Conflict Analysis of Programs with Procedures, Dynamic Thread Creation, and Monitors

Peter Lammich and Markus Müller-Olm

Institut für Informatik, Fachbereich Mathematik und Informatik
Westfälische Wilhelms-Universität Münster
`peter.lammich@uni-muenster.de` and `mmo@math.uni-muenster.de`

Abstract. We study conflict detection for programs with procedures, dynamic thread creation and a fixed finite set of (reentrant) monitors. We show that deciding the existence of a conflict is NP-complete for our model (that abstracts guarded branching by nondeterministic choice) and present a fixpoint-based complete conflict detection algorithm. Our algorithm needs worst-case exponential time in the number of monitors, but is linear in the program size.

1 Introduction

As programming languages with explicit support for parallelism, such as Java, have become popular, the interest in analysis of parallel programs has increased in recent years. A particular problem of parallel programs are conflict situations, where a program is simultaneously in two states that should exclude each other. An example is a data race, where a memory location is simultaneously accessed by two threads, at least one of the accesses being a write access. Particular challenges for analyzing conflicts in a language like Java are dynamic creation of an unlimited number of threads, synchronization via reentrant monitors, and indirect referencing of monitors by their association to objects. It's unrealistic to design an interprocedural analysis that meets all these challenges and is precise. Therefore, this paper concentrates on the first two challenges. More specifically, we develop an analysis that decides the reachability of a conflict situation in (nondeterministic) programs with recursive procedure calls, dynamic thread creation and synchronization via a fixed finite set of reentrant monitors that are statically bound to procedures. We also show this problem to be NP-complete.

Many papers on precise program analysis, e.g. [16, 13, 4], model concurrency via parbegin/parend blocks or parallel procedure calls or assume a fixed set of threads. However, thread-creation cannot be simulated by parbegin/parend for programs with procedures [2]. Precise analysis for programs with thread-creation is treated e.g. in [2, 11]. All the papers mentioned above completely abstract away synchronization. Due to a well-known result of Ramalingam [15], context- and synchronization-sensitive analysis is undecidable. However, Ramalingam considered rendezvous style synchronization which is more powerful than synchronization via monitors studied in this paper. The undecidability border for various properties and synchronization primitives in models with a fixed set of threads is studied more closely in [7]. An interesting approach to address the indirect referencing of monitors extends shape analysis techniques to a concurrent scenario (e.g. [17]). However, this approach does not come with a precision theorem but assesses precision empirically.

The work that comes closest to our goals is by Kahlon et al. [8, 6]. They study analyses for a fixed set of threads communicating via (statically referenced) locks, where each thread is modeled by a pushdown system (which corresponds to programs with recursive procedures). In their model, threads can execute lock/unlock statements for a fixed finite set of locks in a nested fashion, i.e. each thread can only release the lock it acquired last and that was not yet released. Without this nestedness constraint data race detection is undecidable [8, 7]. Moreover, locks are not reentrant in their model, i.e. a thread may not reacquire a lock that it possesses already[1]. In contrast, our model uses reentrant monitors (i.e. „synchronized"-blocks). Monitors correspond to a structured use of nested locks. A monitor can be interpreted as a lock that is acquired upon entering a synchronized block and released when leaving the synchronized block. While synchronized blocks start and end in the same procedure, the lock and unlock statements in [8, 6] can occur anywhere in the program. However, since languages like Java also use synchronized blocks, we believe this restriction being rather harmless.

Our contributions beyond the work of [8] are as follows: We handle thread creation instead of assuming a fixed set of threads; we handle reentrant monitors instead of non-reentrant nested locks; and we use fixpoint based instead of automata based techniques. The running time of our algorithm depends only linearly from the program size, independently of the number of threads actually created. In contrast, the running time of the algorithm of [8] grows at least quadratically with the (statically fixed) number of threads. Moreover, we show conflict analysis to be NP-complete for our model, where NP-hardness also transfers to models with a fixed set of threads including the one of Kahlon et al., which justifies the exponential dependency of the running time of the analysis algorithms from the number of monitors.

This paper is organized as follows: In Section 2 the program model and its semantics is defined. In Section 3, we define an alternative operational semantics that captures only particular schedules, called *restricted executions.* We show that any reachable configuration is also reachable by a restricted execution. In Section 4, we characterize restricted executions by a constraint system. In Section 5, we interpret this constraint system over an abstract domain, obtaining a fixpoint based conflict analysis algorithm needing exponential time in the number of monitors and linear time in the program size. As part of the abstract domain, we use the concept of acquisition histories [8], adapted to reentrant monitors and to a fixpoint based analysis context. Additionally, we show that the conflict analysis problem is NP-complete . Finally, in Section 6, we give a

[1] One can simulate reentrant locks by non-reentrant ones, but at the cost of a worst-case exponential blowup of the program size

conclusion and an outlook to further research. Due to the lack of space, most of the proofs are deferred to an accompanying technical report [9].

2 Program Model

Flowgraphs. We describe programs by nondeterministic interprocedural flowgraphs with monitors. A program $\Pi = (\mathsf{P}, \mathcal{M}, (G_p, \mathsf{m}(p))_{p\in\mathsf{P}})$ consists of a finite set P of procedure names with $\mathsf{main} \in \mathsf{P}$ and a finite set $\mathcal{M}$ of monitor names. Each procedure $p \in \mathsf{P}$, is associated with a flowgraph G_p, describing the body of the procedure, and a set of monitors $\mathsf{m}(p) \subseteq \mathcal{M}$ describing the monitors the procedure synchronizes on, i.e. the monitors in $\mathsf{m}(p)$ are acquired upon entering p and released upon returning from p. A flowgraph $G_p = (\mathsf{N}_p, \mathsf{E}_p, \mathsf{e}_p, \mathsf{r}_p)$ consists of a finite set N_p of control nodes, a set $\mathsf{E}_p \subseteq \mathsf{N}_p \times \mathcal{L}_{\mathsf{Edge}} \times \mathsf{N}_p$ of labeled edges, and distinguished entry and return nodes $\mathsf{e}_p, \mathsf{r}_p \in \mathsf{N}_p$. Edges are labeled with either base, call, or spawn labels: $\mathcal{L}_{\mathsf{Edge}} = \{\mathsf{base}\} \cup \{\mathsf{call}\ p \mid p \in \mathsf{P}\} \cup \{\mathsf{spawn}\ p \mid p \in \mathsf{P}\}$. Intuitively, base edges model basic instructions. We leave there structure unspecified as we will define conflicting situations on the basis of the control state of the program rather than by the executed instructions. Call edges model (potentially recursive) procedure calls and spawn edges model thread creation. As usual, we assume $\mathsf{N}_p \cap \mathsf{N}_{p'} = \emptyset$ for $p \neq p' \in \mathsf{P}$ and define $\mathsf{N} := \bigcup_{p\in\mathsf{P}} \mathsf{N}_p$ and $\mathsf{E} := \bigcup_{p\in\mathsf{P}} \mathsf{E}_p$. We call a procedure $p \in \mathsf{P}$ *initial*, if it is the main procedure ($p = \mathsf{main}$) or a procedure started by a spawn edge ($\exists u, v.(u, \mathsf{spawn}\ p, v) \in \mathsf{E}$). In order to avoid the (unrealistic) possibility that a thread can allocate monitors in the moment it is created, we assume that initial procedures do not synchronize on any monitors. Otherwise, the analysis problem would be more complex (PSPACE-hard). For the rest of this paper, we assume a fixed program $\Pi = (\mathsf{P}, \mathcal{M}, (G_p, \mathsf{m}(p))_{p\in\mathsf{P}})$.

In order to simplify the definition of restricted executions in Section 3, we agree on some further conventions: For each initial procedure p we have $\mathsf{e}_p \neq \mathsf{r}_p$ and there is a procedure q such that $\mathsf{E}_p = \{(\mathsf{e}_p, \mathsf{call}\ q, \mathsf{r}_p)\}$, $\mathsf{e}_q \neq \mathsf{r}_q$, and $\mathsf{E}_q \cap \{(u, l, \mathsf{r}_q) \mid u \in \mathsf{N}_q \wedge l \in \mathcal{L}_{\mathsf{Edge}}\} = \emptyset$, i.e. the return node of q is isolated. These syntactic restrictions guarantee that the execution of any thread starts with a call to a procedure that will never return. However, they do not limit the expressiveness of our model, since we can always rewrite a program to one with the same set of conflicts that satisfies these restrictions, e.g. by introducing starter procedures x_0 and x_1 for every initial procedure x as illustrated in Fig. 1 and replacing $\mathsf{spawn}\ x$ by $\mathsf{spawn}\ x_0$, as well as main by main_0. Note that reaching control node t_x models termination of a thread: Arrived there, it holds no monitors and cannot make any steps.

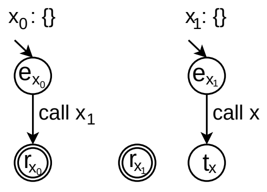

Fig. 1: Starter procedures.

The flowgraph depicted in Fig. 2 is used as a running example throughout this paper. Its main procedure is a_0. For better readability, the x_0 and x_1 procedures (for $x \in \{a, p, r\}$) as well as the base edge labels are not shown. First of all, let us illustrate some effects our analysis has to cope with. Consider nodes 5 and 9. In order to reach 5, we have to call p and then pass through t and in order to reach 9, we have to call q and then pass through s. If one thread calls p first, it is in monitor m_1, and no other thread can pass through s any more. Vice versa, if q is called first, no other thread can pass through t any more. Hence nodes 5 and 9 are not simultaneously reachable, although the sets of monitors held at 5, $\{m_1\}$, and 9, $\{m_2\}$, are disjoint. We will use the concept of *acquisition histories* [8] to handle this effect. Now consider nodes 3 and C. They are simultaneously reachable, because procedure q can create a thread starting with r_0, and after q has returned to node 3, the new thread can pass through procedure t and reach node C. This illustrates that a thread survives the procedure it is created in. In order to cope with this, we need complex procedure summaries. Finally, consider nodes 8 and C. They are not simultaneously reachable, as the thread starting at r_0 cannot pass through procedure t until the initial thread has released monitor m_2. However, if we had no thread creation, but two initial threads of type main and r, the nodes 8 and C would be simultaneously reachable. This illustrates that conflicts in our model with thread creation cannot (easily) be reduced to models with a fixed set of initial threads (as covered by Kahlon et al. [8, 6]).

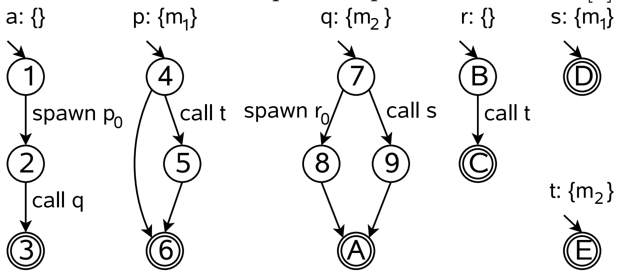

Fig. 2: Example flowgraph with $\mathsf{main} = a_0$.

Operational Semantics. We use the following multiset and list notations: $\mathsf{mset}(R)$ is the set of multisets of elements from R, $\emptyset$ is the empty multiset, $\{x\}$ is the multiset containing x once and $R_1 \uplus R_2$ is the union of the multisets R_1 and R_2. The ambiguity between set and multiset notation is resolved from the context. R^* is the set of lists of elements from R, ε is the empty list, $[e_1, \ldots, e_k]$ is the list of elements $e_1, \ldots, e_k$, and $w_1 w_2$ is the concatenation of the lists w_1 and w_2.

The operational semantics is described as a labeled transition system $\overset{\cdot}{\longrightarrow} \subseteq \mathsf{Conf} \times \mathcal{L}_{\mathsf{LE}} \times \mathsf{Conf}$, where $\mathsf{Conf} := \{\langle s, c\rangle \mid s \in \mathsf{N}^*, c \in \mathsf{mset}(\mathsf{N}^*), \mathsf{cons}(\{s\} \uplus c)\}$ is the set of *monitor consistent* program configurations and $\mathcal{L}_{\mathsf{LE}} := \{l^x | l \in \mathcal{L} \wedge x \in \{\mathsf{L}, \mathsf{E}\}\}$ with $\mathcal{L} := \mathcal{L}_{\mathsf{Edge}} \cup \{\mathsf{ret}\}$ is the set of transition labels. A program configuration $\langle s, c\rangle \in \mathsf{Conf}$ is a pair of a local thread's configuration s and a multiset c of environment threads' configurations. A thread's configuration is modeled as a stack of control nodes, the top element being the current control node and the elements deeper in the stack being stored return addresses. We model stacks as lists with the top of the stack being the first element of the list. For a control node $u \in \mathsf{N}_p$, we define $\mathsf{m}(u) := \mathsf{m}(p)$. For a stack $s \in \mathsf{N}^*$ we define $\mathsf{m}(s) := \bigcup_{n \in s} \mathsf{m}(n)$, for $c \in \mathsf{mset}(\mathsf{N}^*)$, we define $\mathsf{m}(c) = \bigcup_{s \in c} \mathsf{m}(s)$ and for $\langle s, c\rangle \in \mathsf{Conf}$ we define $\mathsf{m}(\langle s, c\rangle) := \mathsf{m}(\{s\} \uplus c)$. A multiset of stacks $c \in \mathsf{mset}(\mathsf{N}^*)$ is *monitor consistent*, if no two threads are inside the same monitor. This is expressed by the predicate $\mathsf{cons}(c) :\Leftrightarrow \nexists s_1, s_2, c_e.\ c = \{s_1\} \uplus \{s_2\} \uplus c_e \wedge \mathsf{m}(s_1) \cap \mathsf{m}(s_2) \neq \emptyset$. Transitions are labeled with the edge that induced the transition or with the ret label for a procedure return. Additionally, we record whether the transition was made on the local thread ($\cdot^{\mathsf{L}}$) or on some environment thread ($\cdot^{\mathsf{E}}$). This distinction is needed when characterizing certain sets of executions by a constraint system in order to distinguish the monitors used by the environment threads from the monitors used by the local thread. We define $\xrightarrow{\cdot}$ to be the least set satisfying the following rules:

$$
\begin{array}{lll}
[\mathsf{base}] & (u, \mathsf{base}, v) \in \mathsf{E} : & \langle [u]r, c\rangle \xrightarrow{\mathsf{base}^{\mathsf{L}}} \langle [v]r, c\rangle \\
[\mathsf{call}] & (u, \mathsf{call}\ q, v) \in \mathsf{E} : & \langle [u]r, c\rangle \xrightarrow{(\mathsf{call}\ q)^{\mathsf{L}}} \langle [\mathsf{e}_q, v]r, c\rangle \qquad \text{if } \mathsf{m}(q) \cap \mathsf{m}(c) = \emptyset \\
[\mathsf{ret}] & q \in \mathsf{P} : & \langle [\mathsf{r}_q]r, c\rangle \xrightarrow{\mathsf{ret}^{\mathsf{L}}} \langle r, c\rangle \\
[\mathsf{spawn}] & (u, \mathsf{spawn}\ q, v) \in \mathsf{E} : & \langle [u]r, c\rangle \xrightarrow{(\mathsf{spawn}\ q)^{\mathsf{L}}} \langle [v]r, \{[\mathsf{e}_q]\} \uplus c\rangle \\
[\mathsf{env}] & \multicolumn{2}{l}{\langle s, \{r\} \uplus c\rangle \xrightarrow{l^{\mathsf{E}}} \langle s, \{r'\} \uplus c'\rangle \text{ if } \langle r, \{s\} \uplus c\rangle \xrightarrow{l^{\mathsf{L}}} \langle r', \{s\} \uplus c'\rangle}
\end{array}
$$

The [base], [call], and [spawn]-rules model the behavior of the corresponding edges. Returning from procedures is modeled by the [ret]-rule. Note that there is no flowgraph edge corresponding to a return step. Finally, the [env]-rule defines the environment steps.

We overload $\xrightarrow{\cdot}$ with its reflexive transitive closure ($\langle s, c\rangle \xrightarrow{w} \langle s', c'\rangle$ with $w \in \mathcal{L}_{\mathsf{LE}}{}^*$) and write $\xrightarrow{*}$ for the execution of an arbitrary path. For $x \in \{\mathsf{L}, \mathsf{E}\}$ and $w = [l_1, \dots, l_n] \in \mathcal{L}^*$ we define $w^x := [l_1^x, \dots, l_n^x]$ and write $c \xrightarrow{w} c'$ as a shorthand notation for $\langle \varepsilon, c\rangle \xrightarrow{w^{\mathsf{E}}} \langle \varepsilon, c'\rangle$. As the empty stack cannot make any steps and holds no monitors, it does not influence the execution. Thus $c \xrightarrow{l} c'$ simply is a transition without explicit local thread. Our semantics preserves monitor consistency of the configurations as the monitor side condition in the [call]-rule ensures that a thread can only enter a procedure if no other thread is inside a monitor the procedure synchronizes on.

The monitors used by a path are the monitors of all procedures that are called on steps of this path: For $l \in \mathcal{L}$, we define $\mathsf{m}(l) := \mathsf{m}(p)$ if $l = \mathsf{call}\ p$ and $\mathsf{m}(l) = \emptyset$ otherwise. We overload this definition for sequences of labels ($\mathsf{m}(w) := \bigcup_{l \in w} \mathsf{m}(l)$ for $w \in \mathcal{L}^*$). For sequences with L/E-labeling $w \in \mathcal{L}_{\mathsf{LE}}{}^*$, we define $\mathsf{m}^{\mathsf{L}}(w)$ to be the set of monitors used by local steps and $\mathsf{m}^{\mathsf{E}}(w)$ to be the set of monitors used by environment steps.

Reachability of a Conflict. For a multiset $C = \{U_1, \dots, U_n\} \in \mathsf{mset}(2^{\mathsf{N}})$ of sets of nodes and a multiset of stacks $c \in \mathsf{mset}(\mathsf{N}^*)$, we define $\mathsf{at}_C(c)$ if and only if $c = c_e \uplus \biguplus_{i=1\dots n}\{[u_i]r_i\}$ for some $c_e \in \mathsf{mset}(\mathsf{N}^*)$, $(r_i \in \mathsf{N}^*)_{i=1\dots n}$, and $(u_i \in U_i)_{i=1\dots n}$, i.e. for each i, c contains an own thread with current control node in U_i. We also define $\mathsf{at}_C(\langle s, c\rangle) :\Leftrightarrow \mathsf{at}_C(\{s\} \uplus c)$. The conflict analysis problem for two sets of control nodes $U, V \subseteq \mathsf{N}$ is to decide the question: *Is there an execution* $\{[\mathsf{e}_{\mathsf{main}}]\} \xrightarrow{*} c$ *with* $\mathsf{at}_{\{U,V\}}(c)$? The reachability problem for a single set U is to decide: *Is there an execution* $\{[\mathsf{e}_{\mathsf{main}}]\} \xrightarrow{*} c$ *with* $\mathsf{at}_{\{U\}}(c)$?

Example 1. In the following example executions, we abbreviate $\mathsf{call}\ p$ by p and $\mathsf{spawn}\ p$ by $+p$. Fig. 3a illustrates an execution of the flowgraph from Fig. 2. The execution starts with a single thread at the entry point of a_0. It calls procedures

a_1, a, and then passes through procedure q. On its way, it spawns two other threads p_0 and r_0. Their steps are interleaved with the initial thread's ones. This execution can be formally described as $\langle \mathsf{e}_{a_0}, \emptyset\rangle \xrightarrow{w} c'$ with transition labels $w = [{a_1}^{\mathsf{L}}, a^{\mathsf{L}}, +{p_0}^{\mathsf{L}}, q^{\mathsf{L}}, {p_1}^{\mathsf{E}}, +{r_0}^{\mathsf{L}}, {r_1}^{\mathsf{E}}, r^{\mathsf{E}}, \mathsf{base}^{\mathsf{L}}, \mathsf{ret}^{\mathsf{L}}, t^{\mathsf{E}}, p^{\mathsf{E}}, \mathsf{ret}^{\mathsf{E}}]$ and end configuration $c' = \langle [3, t_a, \mathsf{r}_{a_0}], \{[C, t_r, \mathsf{r}_{r_0}], [4, t_p, \mathsf{r}_{p_0}]\}\rangle$. Note that this execution witnesses the conflict between nodes 3 and C mentioned above.

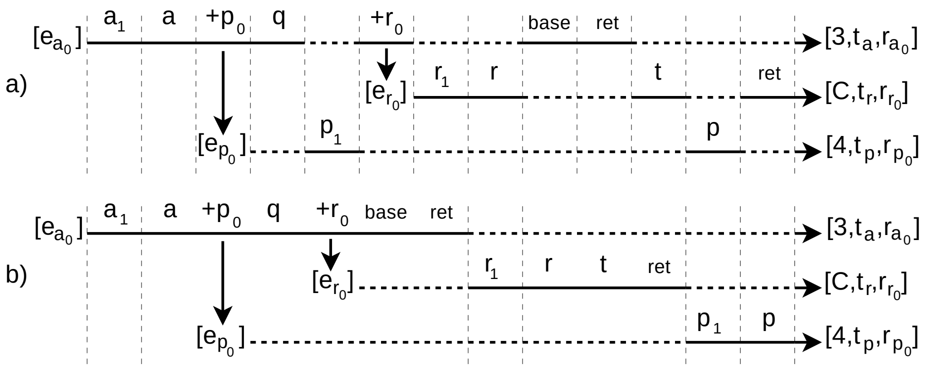

Fig. 3. Sample execution and corresponding restricted execution.

3 Restricted Schedules

In this section we define a restricted operational semantics that only allows a subset of the executions of the original semantics, but preserves the set of reachable configurations, and thus the reachable conflicts. The restricted semantics is better suited for characterization by a constraint system than the original semantics (cf. Section 4).

While in an execution of the original semantics, context switches may occur after each step, the restricted semantics only allows context switches after a thread's last step and before procedure calls that do not return for the rest of the execution. Due to the syntactic convention that assures that the execution of any thread starts with a non-returning call, an atomically scheduled sequence, called a *macrostep*, consists of an initial procedure call, followed by a *same-level path*. A same-level path is a path with balanced calls and returns, i.e. its execution starts and ends at the same stack level, and does not fall below the initial stack level at any point. We define the transition relation of the restricted semantics $\overset{\cdot}{\Longrightarrow} \subseteq \mathsf{Conf} \times \mathsf{MStep} \times \mathsf{Conf}$ with $\mathsf{MStep} := \{([\mathsf{call}\ p]\bar{w})^x \mid p \in \mathsf{P}, \bar{w} \in \mathcal{L}^*, x \in \{\mathsf{L}, \mathsf{E}\}\}$ as the least set satisfying the following rules:

$$[\mathsf{macro}]\ \langle s, c\rangle \overset{([\mathsf{call}\ p]\bar{w})^{\mathsf{L}}}{\Longrightarrow} \langle [v]r', c'\rangle \qquad \text{if } \langle s, c\rangle \xrightarrow{(\mathsf{call}\ p)^{\mathsf{L}}} \langle [\mathsf{e}_p]r', c\rangle \wedge \langle [\mathsf{e}_p], c\rangle \xrightarrow{\bar{w}^{\mathsf{L}}} \langle [v], c'\rangle$$

$$[\mathsf{env}]\quad \langle s, \{r\} \uplus c\rangle \overset{l^{\mathsf{E}}}{\Longrightarrow} \langle s, \{r'\} \uplus c'\rangle \text{ if } \langle r, \{s\} \uplus c\rangle \overset{l^{\mathsf{L}}}{\Longrightarrow} \langle r', \{s\} \uplus c'\rangle$$

The [macro]-rule captures the intuition of a macrostep of the local thread as described above and the [env]-rule infers the environment steps. Note that a same-level execution is written as $\langle [u], c\rangle \xrightarrow{\bar{w}^{\mathsf{L}}} \langle [v], c'\rangle$. As monitors are reentrant, it also implies the executions $\langle [u]r', c\rangle \xrightarrow{\bar{w}^{\mathsf{L}}} \langle [v]r', c'\rangle$ for any stack r', s.t. $\langle [u]r', c\rangle$ is monitor consistent. We extend $\overset{\cdot}{\Longrightarrow}$ to its reflexive transitive closure, write $\overset{*}{\Longrightarrow}$ for the execution of an arbitrary path, and define $c \overset{w}{\Longrightarrow} c' := \langle \varepsilon, c\rangle \overset{w^{\mathsf{E}}}{\Longrightarrow} \langle \varepsilon, c'\rangle$. Note that a transition is then labeled by a sequence of macrosteps $w = [l_1^{x_1}, \ldots, l_n^{x_n}] \in \mathsf{MStep}^*$ with $x_1, \ldots, x_n \in \{\mathsf{L}, \mathsf{E}\}$ where each macrostep $l_i \in \mathsf{MStep}$ $(1 \leq i \leq n)$ is labeled as either a local (l_i^{L}) or an environment (l_i^{E}) step.

As macrosteps always start with a call that does not return for the rest of the execution, the set of allocated monitors does not decrease during an execution:

Theorem 2. *An execution* $\langle s, c\rangle \overset{w}{\Longrightarrow} \langle s', c'\rangle$ *implies* $\mathsf{m}(\langle s, c\rangle) \subseteq \mathsf{m}(\langle s', c'\rangle)$.

Starting with an initial procedure, the sets of configurations reachable by executions of the original semantics and of the restricted semantics are the same:

Theorem 3. *Let* $p \in \mathsf{P}$ *be an initial procedure. A configuration can be reached from* p *by an execution of the original semantics if and only if it can be reached by an execution of the restricted semantics. Formally:* $\{[\mathsf{e}_p]\} \xrightarrow{*} c' \Leftrightarrow \{[\mathsf{e}_p]\} \overset{*}{\Longrightarrow} c'$.

Example 4. Fig. 3b illustrates a restricted execution that reaches the same configuration c' as the execution from Fig. 3a. It is described as $\langle \mathsf{e}_{a_0}, \emptyset\rangle \overset{w'}{\Longrightarrow} c'$ with $w' = [[a_1]^{\mathsf{L}}, [a, +p_0, q, +r_0, \mathsf{base}, \mathsf{ret}]^{\mathsf{L}}, [r_1]^{\mathsf{E}}, [r, t, \mathsf{ret}]^{\mathsf{E}}, [p_1]^{\mathsf{E}}, [p]^{\mathsf{E}}]$.

Monitor Consistent Interleaving. For a single macrostep $l = [\mathsf{call}\ p]\bar{w}$, we define $\mathsf{ent}(l) := \mathsf{m}(p)$ and $\mathsf{pass}(l) = \mathsf{m}(\bar{w})$, i.e. the monitors that are entered and never exited by a macrostep and the monitors that are passed (entered and exited again), respectively. We inductively define the monitor consistent interleaving operator $\otimes : \mathsf{MStep}^* \times \mathsf{MStep}^* \to \mathsf{MStep}^*$ by $\varepsilon \otimes w = w \otimes \varepsilon = \{w\}$ and $[l_1]w_1 \otimes [l_2]w_2 := \bigcup_{i=1,2}\{[l_i]w \mid w \in w_i \otimes [l_{3-i}]w_{3-i} \wedge \mathsf{ent}(l_i) \cap \mathsf{m}([l_{3-i}w_{3-i}]) = \emptyset\}$. Note that the $\otimes$-operator is not aware of the L/E-labeling of its operands, it just copies the labeling to the result. Monitor consistent interleaving is a restriction of the usual interleaving to those interleavings where no monitor is used by one path if it has been entered by the other path. For example, in the flowgraph of Fig. 2, we have $[[r, t, \mathsf{ret}]^{\mathsf{E}}] \otimes [[q]^{\mathsf{L}}] = \{[[r, t, \mathsf{ret}]^{\mathsf{E}}, [q]^{\mathsf{L}}]\}$. Note that the macrostep sequence $[[q]^{\mathsf{L}}, [r, t, \mathsf{ret}]^{\mathsf{E}}]$ is not a monitor consistent interleaving, as q enters monitor m_2 that inhibits execution of the macrostep $[r, t, \mathsf{ret}]^{\mathsf{E}}$. We show that the $\otimes$-operator captures the behavior of our interleaving semantics:

Theorem 5. *For configurations* $\langle s, c_1 \uplus c_2\rangle$, $\langle s', c'\rangle \in \mathsf{Conf}$ *and a macrostep path* $w \in \mathsf{MStep}^*$, *we have* $\langle s, c_1 \uplus c_2\rangle \overset{w}{\Longrightarrow} \langle s', c'\rangle$ *if and only if there exist* $w_1, w_2 \in \mathsf{MStep}^*$ *with* $w \in w_1 \otimes w_2^{\mathsf{E}}$ *and* $c_1', c_2' \in \mathsf{mset}(\mathsf{N}^*)$ *with* $c' = c_1' \uplus c_2'$, $\langle s, c_1\rangle \overset{w_1}{\Longrightarrow} \langle s', c_1'\rangle$, $c_2 \overset{w_2}{\Longrightarrow} c_2'$, $\mathsf{m}(\langle s, c_1\rangle) \cap \mathsf{m}(c_2) = \emptyset$, $\mathsf{m}(\langle s, c_1\rangle) \cap \mathsf{m}(w_2) = \emptyset$, *and* $\mathsf{m}(c_2) \cap \mathsf{m}(w_1) = \emptyset$.

Intuitively, this theorem states that we can split an execution by the threads in its starting configuration into interleavable executions, and, vice versa, can combine interleavable executions into one execution. Note that the interleaving operator $\otimes$ only ensures that monitors allocated by one execution do not interfere with the monitors used by the other execution, but is not aware of the monitors of the start configurations. Hence, the last three conditions in this theorem ensure that the resulting combined configuration is monitor consistent and that the monitors of the start configuration of one execution do not interfere with the monitors used by the other execution.

Example 6. Consider the executions $\langle[7],\emptyset\rangle \xRightarrow{[s]^{\mathsf{L}}} \langle[D,9],\emptyset\rangle$ and $\{[4]\} \xRightarrow{[t]^{\mathsf{E}}} \{[E,5]\}$ of the flowgraph in Fig. 2. Although $w := [[s]^{\mathsf{L}},[t]^{\mathsf{E}}] \in [[s]^{\mathsf{L}}] \otimes [[t]^{\mathsf{E}}]$, there is no execution $\langle[7],[4]\rangle \xRightarrow{w} \langle[D,9],\{[E,5]\}\rangle$, as $\mathsf{m}(\{4\}) \cap \mathsf{m}([[s]^{\mathsf{L}}]) = \{m_1\} \neq \emptyset$.

4 Constraint Systems

In this section we develop a constraint system based characterization of restricted executions starting at a single control node. For a procedure $p \in \mathsf{P}$, we want to represent all executions starting with a call of p and reaching some configuration $\langle s,c\rangle$. However, we omit the initial procedure call in order to make the execution independent from the monitors held at the call site. The representation of such an execution, called a *reaching triple*, is a triple of the first macrostep's same-level path, the remaining macrosteps, and the reached configuration:

$$R^{\mathsf{op}}[p] := \{(\bar{w}, w, \langle s,c\rangle) \mid \exists \tilde{u},\tilde{c}.\ \langle[\mathsf{e}_p],\emptyset\rangle \xrightarrow{\bar{w}^{\mathsf{L}}} \langle[\tilde{u}],\tilde{c}\rangle \xrightarrow{w} \langle s,c\rangle\}$$

The procedure summary information $S^{\mathsf{op}}[u]$ is collected for each control node u in a forwards manner; thus the actual summary for procedure p is $S^{\mathsf{op}}[\mathsf{r}_p]$. It contains triples of a same-level path $\bar{w}$ from the procedure's entry node to u, a macrostep path w of the threads spawned during the execution of the same-level path and the configuration c reached by those threads: $S^{\mathsf{op}}[u] := \{(\bar{w}, w, \langle\varepsilon,c\rangle) \mid \exists\tilde{c}.\ \langle[\mathsf{e}_p],\emptyset\rangle \xrightarrow{\bar{w}^{\mathsf{L}}} \langle[u],\tilde{c}\rangle \wedge \langle\varepsilon,\tilde{c}\rangle \xrightarrow{w} \langle\varepsilon,c\rangle\}$. Note that an artificial ε-component is included in the entries of $S^{\mathsf{op}}[u]$ in order to have entries of the same form in S^{op} and R^{op}. Thus we have $R^{\mathsf{op}}[p], S^{\mathsf{op}}[u] \subseteq \mathsf{D}$ for $\mathsf{D} := \mathcal{L}^* \times \mathsf{MStep}^* \times \mathsf{Conf}$. This allows us to handle these sets more uniformly. The last two elements of a procedure-summary triple collect information about steps that may be executed after the procedure has returned. This accounts for the fact that spawned threads survive the procedure they where created in. Note how restricted schedules reduce the complexity here: If we would collect arbitrarily scheduled executions, we would have to interleave a prefix of the steps of the created threads with the same-level path and record the suffix in the second component. This would complicate the constraint system and also the monitor consistent interleaving operator.

Both, the reaching and the same-level information have a closure property, that results from the fact that the empty path is always executable. Formally, $R^{\mathsf{op}}[p], S^{\mathsf{op}}[u] \in \mathsf{L}_R \subseteq 2^{\mathsf{D}}$ with $\mathsf{L}_R := \{X \mid \forall(\bar{w}, w, \langle s,c\rangle) \in X.\ \exists\tilde{s},\tilde{c}.\ (\bar{w},\varepsilon,\langle\tilde{s},\tilde{c}\rangle) \in X\}$. This closure property is important for the abstraction done later, as it allows us to ignore steps that are not necessary to reach the conflict. Moreover, we have $S^{\mathsf{op}}[u] \in \mathsf{L}_S \subseteq \mathsf{L}_R$ for $\mathsf{L}_S := \{X \in \mathsf{L}_R \mid \forall(\bar{w}, w, \langle s,c\rangle) \in X.\ s = \varepsilon\}$. Both L_R and L_S ordered by set inclusion are complete sub-lattices of $(2^{\mathsf{D}}, \subseteq)$.

Example 7. The executions $\langle[7],\emptyset\rangle \xrightarrow{\bar{w}_1^{\mathsf{L}}} \langle[9],\emptyset\rangle$, $\langle[1],\emptyset\rangle \xrightarrow{{+p_0}^{\mathsf{L}}} \langle[2],\{[\mathsf{e}_{p_0}]\}\rangle$, and $\{[\mathsf{e}_{p_0}]\} \xRightarrow{w_2} \{[5,t_p,\mathsf{r}_{p_0}]\}$ with $\bar{w}_1 := [s,\mathsf{ret}]$ and $w_2 := [[p_1]^{\mathsf{E}},[p,t,\mathsf{ret}]^{\mathsf{E}}]$ of the flowgraph from Fig. 2 give rise to the reaching triples $\mathsf{Tr}_1 := (\bar{w}_1,\varepsilon,\langle[9],\emptyset\rangle) \in R^{\mathsf{op}}[q]$ and $\mathsf{Tr}_2 := ([+p_0],w_2,\langle\varepsilon,\{[5,t_p,\mathsf{r}_{p_0}]\}\rangle) \in S^{\mathsf{op}}[2]$. Moreover, as $\{[\mathsf{e}_{p_0}]\} \xRightarrow{\varepsilon} \{[\mathsf{e}_{p_0}]\}$ is also a valid execution, we have $([+p_0],\varepsilon,\langle\varepsilon,\{[\mathsf{e}_{p_0}]\}\rangle) \in S^{\mathsf{op}}[2]$, witnessing the closure property.

We characterize R^{op} and S^{op} as the least solution of the following system of inequations (constraint system), where the variables $(R[p])_{p\in\mathsf{P}}$ range over L_R and $(S[u])_{u\in\mathsf{N}}$ range over L_S.

$$\begin{array}{lll}
[\mathsf{REMPTY}]\ u \in \mathsf{N}_p : & R[p] & \supseteq S[u]|_{\neg\mathsf{m}(p)} * \{(\varepsilon,\varepsilon,\langle[u],\emptyset\rangle)\} \\
[\mathsf{RCALL}]\ \ (u,\mathsf{call}\ q,v) \in \mathsf{E}_p : & R[p] & \supseteq S[u]|_{\neg\mathsf{m}(p)} * ((u,\mathsf{call}\ q,v);R[q]) \\
[\mathsf{SEMPTY}]\ p \in \mathsf{P} : & S[\mathsf{e}_p] & \supseteq \{\varepsilon,\varepsilon,\langle\varepsilon,\emptyset\rangle\} \\
[\mathsf{SBASE}]\ \ (u,\mathsf{base},v) \in \mathsf{E}_p : & S[v] & \supseteq S[u] * \mathsf{base} \\
[\mathsf{SCALL}]\ \ (u,\mathsf{call}\ q,v) \in \mathsf{E}_p : & S[v] & \supseteq S[u] * \mathsf{call}\ q * S[\mathsf{r}_q] * \mathsf{ret} \\
[\mathsf{SSPAWN}]\ (u,\mathsf{spawn}\ q,v) \in \mathsf{E}_p : & S[v] & \supseteq S[u] * \mathsf{spawn}\ q * \mathsf{env}(R[q])
\end{array}$$

Here l is an abbreviation of $\{([l],\varepsilon,\langle\varepsilon,\emptyset\rangle)\}$ for $l \in \{\mathsf{base},\mathsf{call}\ q,\mathsf{ret},\mathsf{spawn}\ q\}$. The operator $\cdot|_{\neg M} : \mathsf{L}_S \to \mathsf{L}_S$ is defined by $X|_{\neg M} := X \cap \{(\bar{w},w,\langle s,c\rangle) \mid \mathsf{m}^{\mathsf{E}}(w) \cap M = \emptyset\}$. The operators $(u,\mathsf{call}\ q,v);\cdot : \mathsf{L}_R \to \mathsf{L}_R$, $\mathsf{env}(\cdot) : \mathsf{L}_R \to \mathsf{L}_S$, and $* : \mathsf{L}_S \times \mathsf{L}_R \to \mathsf{L}_R$ are the natural extensions to sets of the following definitions:

$$\begin{array}{ll}
(u,\mathsf{call}\ q,v);(\bar{w},w,\langle s,c\rangle) & := \{(\varepsilon,\varepsilon,\langle[u],\emptyset\rangle)\} \cup \\
 & \quad \{(\varepsilon,[([\mathsf{call}\ q]\bar{w})^{\mathsf{L}}]w,\langle s[v],c\rangle)\}|_{\neg\mathsf{m}(v)} \\
\mathsf{env}(\bar{w},w,\langle s,c\rangle) & := (\varepsilon,w^{!\mathsf{E}},\langle\varepsilon,\{s\} \uplus c\rangle) \\
(\bar{w}_1,w_1,\langle\varepsilon,c_1\rangle) * (\bar{w}_2,w_2,\langle s_2,c_2\rangle) & := \{(\bar{w}_1\bar{w}_2,w,\langle s_2,c_1 \uplus c_2\rangle) \mid w \in w_1 \otimes w_2\}
\end{array}$$

Here, the expression $w^{!\mathsf{E}}$ is defined as the relabeling of all steps in w to environment steps. It is straightforward to see that the operators are well-defined w.r.t. their specified signatures and that they are monotonic. Moreover, it is easy to see that $X * Y \in \mathsf{L}_S$ for $X,Y \in \mathsf{L}_S$. Therefore, we can use $*$ also as an operator of type $\mathsf{L}_S \times \mathsf{L}_S \to \mathsf{L}_S$ as done in the constraints for S.

Now we explain the constraints and operators: The main work is done by the $*$-operator which is generalized concatenation. It concatenates the same-level components of its operands, interleaves the macrostep components, and joins the reached configurations. A reaching triple in $R[p]$ is constructed by regarding a same-level path to some node $u \in \mathsf{N}_p$, represented by a summary triple from $S[u]$.[2] We distinguish the cases whether the local thread stays at node u ([REMPTY]) or whether it performs further macrosteps ([RCALL]). In the former case, we append the triple $(\varepsilon,\varepsilon,\{[u],\emptyset\})$. This sets the stack reached by the local thread to $[u]$. As we are not going to return from procedure p any more, we have to filter out triples that use monitors of procedure p in steps from spawned threads (environment steps). This is done by the $\cdot|_{\neg\mathsf{m}(p)}$-operation. Note

[2] Re-using the procedure summary information to describe initial segments of paths is a common technique to save redundant constraints.

that these monitors may be used in local steps, as prepending the call renders the subsequent uses to be reentering. In the latter case, we find a call edge of the form $(u, \mathsf{call}\ q, v)$, as macrosteps always start with a procedure call. The $(u, \mathsf{call}\ q, v); R[q]$-operation constructs a macrostep path from a reaching triple in $R[q]$, by adding the call edge and filtering out triples whose monitors conflict with the monitors $\mathsf{m}(v)$ held at the call site. Note that we also include a triple for the empty path in the result of the $(u, \mathsf{call}\ q, v); \cdot$-operator, in order to make it preserve the closure property. The [SEMPTY]-constraint accounts for the empty path from the beginning of a procedure and the [SBASE]- and [SCALL]-constraints propagate information over base and call edges respectively. The [SSPAWN]-constraint describes the effect of a spawn edge. The steps of the spawned thread are constructed from the $R[q]$-information. From the point of view of the thread executing the spawn edge, they are environment steps. The env-operation does the necessary L/E-relabeling. Note that the same-level components of triples in $R[q]$ are always empty, because due to our conventions a spawned procedure begins with a non-returning call. Hence, the env-operator ignores the same-level path component of its operand. By the well-known Knaster-Tarski fixpoint theorem the above constraint system has a least solution. In the following, R and S refer to the components of the least solution.

Theorem 8 (Correctness). *The least solution (R, S) is equal to the operational characterization, i.e. $R[p] = R^{\mathsf{op}}[p]$ and $S[u] = S^{\mathsf{op}}[u]$ for $p \in \mathsf{P}, u \in \mathsf{N}$.*

5 Abstractions

In this section, we develop an abstract interpretation of the constraint system over a finite domain that allows us to do conflict analysis by effective fixpoint computation. First, we briefly recall the concept of acquisition histories and describe our abstract domain and the abstract operators. We then analyze the running time of the resulting algorithm and show that the conflict detection problem is NP-complete.

Acquisition Histories. The concept of acquisition histories was introduced by Kahlon, Ivancic, and Gupta [8, 6] to decide the interleavability of executions allocating locks in a well-nested fashion, but can also be applied to our reentrant monitors. The idea of acquisition histories is that two executions w_1 and w_2 are interleavable if and only if there is no *conflicting pair* of monitors m_1, m_2, that is w_1 enters m_1 and then uses m_2 and, vice versa, w_2 enters m_2 and then uses m_1. We define the set of acquisition histories by $\mathcal{H} := \{h : \mathcal{M} \to 2^{\mathcal{M}} \mid \forall m.\ h(m) = \emptyset \vee m \in h(m)\}$. Intuitively, an acquisition history maps all monitors m that are entered during an execution to the set of all monitors that are used after or in the same step as entering m. Hence we can define interleavability of two acquisition histories as $h_1 * h_2 :\Leftrightarrow \nexists m_1, m_2.\ m_2 \in h_1(m_1) \wedge m_1 \in h_2(m_2)$. In order to construct the acquisition history of a path backwards, we define the operator $\cdot;\cdot\ :\ 2^{\mathcal{M}} \times 2^{\mathcal{M}} \times \mathcal{H} \to \mathcal{H}$, that prepends a macrostep to an acquisition history: $((M_e, M_p); h)(m) :=$ if $m \in M_e$ then $M_e \cup M_p$ else $h(m)$.

Intuitively, M_e is the set of monitors entered by the prepended macrostep and M_p is the set of monitors used in the whole path, including the prepended step. We define the acquisition history of a macrostep path by: $\alpha^{\mathsf{ah}}(\varepsilon) := \lambda m.\emptyset$ and $\alpha^{\mathsf{ah}}([l]w) := (\mathsf{ent}(l), \mathsf{pass}(l) \cup \mathsf{m}(w)); \alpha^{\mathsf{ah}}(w)$. We define a pointwise subset ordering on acquisition histories by $h \preceq h' :\Leftrightarrow \forall m.\ h(m) \subseteq h'(m)$. Obviously, this is an ordering and we have $h \preceq h' \wedge h' * h_2 \Rightarrow h * h_2$, i.e. a smaller acquisition history is interleavable with everything a bigger one is. The following theorem states that acquisition histories can be used to decide whether two paths are interleavable. It can be proven along the lines of [8].

Theorem 9. *For macrostep paths $w_1, w_2 \in \mathsf{MStep}^*$ we have $w_1 \otimes w_2 \neq \emptyset$ if and only if $\alpha^{\mathsf{ah}}(w_1) * \alpha^{\mathsf{ah}}(w_2)$.*

Abstract Domain. Let $U, V \subseteq \mathsf{N}$ be the two sets defining the conflict of interest. For a reaching triple, we record up to four abstract values from the set $\mathsf{D}^\sharp := \mathsf{D}_0^\sharp \cup \mathsf{D}_1^\sharp \cup \mathsf{D}_2^\sharp$, where $\mathsf{D}_0^\sharp := 2^{\mathcal{M}}$, $\mathsf{D}_1^\sharp := \mathcal{C}_1 \times (2^{\mathcal{M}})^3 \times \mathcal{H}$ with $\mathcal{C}_1 := \{\{U\}, \{V\}\}$, and $\mathsf{D}_2^\sharp := (2^{\mathcal{M}})^3$. While entries from $\mathsf{D}_0^\sharp$ are recorded for every reaching triple, entries from $\mathsf{D}_1^\sharp$ are recorded only for triples reaching one of the given sets U or V as specified by the first component, and entries from $\mathsf{D}_2^\sharp$ are recorded only for triples reaching a conflict. More specifically, the recorded information is specified by the abstraction functions $(\alpha_i :\ \mathsf{D} \to 2^{\mathsf{D}_i^\sharp})_{i=0,1,2}$:

$$\begin{aligned}
\alpha_0(\bar{w}, w, \langle s, c\rangle) &:= \{\mathsf{m}(\bar{w})\}\\
\alpha_1(\bar{w}, w, \langle s, c\rangle) &:= \{(C, \mathsf{m}(\bar{w}), \mathsf{m}^{\mathsf{L}}(w), \mathsf{m}^{\mathsf{E}}(w), \alpha^{\mathsf{ah}}(w)) \mid C \in \mathcal{C}_1, \mathsf{at}_C(\langle s, c\rangle)\}\\
\alpha_2(\bar{w}, w, \langle s, c\rangle) &:= \{(\mathsf{m}(\bar{w}), \mathsf{m}^{\mathsf{L}}(w), \mathsf{m}^{\mathsf{E}}(w)) \mid \mathsf{at}_{\{U,V\}}(\langle s, c\rangle)\}
\end{aligned}$$

For $X \subseteq \mathsf{D}$, we define $\alpha_i(X) := \bigcup\{\alpha_i(x) \mid x \in X\}$. In order to treat entries from $\mathsf{D}_0^\sharp$,$\mathsf{D}_1^\sharp$,$\mathsf{D}_2^\sharp$ uniformly, we sometimes identify entries $M \in \mathsf{D}_0^\sharp$ with the tuple $(\emptyset, M, \emptyset, \emptyset, -)$ and entries $(\bar{M}, M_{\mathsf{L}}, M_{\mathsf{E}}) \in \mathsf{D}_2^\sharp$ with $(\{U, V\}, \bar{M}, M_{\mathsf{L}}, M_{\mathsf{E}}, -)$ and define $\mathcal{C} := \mathcal{C}_1 \cup \{\emptyset, \{U, V\}\}$. The symbol $-$ is a substitute for an acquisition history and we define $(M_e, M_p); - := -$ and agree that $- \preceq -$. Note that we never compare acquisition histories with $-$.

On $\mathsf{D}^\sharp$ we define an ordering $\leq$ by $(C, \bar{M}, M_{\mathsf{L}}, M_{\mathsf{E}}, h) \leq (C', \bar{M}', M_{\mathsf{L}}', M_{\mathsf{E}}', h')$ if and only if $C = C'$, $\bar{M} \subseteq \bar{M}'$, $M_{\mathsf{L}} \subseteq M_{\mathsf{L}}'$, $M_{\mathsf{E}} \subseteq M_{\mathsf{E}}'$, and $h \preceq h'$. Intuitively, $d < d'$ means that d reaches the same set $C \in \mathcal{C}$ of interesting nodes as d', but with weaker monitor requirements. Thus d' can be substituted by d in any context, and it is sufficient to collect the minimal elements when abstracting a set of reaching triples. Therefore we work with antichains. Formally, for an ordered set $(X, \leq)$ we write $(\mathsf{ac}(X), \sqsubseteq)$ for the complete lattice of antichains of X, i.e. $\mathsf{ac}(X) := \{M \subseteq X \mid \forall m, m' \in M.\ \neg m < m'\}$ and $M \sqsubseteq M' :\Leftrightarrow \forall m \in M. \exists m' \in M'.\ m \leq m'$. For an arbitrary set $M \subseteq X$, we write M^{ac} for the antichain reduction of M, i.e. the set of minimal elements of M. Note that $\cdot^{\mathsf{ac}}$ distributes over union and for $\mathcal{X} \subseteq \mathsf{ac}(X)$, the supremum of $\mathcal{X}$ is $\bigsqcup \mathcal{X} = (\bigcup \mathcal{X})^{\mathsf{ac}}$. Now we define our abstract domain by $\mathsf{L}^\sharp := \mathsf{ac}(\mathsf{D}^\sharp)$ and our abstraction $\alpha : \mathsf{L}_R \to \mathsf{L}^\sharp$ by $\alpha(X) = (\alpha_0(X) \cup \alpha_1(X) \cup \alpha_2(X))^{\mathsf{ac}}$. The abstraction α distributes over union, i.e. $\alpha(\bigcup \mathcal{X}) = \bigsqcup\{\alpha(X) \mid X \in \mathcal{X}\}$ for all $\mathcal{X} \subseteq \mathsf{L}_R$. Hence it is the lower adjoint of a Galois connection [12].

Example 10. Consider the reaching triples Tr_1 and Tr_2 introduced in Example 7. For $U := \{5\}$ and $V := \{9\}$, we have $\alpha(\{\mathsf{Tr}_1\}) = \{\{m_1\}\} \cup \{\mathsf{Tr}_1^\sharp\}$ with $\alpha_1(\mathsf{Tr}_1) = \{(\{V\}, \{m_1\}, \emptyset, \emptyset, \lambda m.\emptyset)\} =: \{\mathsf{Tr}_1^\sharp\}$ and $\alpha(\{\mathsf{Tr}_2\}) = \{\emptyset\} \cup \{\mathsf{Tr}_2^\sharp\}$ with $\alpha_1(\mathsf{Tr}_2) = \{(\{U\}, \emptyset, \emptyset, \{m_1, m_2\}, (m_1 \mapsto \{m_1, m_2\}))\} =: \{\mathsf{Tr}_2^\sharp\}$.

The $X * Y$-operation combines two reaching triples $t_1 := (\bar{w}_1, w_1, \langle \varepsilon, c_1 \rangle) \in X$ and $t_2 := (\bar{w}_2, w_2, \langle s, c_2 \rangle) \in Y$. The reached configuration of the resulting triples is $\langle s, c_1 \uplus c_2 \rangle$. For $C \in \mathcal{C}$, there are four cases for $\mathsf{at}_C(\langle s, c_1 \uplus c_2 \rangle)$. Either $\mathsf{at}_C(c_1)$, or $\mathsf{at}_C(c_2)$, or $C = \{U, V\}$ and $\mathsf{at}_{\{U\}}(c_1) \wedge \mathsf{at}_{\{V\}}(\langle s, c_2 \rangle)$ or $\mathsf{at}_{\{V\}}(c_1) \wedge \mathsf{at}_{\{U\}}(\langle s, c_2 \rangle)$. In the first case the interesting nodes are all reached by t_1. We then consider the triple $\tilde{t} := (\bar{w}_2, \varepsilon, \langle \tilde{s}, \tilde{c} \rangle) \in Y$ that exists due to the closure property of L_R. We have $\alpha(t_1 * \tilde{t}) = \alpha(\bar{w}_1\bar{w}_2, w_1, \langle \tilde{s}, c_1 \uplus \tilde{c} \rangle) \sqsubseteq \alpha(t_1 * t_2)$. Thus for the abstraction of the result, we have to only consider $\alpha(t_1 * \tilde{t})$, which has the same acquisition history as t_1. Analogously, in the second case we only need to consider interleavings of the form $\tilde{t} * t_2$ for some $\tilde{t} = (\bar{w}_1, \varepsilon, \langle \varepsilon, \tilde{c} \rangle) \in X$. In the last two cases, the abstractions of both t_1 and t_2 contain the acquisition histories of w_1 and w_2, respectively. These can be used to check whether an interleaving exists. The abstraction of the resulting triples is then in $\mathsf{D}_2^\sharp$ (i.e. reaching a conflict) and thus contains no acquisition history. The operator $*^\sharp : \mathsf{L}^\sharp \times \mathsf{L}^\sharp \to \mathsf{L}^\sharp$ captures the ideas described above and is defined as the natural extension to antichains of the following definition: $(C_1, \bar{M}_1, M_{\mathsf{L}1}, M_{\mathsf{E}1}, h_1) *^\sharp (C_2, \bar{M}_2, M_{\mathsf{L}2}, M_{\mathsf{E}2}, h_2) := \{(C_i, \bar{M}_1 \cup \bar{M}_2, M_{\mathsf{L}i}, M_{\mathsf{E}i}, h_i) \mid i = 1,2\}^{\mathsf{ac}} \sqcup \{(\{U, V\}, \bar{M}_1 \cup \bar{M}_2, M_{\mathsf{L}1} \cup M_{\mathsf{L}2}, M_{\mathsf{E}1} \cup M_{\mathsf{E}2}, -) \mid C_i = \{U\} \wedge C_{3-i} = \{V\} \wedge h_1 * h_2 \wedge i = 1,2\}^{\mathsf{ac}}$. The definitions of the other abstract operators are straightforward (extended to antichains where necessary):

$$\begin{aligned}
&X|^\sharp_{\neg M} && := X \cap \{(\cdot,\cdot,\cdot,M_{\mathsf{E}},\cdot) \in \mathsf{D}^\sharp \mid M_{\mathsf{E}} \cap M = \emptyset\}\\
&\mathsf{env}^\sharp((C, \bar{M}, M_{\mathsf{L}}, M_{\mathsf{E}}, h)) && := \{(C, \emptyset, \emptyset, M_{\mathsf{L}} \cup M_{\mathsf{E}}, h)\}\\
&(u, \mathsf{call}\ q, v);^\sharp (C, \bar{M}, M_{\mathsf{L}}, M_{\mathsf{E}}, h) && := \alpha(\varepsilon, \varepsilon, \langle [u], \emptyset \rangle) \sqcup\\
&\quad \{(C, \emptyset, \mathsf{m}(q) \cup \bar{M} \cup M_{\mathsf{L}}, M_{\mathsf{E}}, (\mathsf{m}(q), \bar{M} \cup M_{\mathsf{L}} \cup M_{\mathsf{E}}); h) \mid C \neq \emptyset\}|^\sharp_{\neg \mathsf{m}(v)}
\end{aligned}$$

Example 11. We show how our analysis utilizes acquisition histories to prevent detection of a spurious conflict between nodes 5 and 9 in the flowgraph of Fig. 2. So let us assume $U := \{5\}$ and $V := \{9\}$. The combination of the paths to U and V is done by the [RCALL]-constraint for the edge $(2, \mathsf{call}\ q, 3)$. We consider the reaching triples $\mathsf{Tr}_1^\sharp \in R^\sharp[q]$ and $\mathsf{Tr}_2^\sharp \in S^\sharp[2]$ from Example 10. We have $\mathsf{Tr}_2^\sharp = (\{U\}, \emptyset, \emptyset, \{m_1, m_2\}, (m_1 \mapsto \{m_1, m_2\})) \in S^\sharp[2]|^\sharp_{\neg \mathsf{m}(2)}$ as $\mathsf{m}(2) = \emptyset$ and from $\mathsf{Tr}_1^\sharp$ we get $(\{V\}, \emptyset, \{m_1, m_2\}, \emptyset, (m_2 \mapsto \{m_1, m_2\})) \in (2, \mathsf{call}\ q, 3);^\sharp R^\sharp[q]$. However, the acquisition histories $h_1 := (m_1 \mapsto \{m_1, m_2\})$ and $h_2 := (m_2 \mapsto \{m_1, m_2\})$ are not interleavable $(\neg h_1 * h_2)$ because of the conflicting pair of monitors m_1, m_2 (i.e. $m_2 \in h_1(m_1)$ and $m_1 \in h_2(m_2)$). Therefore, these two entries are not combined by the $*^\sharp$-operator.

Lemma 12. *The abstract operators mirror the corresponding concrete operators precisely, i.e. for $X_R \in \mathsf{L}_R$ and $X_S \in \mathsf{L}_S$ the following holds:*

$$\begin{aligned}
&\alpha(X_S|_{\neg M}) = \alpha(X_S)|^\sharp_{\neg M} && \alpha((u, \mathsf{call}\ q, v); X_R) = (u, \mathsf{call}\ q, v);^\sharp \alpha(X_R)\\
&\alpha(\mathsf{env}(X_R)) = \mathsf{env}^\sharp(\alpha(X_R)) && \alpha(X_S * X_R) = \alpha(X_S) *^\sharp \alpha(X_R)
\end{aligned}$$

The proof for the $*$-operator follows the ideas described above and is omitted here due to the limited space. The proofs for the other operators are straightforward.

Theorem 13. *Let* $(R^\sharp, S^\sharp)$ *be the least solution of the constraint system interpreted over the abstract domain* $\mathsf{L}^\sharp$ *using the abstract operators and replacing the constants by their abstractions. It exactly matches the least solution* (R, S) *of the concrete constraint system, i.e.* $\forall p \in \mathsf{P}.\ \alpha(R[p]) = R^\sharp[p]$ *and* $\forall u \in \mathsf{N}.\ \alpha(S[u]) = S^\sharp[u]$*. It can be computed in time* $O((|\mathsf{N}| + |\mathsf{E}|) \cdot 2^{\mathsf{poly}(|\mathcal{M}|)})$.

Theorem 13 follows from Lemma 12 by standard results of abstract interpretation, see, e.g. [3, 5].

Corollary 14. *Conflict analysis can be done in time* $O((|\mathsf{N}| + |\mathsf{E}|) \cdot 2^{\mathsf{poly}(|\mathcal{M}|)})$, *i.e. linear in the program size and exponential in the number of monitors.*

Proof. From Theorems 3, 8, 13, the definitions of R^{op} and α, and the convention that the main-procedure starts with a non-returning call, it is straightforward to show that $\exists c.\ \{[\mathsf{e}_{\mathsf{main}}]\} \xrightarrow{*} c \wedge \mathsf{at}_{\{U,V\}}(c)$ if and only if $R^\sharp[\mathsf{e}_{\mathsf{main}}] \cap \mathsf{D}_2^\sharp \neq \emptyset$. Thus, an algorithm for conflict analysis can compute the least solution of the abstract constraint system and check whether $R^\sharp[\mathsf{e}_{\mathsf{main}}]$ contains an entry from $\mathsf{D}_2^\sharp$. $\square$

Theorem 15. *Deciding whether a given flowgraph has a conflict is NP-complete.*

Proof. We sketch the NP-hardness direction here, that justifies the exponential running time of our algorithm. We show NP-hardness of the reachability problem for a single control node (that can obviously be reduced to conflict detection) by a reduction from 3SAT. For a formula in conjunctive normal form (CNF) $\bigwedge_{1\le i\le n} \bigvee_{1\le j\le 3} l_{ij}$ with $l_{ij} \in \{x_1, \bar{x}_1, ..., x_m, \bar{x}_m\}$, we construct the following program[3] with the procedures main, $p_1, ..., p_{m+1}$,c and monitors $x_1, \bar{x}_1, ..., x_m, \bar{x}_m$:

```
proc main {call $p_1$}
proc $p_i$ /* $1 \le i \le m$ */ {
   {sync ($x_i$) {call $p_{i+1}$}} OR {sync ($\bar{x}_i$) {call $p_{i+1}$}} }
proc $p_{m+1}$ {spawn c; loop forever}
proc c {
   {sync ($l_{11}$){skip}} OR {sync ($l_{12}$){skip}} OR {sync ($l_{13}$){skip}};
   ...
   {sync ($l_{n1}$){skip}} OR {sync ($l_{n2}$){skip}} OR {sync ($l_{n3}$){skip}};
   u: // Control node that is checked to be reachable }
```

The statement `sync` (m) { ... } denotes a block synchronized on monitor m and `OR` denotes nondeterministic choice. Intuitively, the procedures p_1 to p_m guess the values of the variables, where `sync` (x_i) corresponds to setting x_i to false, and vice versa, `sync` $(\bar{x}_i)$ corresponds to setting x_i to true. Finally, procedure q checks whether the clauses are satisfied. Control node u is reachable if and only if the formula is satisfiable.

This construction exploits dynamic thread creation and uses a procedure that cannot terminate. One can do similar constructions for the simultaneous

[3] The translation of this textual representation to our model is straightforward.

reachability of two control nodes in a setting with two fixed threads where all procedures eventually terminate. Thus conflict analysis is also NP-hard for models like the one used in [8]. However, for the model of [8] reachability of a single program point is decidable in polynomial time which highlights the inherent complexity of thread creation.

6 Conclusion

In this paper we studied conflict analysis for a program model with procedure calls, dynamic thread creation and synchronization via reentrant monitors. We showed that conflict analysis is NP-complete. We then used the concept of restricted schedules to come to grips with the arbitrary interleaving between threads. We showed that every reachable configuration is also reachable by an execution with a restricted schedule. We developed a constraint system based characterization of restricted executions, and used abstract interpretation to derive an algorithm for conflict checking that is linear in the program size and exponential only in the number of monitors.

We have developed a formal proof of a similar approach to conflict analysis [10] in Isabelle/HOL [14]. The formalization of the flowgraphs, operational semantics, restricted schedules and acquisition histories are the same as in this paper. The constraint systems follow similar ideas but the abstract constraint systems there are justified directly w.r.t. to the operational semantics instead of using abstract interpretation. Moreover, the height of the abstract domain quadratically depends on the number of procedures. The NP-completeness result has not been formalized in Isabelle/HOL.

Further research required on this topic includes the following: Our algorithm is exponential in the number of monitors. However, for real programs, the nesting depth of monitors is usually significantly smaller than their number. There is strong evidence that this observation can be exploited to design a more efficient analysis. A similar effect was also described in [8] for a model with a fixed set of threads. Furthermore, our algorithm is only able to check for conflicts — while this is an important practical problem, there are other interesting problems like bitvector analysis or high-level data races [1], which may be tackled by generalizing our approach. In order to apply our algorithm to languages with dynamic referencing of monitors (like Java), a preceeding pointer analysis is required. The combination of our analysis with such analyses has to be investigated.

Acknowledgment. We thank Ahmed Bouajjani, Javier Esparza, Nicholas Kidd, Thomas Reps, Helmut Seidl, Bernhard Steffen, Dejvuth Suwimonteerabuth, and Tayssir Touili for interesting discussions, and the anonymous referees for very helpful comments.

References

1. C. Artho, K. Havelund, and A. Biere. High-level data races, 2003.

2. A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *Proc. of CONCUR'05*. Springer, 2005.
3. P. Cousot. Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. *Electronic Notes in Theoretical Computer Science*, 6, 1997. URL: `www.elsevier.nl/locate/entcs/volume6.html`.
4. J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *Proc. of POPL'00*, pages 1–11. Springer, 2000.
5. R. Giacobazzi, F. Ranzato, and F. Scozzari. Making abstract interpretations complete. *Journal of the ACM*, 47(2):361–416, 2000.
6. V. Kahlon and A. Gupta. An automata-theoretic approach for model checking threads for LTL properties. In *Proc. of LICS 2006*, pages 101–110. IEEE Computer Society, 2006.
7. V. Kahlon and A. Gupta. On the analysis of interacting pushdown systems. In *POPL*, pages 303–314, 2007.
8. V. Kahlon, F. Ivancic, and A. Gupta. Reasoning about threads communicating via locks. In *Proc. of CAV 2005*, pages 505–518. Springer, 2005.
9. P. Lammich and M. Müller-Olm. Conflict analysis of programs with procedures, dynamic thread creation, and monitors. Available from `http://cs.uni-muenster.de/u/mmo/pubs/`. Technical Report.
10. P. Lammich and M. Müller-Olm. Formalization of conflict analysis of programs with procedures, thread creation, and monitors. In G. Klein, T. Nipkow, and L. Paulson, editors, *The Archive of Formal Proofs*. http://afp.sf.net/entries/DiskPaxos.shtml, Dec. 2007. Formal proof development.
11. P. Lammich and M. Müller-Olm. Precise fixpoint-based analysis of programs with thread-creation. In *Proc. of CONCUR 2007*, pages 287–302. Springer, 2007.
12. A. Melton, D. A. Schmidt, and G. E. Strecker. Galois connections and computer science applications. In D. Pitt, S. Abramsky, A. Poigné, and D. Rydeheard, editors, *Category Theory and Computer Programming*, volume 240 of *LNCS*, pages 299–312. Springer-Verlag, 1985.
13. M. Müller-Olm. Precise interprocedural dependence analysis of parallel programs. *Theor. Comput. Sci.*, 311(1-3):325–388, 2004.
14. T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.
15. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *TOPLAS*, 22(2):416–430, 2000.
16. H. Seidl and B. Steffen. Constraint-Based Inter-Procedural Analysis of Parallel Programs. *Nordic Journal of Computing (NJC)*, 7(4):375–400, 2000.
17. E. Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. *ACM SIGPLAN Notices*, 36(3):27–40, 2001.