property. One then annotates the cut-nodes with an index corresponding to their *level*. This is done in our actual implementation.

3 Analysis of Java Programs

3.1 Undecidability in the Presence of `wait`-Calls

Monitor-DPNs assume that lock usages can be bound to procedure calls which requires that the locks are used in a syntactically well-nested fashion. It seems that, by the very nature of block structure, locks are used in a well-nested fashion in synchronized-blocks and -methods in Java. However, this is no longer true in presence of calls to the `wait`-method of an object used as a lock. Once the `wait`-method is called on a lock, the corresponding lock is released, independently of whether it is the lock that has been acquired last or not. This clearly breaks well-nestedness. Note that, due to reentrance, well-nestedness can be broken even if the `wait`-method is called only on the lock of the innermost enclosing synchronized block or method, that is even if locking is syntactically well-nested. Ignoring this effect would render our analysis unsound as it would consider too few schedules.

Unfortunately there is no hope to handle this construct completely in the presence of recursion. We show now that reachability is undecidable in presence of `wait`-calls. As done by Kahlon *et al.* [13] for general non-well-nested locking it suffices to simulate pairwise rendezvous. Based on this one can reduce the undecidable emptiness problem for the intersection of context free languages to a reachability problem as done by Ramalingam [19]. We need three auxiliary locks and one lock for each symbol to be communicated. Initially, the first thread holds the locks $\{x_0, \ldots x_n, t\}$ and the other $\{s, r\}$. The following code models the synchronous communication of the symbol associated with x_i:

$$x_i\,.\,\mathrm{wait}\,(1)\,;\ \textbf{synchronized}\,(r)\,\{\,\}\,;\ \textbf{synchronized}\,(s)\,\{\,\}\,;t\,.\,\mathrm{wait}\,(1)\,;$$
$$\textbf{synchronized}\,(x_i)\,\{\,\}\,;r\,.\,\mathrm{wait}\,(1)\,;s\,.\,\mathrm{wait}\,(1)\,;\ \textbf{synchronized}\,(t)\,\{\,\}\,;$$

These lines can only be executed in lock step as each thread relies on the other thread offering its lock. Each thread needs to hold two locks so that no thread can step through its block multiple times while the other offers a lock. The initial configuration in which both threads hold the appropriate locks can easily be enforced with two additional locks and the following code at the beginning. Here at points A and B the actual code is inserted and the points directly afterwards are checked for parallel reachability. We use $\textbf{sync}(l_0 \ldots l_n)\{\mathrm{S}\}$ to abbreviate $\textbf{synchronized}(l_0)\{\ldots\textbf{synchronized}(l_n)\{\mathrm{S}\}\ldots\}$.

$$\textbf{sync}\,(a, x_0, \ldots x_n, t)\,\{\,\textbf{synchronized}\,(\mathrm{b})\,\{\,\}\,;\mathrm{a}\,.\,\mathrm{wait}\,(1)\,;\quad /{*}A{*}/\,\}$$
$$\textbf{sync}\,(b, s, r)\,\{\,\mathrm{b}\,.\,\mathrm{wait}\,(1)\,;\ \textbf{synchronized}\,(\mathrm{a})\,\{\,\}\,;\quad /{*}B{*}/\,\}$$

3.2 Implementation

We implemented our approach for the analysis of concurrent Java programs. The analysis of a Java program consists of two phases. In the first phase a DPN-model of the program under analysis is generated and in the second phase this model is analysed using the described techniques.

3.2.1 Model Generation

For model generation we utilize the T.J. Watson Libraries for Analysis (WALA)[1] targeting Java-Bytecode. WALA provides control flow and points-to information in the form of control flow, call and heap graphs. Following Esparza and Knoop [7], we encoded the local state in the stack symbols of the DPN, which consist, in the most basic setting, only of the control point of the program. The control states of the DPN are used to model the thread state (e.g. thrown exceptions) and return information of synchronized blocks as these are modeled as multi-exit procedures. Modeling exceptions precisely is important as they can be thrown by most Java-instructions. If one would abstract from exceptions (modeling them as regular control flow) the DPN could for instance skip uses of locks in called procedures by using abstracted exceptional control flow. By explicitly modeling the exceptional flow, we avoid that in the DPN critical parts are reachable spuriously after using exceptional control flow.

In order to model lock operations in the DPN, in a first step a finite set of possible locks has to be identified. As locks are bound to dynamically allocated objects in Java, we identify statically unique objects in the points-to information provided by WALA with a simple reachability analysis on the call and control flow graphs. That is, we search for abstract objects for which we can guarantee that there will be at most one instance at runtime because either the associated allocation instruction can be executed at most once, or they represent a class object. Moreover, we implemented a version of the random isolation technique from Kidd *et al.* [14]. This technique *randomly isolates* one instance of a class of objects and by proving the desired property for the isolated instance concludes to the validity of the property for all instances. This is possible if the property to be checked is, in some sense, coupled to an instance, for example, a data race or def/use dependency that occurs on a field of a specific instance. This is useful if, for instance, access to a field is guarded by the object it belongs to. Then, one can assume that either two threads use the same lock object, or they will access different fields. While Kidd *et al.* implemented random isolation as a source code transformation, we integrated this technique directly into the DPN-model, which can easily keep track of the needed information in its local state. As already mentioned, one needs to take care of potential `wait`-calls in the program. We use a simple data-flow analysis to statically identify monitors inside which such a call may occur and approximate these as regular procedure calls in the DPN-model.

The DPN-model generated in this way is in many cases too big to be handled directly. For example a single call to some Java API methods can generate more than 100 000 control flow instructions. We therefore use a simple pruning technique on the call graph to reduce the size of the model by automatically removing *uninteresting* procedure calls and overapproximating their possible effects on the control state. This technique depends on the analysis instance and therefore different models have to be generated for different instances. E.g., for a data race analysis methods which do neither directly nor indirectly use any locks or access the field in question are uninteresting and can be pruned. Most analyses like bit vector analyses posses a similar property which allows to prune the model accordingly. Further pruning on the instruction level would also be possible, but was not implemented yet.

3.2.2 DPN Analysis

For analysis we translate the generated DPN-model to a tree automaton which is encoded in a logical program for the XSB system[2]. XSB is a logic programming and deductive database system. This allows for a simple symbolic encoding of the automata manipulating, for instance, sets of locks or graphs. The

[1] `http://wala.sf.net`
[2] `http://xsb.sf.net`

used tree automata operations can be implemented straightforwardly in a logical program utilizing the supported tabled evaluation. To alleviate the exponential blow up in the number of locks, we utilize a special emptiness check for product tree automata, which explores one component top-down and the other bottom-up. If one inspects the defined automata, one recognizes that some of them are extremely nondeterministic when evaluated upwards and others when evaluated downwards, but *nearly deterministic* in the other direction. The emptiness check is implemented by a predicate which marks states for which there exists a tree recognized with them. By ordering the premises of the non-emptiness predicate suitably, the system evaluates the automata appropriately in different directions. As another optimization, we use XSB's answer subsumption based upon a partial order on the states of the automata. That is, if we have two states with the same *conflict potential* and one state has a strictly more restrictive acquisition/release history, then that state can be dropped. By instrumenting the non-emptiness predicate, a witness for the execution violating the checked property can be constructed.

3.3 Application

As an application for the simple reachability analyses we developed a plugin for the Eclipse IDE that runs data race checks on shared fields as described in Example 1. Checking small programs (up to 1k LOC) takes between several milliseconds up to a few seconds (not checking library fields). We investigated several found races manually based on the generated witnesses. Many were actual races, mostly found in exception handlers. False positives were often due to the inability to model the used locks in the DPN and the imprecise field abstraction based on points-to information.

As a second application, the iterated reachability analyses were integrated as a def-use dependency checker into the Joana tool for information flow control[3]. Joana builds a system dependency graph (SDG) that contains so called *interference edges*, which model def-use dependencies between different threads. After generating the system dependency graph with Joana, we check individual interference edges for feasibility. That is, we check whether the writing instruction can be executed before the reading instruction without an intervening *killing* instruction. Recall the examples from Table 1 in Section 1, where, in an information flow control setting, we want to prove that there is no information flow from the assignment of 42 to the `print` statement. Effects of thread creation like in the first example are already handled by Joana, hence it is able to prove the absence of information flow in the first example. In the other examples Joana suspects an information flow which is due to spurious interference edges in the second, third, fifth and sixth example. Our analysis is able to remove these spurious interference edges in these examples, allowing Joana to prove the absence of information flow. In the fourth example both interference edges are feasible on their own, but not both within one execution. We did not yet implement automatic handling of this effect, but were able to handle this with our analysis in a mock-up setting. We intend to integrate handling of those effects with a deeper integration into the slicing algorithm of Joana. This will need to check only *critical* SDG-subpaths for feasibility, as otherwise there are too many paths to be checked.

In order to test performance of the analysis for real world applications, we also tried to eliminate interference edges assumed by Joana in a suite of 24 programs. The suite contained concurrent programs from different benchmark suites and some genuine JavaME programs. The programs contained a total of 32 310 lines of code (excluding library code) ranging from 29 to 6 252 LOC with an average of 1 346. In total, 10 376 dependencies have been checked. Analysis time per dependency varied between 0.16 and 549 seconds with an average of 12 seconds. The number of transitions of the DPN-models ranged from

[3] `http://joana.ipd.kit.edu`

140 to 8 801 with an average of 3 393. We could not find additional infeasible dependencies within these applications. This is likely due to the fact that Joana already excludes many dependencies which are infeasible due to effects of thread creation and only twelve of the programs contained abstractable locks (with a maximum of two). A further reason might be that, due to possibly spurious exceptional control flow, some of the more subtle effects treated by our analysis are not visible in the abstract program model.

4 Conclusion

The aim of this work was to implement DPN-based reachability analyses and explore their applicability for the improvement of an information flow control analysis for Java. For this different existing and new techniques had to be combined on a level appropriate for implementation. Notably, we introduced an iterable, tree-regular characterization of reachable configurations for DPN, based on execution trees. This allows us to conclude that post* preserves tree-regularity. We gave a modular description of the relevant constructions in the form of finite tree automata. This was in particular critical for obtaining an efficient evaluation strategy. The implementation of the non-iterated analysis performs well in most cases and yields encouraging results. The nondeterministic placement of the cut impacts on the performance of the iterated analysis and encourages the development of further optimizations.

Acknowledgments Using DPN analysis in the context of IFC-Analysis for Java was envisioned in a joint DFG project with Gregor Snelting's group from Karlsruhe Institute of Technology (KIT) who develop the Joana framework. We thank our colleagues from KIT for the good collaboration, in particular Jürgen Graf who helped us getting started with the WALA framework and integrating the analysis into the Joana framework and Dennis Giffhorn who assembled the test suite that was used. We thank Alexander Wenner for lots of helpful discussions especially on the decidability issue in the presence of `wait`-calls.

We dedicate this paper to Dave Schmidt, one of the pioneers in studying connections between model checking and data flow analysis, at the occasion of his 60th birthday.

References

[1] Ahmed Bouajjani, Javier Esparza, Alain Finkel, Oded Maler, Peter Rossmanith, Bernard Willems & Pierre Wolper (2000): *An efficient automata approach to some problems on context-free grammars*. *Inf. Process. Lett.* 74(5-6), pp. 221–227, doi:10.1016/S0020-0190(00)00055-7.

[2] Ahmed Bouajjani, Javier Esparza & Oded Maler (1997): *Reachability Analysis of Pushdown Automata: Application to Model-Checking*. In: *CONCUR '97*, *LNCS* 1243, Springer, pp. 135–150, doi:10.1007/3-540-63141-0_10.

[3] Ahmed Bouajjani, Markus Müller-Olm & Tayssir Touili (2005): *Regular Symbolic Analysis of Dynamic Networks of Pushdown Systems*. In: *CONCUR '05*, *LNCS* 3653, Springer, pp. 473–487, doi:10.1007/11539452_36.

[4] J. Richard Büchi (1964): *Regular canonical systems*. *Archiv für mathematische Logik und Grundlagenforschung* 6, pp. 91–111, doi:10.1007/BF01969548.

[5] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison & M. Tommasi (2007): *Tree Automata Techniques and Applications*. Available on: `http://www.grappa.univ-lille3.fr/tata`.

[6] Joost Engelfriet (1975): *Bottom-up and top-down tree transformations— a comparison*. *Theory of Computing Systems* 9, pp. 198–231, doi:10.1007/BF01704020.

[7] Javier Esparza & Jens Knoop (1999): *An Automata-Theoretic Approach to Interprocedural Data-Flow Analysis*. In: *FoSSaCS '99*, *LNCS* 1578, Springer, pp. 14–30, doi:10.1007/3-540-49019-1_2.

[8] Javier Esparza & Andreas Podelski (2000): *Efficient Algorithms for pre* and post* on Interprocedural Parallel Flow Graphs*. In: *POPL '00*, ACM, pp. 1–11, doi:10.1145/325694.325697.

[9] Thomas Gawlitza, Peter Lammich, Markus Müller-Olm, Helmut Seidl & Alexander Wenner (2011): *Join-Lock-Sensitive Forward Reachability Analysis for Concurrent Programs with Dynamic Process Creation*. In: *VMCAI '11*, *LNCS* 6538, Springer, pp. 199–213, doi:10.1007/978-3-642-18275-4_15.

[10] Dennis Giffhorn (2012): *Slicing of Concurrent Programs and its Application to Information Flow Control*. Ph.D. thesis, Karlsruher Institut für Technologie, Fakultät für Informatik.

[11] Christian Hammer & Gregor Snelting (2009): *Flow-sensitive, context-sensitive, and object-sensitive information flow control based on program dependence graphs*. International Journal of Information Security 8(6), pp. 399–422, doi:10.1007/s10207-009-0086-1.

[12] Vineet Kahlon & Aarti Gupta (2006): *An Automata-Theoretic Approach for Model Checking Threads for LTL Properties*. In: *LICS '06*, IEEE, pp. 101–110, doi:10.1109/LICS.2006.11.

[13] Vineet Kahlon, Franjo Ivancic & Aarti Gupta (2005): *Reasoning About Threads Communicating via Locks*. In: *CAV '05*, *LNCS* 3576, Springer, pp. 505–518, doi:10.1007/11513988_49.

[14] Nicholas Kidd, Thomas Reps, Julian Dolby & Mandana Vaziri (2009): *Finding Concurrency-Related Bugs Using Random Isolation*. In: *VMCAI '09*, LNCS 5403, Springer, pp. 198–213, doi:10.1007/978-3-540-93900-9_18.

[15] Peter Lammich, Markus Müller-Olm & Alexander Wenner (2009): *Predecessor Sets of Dynamic Pushdown Networks with Tree-Regular Constraints*. In: *CAV '09*, *LNCS* 5643, Springer, pp. 525–539, doi:10.1007/978-3-642-02658-4_39.

[16] Denis Lugiez & Philippe Schnoebelen (1998): *The Regular Viewpoint on PA-Processes*. In Davide Sangiorgi & Robert de Simone, editors: *CONCUR '98*, *LNCS* 1466, Springer, pp. 50–66, doi:10.1007/BFb0055615.

[17] Denis Lugiez & Philippe Schnoebelen (2002): *The regular viewpoint on PA-processes*. Theoretical Computer Science 274(1–2), pp. 89 – 115, doi:10.1016/S0304-3975(00)00306-6.

[18] Markus Müller-Olm & Helmut Seidl (2001): *On Optimal Slicing of Parallel Programs*. In: *STOC '01*, ACM, pp. 647–656, doi:10.1145/380752.380864.

[19] G. Ramalingam (2000): *Context-sensitive synchronization-sensitive analysis is undecidable*. *ACM Trans. Program. Lang. Syst.* 22(2), pp. 416–430, doi:10.1145/349214.349241.

[20] David A. Schmidt (1998): *Data Flow Analysis is Model Checking of Abstract Interpretations*. In: *POPL '98*, ACM, pp. 38–48, doi:10.1145/268946.268950.

[21] David A. Schmidt & Bernhard Steffen (1998): *Program Analysis as Model Checking of Abstract Interpretations*. In Giorgio Levi, editor: *SAS '98*, *LNCS* 1503, Springer, pp. 351–380, doi:10.1007/3-540-49727-7_22.

[22] Bernhard Steffen (1991): *Data Flow Analysis as Model Checking*. In: Theoretical Aspects of Computer Software, *LNCS* 526, Springer, pp. 346–364, doi:10.1007/3-540-54415-1_54.