# Precise Fixpoint-Based Analysis of Programs with Thread-Creation and Procedures

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 present a fixpoint-based algorithm for context-sensitive interprocedural kill/gen-analysis of programs with thread creation. Our algorithm is precise up to abstraction of synchronization common in this line of research; it can handle forward as well as backward problems. We exploit a structural property of kill/gen-problems that allows us to analyze the influence of environment actions independently from the local transfer of data flow information. While this idea has been used for programs with parbegin/parend blocks before in work of Knoop/Steffen/Vollmer and Seidl/Steffen, considerable refinement and modification is needed to extend it to thread creation, in particular for backward problems. Our algorithm computes annotations for all program points in time depending linearly on the program size, thus being faster than a recently proposed automata based algorithm by Bouajjani et. al..

## 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. Most papers on precise analysis, such as [5, 13, 10, 9, 3, 4], use parbegin/parend blocks or their interprocedural counterpart, parallel procedure calls, as a model for parallelism. However, this is not adequate for analyzing languages like Java, because in presence of procedures or methods the thread-creation primitives used in such languages cannot be simulated by parbegin/parend [1]. This paper presents an efficient, fixpoint-based algorithm for precise kill/gen-analysis of programs with both thread-creation and parallel calls.

Due to known undecidability and complexity results efficient and precise analyses can only be expected for program models that ignore certain aspects of behavior. As common in this line of research (compare e.g. [13, 10, 9, 3, 4, 1]) we consider flow- and context-sensitive analysis of a program model without synchronization. Note that by a well-known result of Ramalingam [12], context- and synchronization-sensitive analysis is undecidable. We focus on kill/gen problems, a practically relevant class of dataflow problems that comprises the well-known bitvector problems, e.g. live variables, available expressions, etc. Note that only slightly more powerful analyses, like copy constants or truly live variables are intractable or even undecidable (depending on the atomicity of assignments) for parallel programs [10, 9].

Extending previous work [5], Seidl and Steffen proposed an efficient, fixpoint-based algorithm for precise kill/gen-analysis of programs with parallel procedure calls [13]. Adopting their idea of using a separate analysis of possible interference, we construct an algorithm that treats thread creation in addition to parallel call. This extension requires considerable modification. In particular, possible interference has a different nature in presence of thread creation because a thread can survive the procedure that creates it. Also backwards analysis is inherently different from forward analysis in presence of thread creation. As our algorithm handles both thread creation and parallel procedure calls it strictly generalizes Seidl and Steffen's algorithm from [13]. It also treats backwards kill/gen problems for arbitrary programs, while [13] assumes that every forward reachable program point is also backwards reachable.

In [1], an automata based approach to reachability analysis of a slightly stronger program model than ours is presented. In order to compute bitvector analysis information for multiple program points, which is often useful in the context of program optimization, this automata based algorithm must be iterated for each program point, each iteration needing at least linear time in the program size. In contrast, our algorithm computes the analysis information for *all* program points in linear time. Moreover, our algorithm can compute with whole bitvectors, exploiting efficiently implementable bitvector operations, whereas the automata based algorithm must be iterated for each bit. To the best of our knowledge, there has been no precise interprocedural analysis of programs with thread creation that computes information for all program points in linear time.

In a preliminary and less general version of this paper [8], we already covered forward analysis for programs without parallel calls.

This paper is organized as follows: After defining flow graphs and their operational semantics in Section 2, we define the class of kill/gen analysis problems and their *MOP-solution*, using the operational semantics as a reference point (Section 3). We then develop a fixpoint-based characterization of the MOP-solution amenable to algorithmic treatment for both forward and backward problems (Sections 4 and 5), thereby relying on information about the *possible interference*, whose computation is deferred to Section 6. We generalize our treatment to parallel procedure calls in Section 7. Section 8 indicates how to construct a linear-time algorithm from our results and discusses future research.

## 2 Parallel Flow Graphs

A parallel flowgraph $(P, (G_p)_{p \in P})$ consists of a finite set $P$ of procedure names, with $\mathsf{main} \in P$. For each procedure $p \in P$, there is a directed, edge annotated finite graph $G_p = (N_p, E_p, \mathsf{e}_p, \mathsf{r}_p)$ where $N_p$ is the set of control nodes of procedure $p$ and $E_p \subseteq N_p \times \mathcal{A} \times N_p$ is the set of edges that are annotated with base, call or spawn statements: $\mathcal{A} := \{\mathsf{base}\ b \mid b \in \mathcal{B}\} \cup \{\mathsf{call}\ p \mid p \in P\} \cup \{\mathsf{spawn}\ p \mid p \in P\}$.

The set $\mathcal{B}$ of base edge annotations is not specified further for the rest of this paper. Each procedure $p \in P$ has an entry node $\mathsf{e}_p \in N_p$ and a return node $\mathsf{r}_p \in N_p$. As usual we assume that the nodes of the procedures are disjoint, i.e. $N_p \cap N_{p'} = \emptyset$ for $p \neq p'$ and define $N = \bigcup_{p \in P} N_p$ and $E = \bigcup_{p \in P} E_p$.

We use $\mathcal{M}(X)$ to denote the set of multisets of elements from $X$. The empty multiset is $\emptyset$, $\{a\}$ is the multiset containing $a$ once and $A \uplus B$ is multiset union.

We describe the operational semantics of a flowgraph by a labeled transition system $\xrightarrow{\cdot} \subseteq \mathsf{Conf} \times \mathcal{L} \times \mathsf{Conf}$ over configurations $\mathsf{Conf} := \mathcal{M}(N^*)$ and labels $\mathcal{L} := E \cup \{\mathsf{ret}\}$. A configuration consists of the stacks of all threads running in parallel. A stack is modeled as a list of control nodes, the first element being the current control node at the top of the stack. We use $[]$ for the empty list, $[e]$ for the list containing just $e$ and write $r_1 r_2$ for the concatenation of $r_1$ and $r_2$. Execution starts with the initial configuration, $\{[\mathsf{e}_{\mathsf{main}}]\}$. Each transition is labeled with the corresponding edge in the flowgraph or with $\mathsf{ret}$ for the return from a procedure. We use an *interleaving semantics*, nondeterministically picking the thread that performs the next transition among all available threads. Thus we define $\xrightarrow{\cdot}$ by the following rules:

| | | |
|---|---|---|
| [base] | $(\{[u]r\} \uplus c) \xrightarrow{e} (\{[v]r\} \uplus c)$ | for edges $e = (u, \mathsf{base}\ a, v) \in E$ |
| [call] | $(\{[u]r\} \uplus c) \xrightarrow{e} (\{[\mathsf{e}_q][v]r\} \uplus c)$ | for edges $e = (u, \mathsf{call}\ q, v) \in E$ |
| [ret] | $(\{[\mathsf{r}_q]r\} \uplus c) \xrightarrow{\mathsf{ret}} (\{r\} \uplus c)$ | for procedures $q \in P$ |
| [spawn] | $(\{[u]r\} \uplus c) \xrightarrow{e} (\{[v]r\} \uplus \{[\mathsf{e}_q]\} \uplus c)$ | for edges $e = (u, \mathsf{spawn}\ q, v) \in E$ |

We extend $\xrightarrow{e}$ to finite sequences $w \in \mathcal{L}^*$ in the obvious way. For technical reasons, we assume that every edge $e \in E$ of the flowgraph is dynamically reachable, i.e. there is a path of the form $\{[\mathsf{e}_{\mathsf{main}}]\} \xrightarrow{w[e]} \_$ (we use _ as wildcard, i.e. an anonymous, existentially quantified variable). This assumption is harmless, as unreachable edges can be determined by a simple analysis and then removed which does not affect the analysis information we are interested in.

## 3 Dataflow Analysis

Dataflow analysis provides a generic lattice-based framework for constructing program analyses. A specific dataflow analysis is described by a tuple $(L, \sqsubseteq, f)$ where $(L, \sqsubseteq)$ is a complete lattice representing analysis information and $f : \mathcal{L} \to (L \overset{\mathsf{mon}}{\to} L)$ maps a transition label $e$ to a monotonic function $f_e$ that describes how a transition labeled $e$ transforms analysis information. We assume that only base-transitions have transformers other than the identity and extend transformers to finite paths by $f_{e_1 \ldots e_n} := f_{e_n} \circ \ldots \circ f_{e_1}$.

In this paper we consider kill/gen-analyses, i.e. we require $(L, \sqsubseteq)$ to be distributive and the transformers to have the form $f_e(x) = (x \sqcap \mathsf{kill}_e) \sqcup \mathsf{gen}_e$ for some $\mathsf{kill}_e, \mathsf{gen}_e \in L$. Note that all transformers of this form are monotonic and that the set of these transformers is closed under composition of functions. In order to allow effective fixpoint computation, we assume that $(L, \sqsubseteq)$ has finite height. As $(L, \sqsubseteq)$ is distributive, this implies that the meet operation distributes over arbitrary joins, i.e. $(\bigsqcup M) \sqcap x = \bigsqcup\{m \sqcap x \mid m \in M\}$ for all $x \in L$ and $M \subseteq L$. Thus, all transformers are positively disjunctive which is important for precise abstract interpretation.

Kill/gen-analyses comprise classic analyses like determination of live variables, available expressions or potentially uninitialized variables.

Depending on the analysis problem, one distinguishes forward and backward dataflow analyses. The analysis information for forward analysis is an abstraction of the program executions that reach a certain control node, while the backward analysis information concerns executions that leave the control node.

The *forward analysis problem* is to calculate, for each control node $u \in N$, the least upper bound of the transformers of all paths reaching a configuration at control node $u$ applied to an initial value $x_0$ describing the analysis information valid at program start. A configuration $c \in \mathsf{Conf}$ is *at* control node $u \in N$ (we write $\mathsf{at}_u(c)$), iff it contains a stack with top node $u$. We call the solution of the forward analysis problem $\mathsf{MOP}^\mathsf{F}$ and define

$$\mathsf{MOP}^\mathsf{F}[u] := \alpha^\mathsf{F}(\mathsf{Reach}[u])$$

where $\mathsf{Reach}[u] := \{w \mid \exists c : \{[\mathsf{e}_\mathsf{main}]\} \xrightarrow{w} c \wedge \mathsf{at}_u(c)\}$ is the set of paths reaching $u$ and $\alpha^\mathsf{F}(W) := \bigsqcup\{f_w(x_0) \mid w \in W\}$ is the abstraction function from concrete sets of reaching paths to the abstract analysis information we are interested in.[1]

The *backward analysis problem* is to calculate, for each control node $u \in N$, the least upper bound of the transformers of all reversed paths leaving reachable configurations at control node $u$, applied to the least element of $L$, $\bot_L$. We call the solution of the backward analysis problem $\mathsf{MOP}^\mathsf{B}$ and define

$$\mathsf{MOP}^\mathsf{B}[u] := \alpha^\mathsf{B}(\mathsf{Leave}[u])$$

where $\mathsf{Leave}[u] := \{w \mid \exists c : \{[\mathsf{e}_\mathsf{main}]\} \xrightarrow{*} c \xrightarrow{w} \_ \wedge \mathsf{at}_u(c)\}$ is the set of paths leaving $u$, $\alpha^\mathsf{B}(W) := \bigsqcup\{f_{w^R}(\bot_L) \mid w \in W\}$ is the corresponding abstraction function and $w^R$ denotes the word $w$ in reverse order, e.g. $f_{(e_1 \ldots e_n)^R} = f_{e_1} \circ \ldots \circ f_{e_n}$.

Note that we do not use an initial value in the definition of backward analysis, because we have no notion of termination that would give us a point where to apply the initial value. This model is adequate for interactive programs that read and write data while running.

## 4 Forward Analysis

Abstract interpretation [2, 11] is a standard and convenient tool for constructing fixpoint-based analysis algorithms and arguing about their soundness and precision. Thus, a natural idea would be to compute the $\mathsf{MOP}^\mathsf{F}$-solution as an abstract interpretation of a system of equations or inequations (*constraint system*) that characterizes the sets $\mathsf{Reach}[u]$.

[1] MOP originally stands for meet over all paths. We use the dual lattice here, but stick to the term MOP for historical reasons.

Unfortunately, it follows from results in [1] that no such constraint system exists in presence of thread creation and procedures (using the natural operators "concatenation" and "interleaving" from [13]). In order to avoid this problem, we derive an alternative characterization of the $\mathsf{MOP}$-solution as the join of two values, each of which can be captured by abstract interpretation of appropriate constraint systems. In order to justify this alternative characterization, we argue at the level of program paths. That is, we perform part of the abstraction already on the path level. More specifically, we classify the transitions of a reaching path into *directly reaching transitions* and *interfering transitions* and show that these transitions are quite independent. We then show how to obtain the MOP-solution from the set of *directly reaching paths* (consisting of directly reaching transitions only) and the *possible interference* (the set of interfering transitions on reaching paths), and how to characterize these sets as constraint systems. The idea of calculating the MOP-solution using possible interference is used already by [13] in a setting with parallel procedure calls. However, while in [13] this idea is used just in order to reduce the size of the constraint system, in our case it is essential in order to obtain a constraint-based characterization at all, due to results of [1] mentioned above.

The classification of transitions is illustrated by Fig. 1. The vertical lines symbolize the executions of single threads, horizontal arrows are thread creation. The path depicted in this figure reaches the control node $u$ in one thread. The directly reaching transitions are marked with thick lines. The other transitions are interfering transitions, which are executed concurrently with the directly reaching transitions, so that the whole path is some interleaving of directly reaching and interfering transitions.
A key observation is that due to the lack of synchronization each potentially interfering transition $e$ can take place at the very end of some path reaching $u$; thus, the information at $u$ cannot be stronger than $\mathsf{gen}_e$. In order to account for this, we use the least upper bound of all this $\mathsf{gen}_e$-values (see Theorem 2 and the definition of $\alpha^{\mathsf{PI}}$ below). This already covers the effect of interfering transitions completely: the $\mathsf{kill}_e$-parts have no strengthening effect for the information at $u$, because the directly reaching path reaches $u$ without executing any of the interfering transitions.

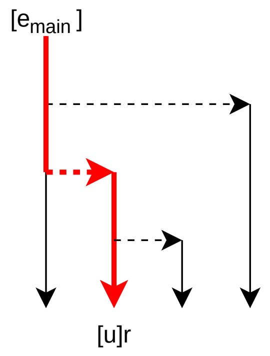

**Fig. 1:** Directly reaching and interfering transitions.

In order to formalize these ideas, we distinguish directly reaching from interfering transitions in the operational semantics by marking one stack of a configuration as executing directly reaching transitions. Transitions of unmarked stacks are interfering ones. If the marked stack executes a spawn, the marker can either stay with this stack or be transferred to the newly created thread. In Fig. 1 this corresponds to pushing the marker along the thick lines.

In the actual formalization, we mark a single control node in a stack instead of the stack as a whole. This is mainly in order to allow us a more smooth generalization to the case of parallel procedure calls (Section 7). In a procedure call from a marked node we either move the marker up the stack to the level of the newly created procedure or retain it at the level of the calling procedure. Note that we can move the marker from the initial configuration $\{[\mathsf{e}_{\mathsf{main}}]\}$ to any reachable control node $u$ by just using transitions on control nodes above the marker. These transitions formalize the directly reaching transitions. The notation for a node that may be marked is $u^m$, with $m \in \{\circ, \bullet\}$ and $u \in N$ where $u^\circ$ means that the node is not marked, and $u^\bullet$ means that the node is marked. We now define the following rule templates, instantiated for different values of $x$ below:

$$\begin{array}{lll}
\text{[base]} & (\{[u^m]r\} \uplus c)\xrightarrow{e}_x(\{[v^m]r\} \uplus c) & e=(u,\mathsf{base}\ a,v)\in E \\
\text{[call]} & (\{[u^m]r\} \uplus c)\xrightarrow{e}_x(\{[\mathsf{e}_q^\circ][v^m]r\} \uplus c) & e=(u,\mathsf{call}\ q,v)\in E \\
\text{[ret]} & (\{[\mathsf{r}_q^\circ]r\} \uplus c)\xrightarrow{\mathsf{ret}}_x(\{r\} \uplus c) & q\in P \\
\text{[spawn]} & (\{[u^m]r\} \uplus c)\xrightarrow{e}_x(\{[v^m]r\} \uplus \{[\mathsf{e}_q^\circ]\} \uplus c) & e=(u,\mathsf{spawn}\ q,v)\in E
\end{array}$$

Using these templates, we define the transition relations $\overset{\cdot}{\longrightarrow}_{\mathsf{m}}$ and $\overset{\cdot}{\longrightarrow}_{\mathsf{i}}$. The relation $\overset{\cdot}{\longrightarrow}_{\mathsf{m}}$ is defined by adding the additional side condition that some node in $[u^m]r$ must be marked to the rules [base], [call] and [spawn]. The [ret]-rule gets the additional condition that some node in $r$ must be marked (in particular, $r$ must not be empty). The relation $\overset{\cdot}{\longrightarrow}_{\mathsf{i}}$ is defined by adding the condition that no node in $[u^m]r$ or $r$ respectively must be marked.

Intuitively, $\overset{\cdot}{\longrightarrow}_{\mathsf{m}}$ describes transitions on marked stacks only, but cannot change the position of the marker; $\overset{\cdot}{\longrightarrow}_{\mathsf{i}}$ captures interfering transitions. To be able to push the marker to called procedures or spawned threads, we define the transition relation $\overset{\cdot}{\longrightarrow}_{\mathsf{p}}$ by the following rules:

$$\begin{array}{lll}
\text{[call.push]} & (\{[u^\bullet]r\} \uplus c)\xrightarrow{e}_{\mathsf{p}}(\{[\mathsf{e}_q^\bullet][v^\circ]r\} \uplus c) & e=(u,\mathsf{call}\ q,v)\in E \\
\text{[spawn.push]} & (\{[u^\bullet]r\} \uplus c)\xrightarrow{e}_{\mathsf{p}}(\{[v^\circ]r\} \uplus \{[\mathsf{e}_q^\bullet]\} \uplus c) & e=(u,\mathsf{spawn}\ q,v)\in E
\end{array}$$

According to the ideas described above, we get:

**Lemma 1.** *Given a reaching path* $\{[\mathsf{e}_{\mathsf{main}}]\}\xrightarrow{w}\{[u]r\} \uplus c$*, there are paths* $w_1, w_2$ *with* $w \in w_1 \otimes w_2$*, such that* $\exists \hat{c} : \{[\mathsf{e}_{\mathsf{main}}^\bullet]\}\xrightarrow{w_1}_{\mathsf{mp}}\{[u^\bullet]r\} \uplus \hat{c}\xrightarrow{w_2}_{\mathsf{i}}\{[u^\bullet]r\} \uplus c$.

Here, $w_1 \otimes w_2$ denotes the set of all interleavings of the finite sequences $w_1$ and $w_2$ and $\overset{\cdot}{\longrightarrow}_{\mathsf{mp}} := \overset{\cdot}{\longrightarrow}_{\mathsf{m}} \cup \overset{\cdot}{\longrightarrow}_{\mathsf{p}}$ executes the directly reaching transitions, resulting in the configuration $\{[u^\bullet]r\} \uplus \hat{c}$. The interfering transitions in $w_2$ operate on the threads from $\hat{c}$. These threads are either freshly spawned and hence in their initial configuration with just the thread's entry point on the stack, or they have been left by a transition according to rule [spawn.push] and hence are at the target node of the spawn edge and may have some return nodes on the stack.

Now we define the set $\mathsf{R}^{\mathsf{op}}[u]$ of directly reaching paths to $u$ as

$$\mathsf{R}^{\mathsf{op}}[u] := \{w \mid \exists r, c : \{[\mathsf{e}_{\mathsf{main}}^\bullet]\}\xrightarrow{w}_{\mathsf{mp}}\{[u^\bullet]r\} \uplus c\}$$

and the possible interference at $u$ as

$$\mathsf{PI}^{\mathsf{op}}[u] := \{e \mid \exists r, c, w : \{[\mathsf{e}_{\mathsf{main}}^\bullet]\}\xrightarrow{*}_{\mathsf{mp}}\{[u^\bullet]r\} \uplus c\xrightarrow{w[e]}_{\mathsf{i}}\_\}\,.$$

The following theorem characterizes the $\mathsf{MOP}^\mathsf{F}$-solution based on $\mathsf{R}^\mathsf{op}$ and $\mathsf{PI}^\mathsf{op}$:

**Theorem 2.** $\mathsf{MOP}^\mathsf{F}[u] = \alpha^\mathsf{F}(\mathsf{R}^\mathsf{op}[u]) \sqcup \alpha^\mathsf{PI}(\mathsf{PI}^\mathsf{op}[u])$

Here, $\alpha^\mathsf{PI}(E) := \bigsqcup\{\mathsf{gen}_e \mid e \in E\}$ abstracts sets of edges to the least upper bound of their $\mathsf{gen}_e$-values.

*Proof.* For the $\sqsubseteq$-direction, we fix a reaching path $w \in \mathsf{Reach}[u]$ and show that its abstraction $f_w(x_0)$ is smaller than the right hand side. Using Lemma 1 we split $w$ into the directly reaching path $w_1$ and the interfering transitions $w_2$, such that $w \in w_1 \otimes w_2$. Because we use kill/gen-analysis over distributive lattices, we have the approximation $f_w(x_0) \sqsubseteq f_{w_1}(x_0) \sqcup \bigsqcup\{\mathsf{gen}_e \mid e \in w_2\}$ [13]. Obviously, these two parts are smaller than $\alpha^\mathsf{F}(\mathsf{R}^\mathsf{op}[u])$ and $\alpha^\mathsf{PI}(\mathsf{PI}^\mathsf{op}[u])$ respectively, and thus the proposition follows.

For the $\sqsupseteq$-direction, we first observe that any directly reaching path is also a reaching path, hence $\mathsf{MOP}^\mathsf{F}[u] \sqsupseteq \alpha^\mathsf{F}(\mathsf{R}^\mathsf{op}[u])$. Moreover, for each transition $e \in \mathsf{PI}^\mathsf{op}[u]$ a path $w[e] \in \mathsf{Reach}[u]$ can be constructed. Its abstraction $(f_w(x_0) \sqcap \mathsf{kill}_e) \sqcup \mathsf{gen}_e$ is obviously greater than $\mathsf{gen}_e$. Thus, also $\mathsf{MOP}^\mathsf{F}[u] \sqsupseteq \alpha^\mathsf{PI}(\mathsf{PI}^\mathsf{op}[u])$. Altogether the proposition follows. □

*Constraint systems.* In order to compute the right hand side of the equation in Theorem 2 by abstract interpretation, we characterize the directly reaching paths and the possible interference as the least solutions of constraint systems. We will focus on the directly reaching paths here. The constraints for the possible interference are developed in Section 6, because we can reuse results from backward analysis for their characterization. In order to precisely treat procedures, we use a well-known technique from interprocedural program analysis, that first characterizes so called *same-level paths* and then uses them to assemble the directly reaching paths. A same-level path starts and ends at the same stack-level, and never returns below this stack level. We are interested in the same-level paths starting at the entry node of a procedure and ending at some node $u$ of this procedure. We define the set of these paths as $\mathsf{S}^\mathsf{op}[u] := \{w \mid \exists c : \{[\mathsf{e}_p^\bullet]\} \xrightarrow{w}_\mathsf{m} \{[u^\bullet]\} \uplus c\}$ for $u \in N_p$. It is straightforward to show that $\mathsf{lfp}(\mathsf{S}) = \mathsf{S}^\mathsf{op}$ for the least solution $\mathsf{lfp}(\mathsf{S})$ of the constraint system $\mathsf{S}$ over variables $\mathsf{S}[u] \in \mathcal{P}(\mathcal{L}^*)$, $u \in N$ with the following constraints:

| | | |
|---|---|---|
| [init] | $\mathsf{S}[\mathsf{e}_q] \supseteq \{\varepsilon\}$ | for $q \in P$ |
| [base] | $\mathsf{S}[v] \supseteq \mathsf{S}[u]; e$ | for $e = (u, \mathsf{base}\ \_, v) \in E$ |
| [call] | $\mathsf{S}[v] \supseteq \mathsf{S}[u]; e; \mathsf{S}[\mathsf{r}_q]; \mathsf{ret}$ | for $e = (u, \mathsf{call}\ q, v) \in E$ |
| [spawn] | $\mathsf{S}[v] \supseteq \mathsf{S}[u]; e$ | for $e = (u, \mathsf{spawn}\ q, v) \in E$ |

The operator ; is list concatenation lifted to sets. The directly reaching paths are characterized by the constraint system $\mathsf{R}$ over variables $\mathsf{R}[u] \in \mathcal{P}(\mathcal{L}^*)$, $u \in N$ with the following constraints:

| | | |
|---|---|---|
| [init] | $\mathsf{R}[\mathsf{e}_\mathsf{main}] \supseteq \{\varepsilon\}$ | |
| [reach] | $\mathsf{R}[u] \supseteq \mathsf{R}[\mathsf{e}_p]; \mathsf{S}^\mathsf{op}[u]$ | for $u \in N_p$ |
| [callp] | $\mathsf{R}[\mathsf{e}_q] \supseteq \mathsf{R}[u]; e$ | for $e = (u, \mathsf{call}\ q, \_) \in E$ |
| [spawnp] | $\mathsf{R}[\mathsf{e}_q] \supseteq \mathsf{R}[u]; e$ | for $e = (u, \mathsf{spawn}\ q, \_) \in E$ |

Intuitively, the constraint [reach] corresponds to the transitions that can be performed by the $\stackrel{\cdot}{\longrightarrow}_{\mathsf{m}}$ part of $\stackrel{\cdot}{\longrightarrow}_{\mathsf{mp}}$, and the [callp]- and [spawnp]-constraints correspond to the $\stackrel{\cdot}{\longrightarrow}_{\mathsf{p}}$ part. It is again straightforward to show $\mathsf{lfp}(\mathsf{R}) = \mathsf{R}^{\mathsf{op}}$.

Using standard techniques of abstract interpretation [2, 11], we can construct an abstract version $\mathsf{R}^{\#}$ of $\mathsf{R}$ over the domain $(L, \sqsubseteq)$ using an abstract version $\mathsf{S}^{\#}$ of $\mathsf{S}$ over the domain $(L \overset{\mathsf{mon}}{\rightarrow} L, \sqsubseteq)$ and show:

**Theorem 3.** $\mathsf{lfp}(\mathsf{R}^{\#}) = \alpha^{\mathsf{F}}(\mathsf{lfp}(\mathsf{R}))$.

## 5 Backward Analysis

For backward analysis, we consider the paths leaving $u$. Recall that these are the paths starting at a reachable configuration of the form $\{[u]r\} \uplus c$. Such a path is an interleaving of a path from $[u]r$ and transitions originating from $c$. The latter ones are covered by the possible interference $\mathsf{PI}^{\mathsf{op}}[u]$. It turns out that in order to come to grips with this interleaving we can use a similar technique as for forward analysis. We define the *directly leaving paths* as

$$\mathsf{L}^{\mathsf{op}}[u] := \{w \mid \exists r, c : \{[\mathsf{e}_{\mathsf{main}}]\} \xrightarrow{*} \{[u]r\} \uplus c \wedge \{[u]r\} \xrightarrow{w} \_\}$$

and show the following characterization:

**Theorem 4.** $\mathsf{MOP}^{\mathsf{B}}[u] = \alpha^{\mathsf{B}}(\mathsf{L}^{\mathsf{op}}[u]) \sqcup \alpha^{\mathsf{PI}}(\mathsf{PI}^{\mathsf{op}}[u])$.

The proof is similar to that of Theorem 2. It is deferred to the appendix of [7] due to lack of space.

In the forward case, the set of directly reaching paths could be easily described by a constraint system on sets of paths. The analogous set of directly leaving paths, however, does not appear to have such a simple characterization, because the concurrency effects caused by threads created on these paths have to be tackled. This is hard in combination with procedures, as threads created inside an instance of a procedure can survive termination of that instance. In order to treat this effect, we have experimented with a complex constraint system on sets of pairs of paths. It turned out that this complexity disappears in the abstract version of this constraint system. In order to give a more transparent justification for the resulting abstract constraint systems, we develop – again arguing on the path level – an alternative characterization of $\alpha^{\mathsf{B}}(\mathsf{L}^{\mathsf{op}}[u])$ through a subset of representative paths that is easy to characterize. Thus, again we transfer part of the abstraction to the path level.

More specifically, we only consider directly leaving paths that execute transitions of a created thread immediately after the corresponding spawn transition. From the point of view of the initial thread from which the path is leaving, the transitions of newly created threads are executed as early as possible. Formally, we define the relation $\stackrel{\cdot}{\Longrightarrow}_{\mathsf{x}} \subseteq \mathsf{Conf} \times \mathcal{L}^* \times \mathsf{Conf}$ by the following rules:

$$\begin{array}{ll}
c \overset{e}{\Longrightarrow}_{\mathsf{x}} c' & \text{if } c \xrightarrow{e}_{x} c' \text{ and } e \text{ is no spawn edge} \\
c \overset{[e]w}{\Longrightarrow}_{\mathsf{x}} c' & \text{if } c \xrightarrow{e}_{x} c' \uplus \{[\mathsf{e}_p]\},\ e = \mathsf{spawn}\ p \text{ and } \{[\mathsf{e}_p]\} \xrightarrow{w} \_ \\
c \overset{w_1 w_2}{\Longrightarrow}_{\mathsf{x}} c' & \text{if } \exists \hat{c} : c \overset{w_1}{\Longrightarrow}_{\mathsf{x}} \hat{c} \wedge \hat{c} \overset{w_2}{\Longrightarrow}_{\mathsf{x}} c'
\end{array}$$

Here $x$ selects some set of transition rules, i.e. $x = \mathsf{mp}$ means that $\overset{\cdot}{\longrightarrow}_{\mathsf{mp}}$ is used for $\overset{\cdot}{\longrightarrow}_x$. If $x$ is empty, the standard transition relation $\overset{\cdot}{\longrightarrow}$ is used.

The set of representative directly leaving paths is defined by

$$\mathsf{L}^{\mathsf{op}}_{\subseteq}[u] := \{w \mid \exists r, c : \{[\mathsf{e}_{\mathsf{main}}]\} \xrightarrow{*} \{[u]r\} \uplus c \wedge \{[u]r\} \xRightarrow{w} \_ \} \,.$$

Exploiting structural properties of kill/gen-functions, we can show:

**Lemma 5.** *For each $u \in N$ we have $\alpha^{\mathsf{B}}(\mathsf{L}^{\mathsf{op}}[u]) = \alpha^{\mathsf{B}}(\mathsf{L}^{\mathsf{op}}_{\subseteq}[u])$.*

*Proof.* The $\sqsupseteq$-direction is trivial, because we obviously have $\mathsf{L}^{\mathsf{op}}[u] \supseteq \mathsf{L}^{\mathsf{op}}_{\subseteq}[u]$ and $\alpha^{\mathsf{B}}$ is monotonic. For the $\sqsubseteq$-direction we consider a directly leaving path $\{[u]r\} \xrightarrow{w} \_$ with $w = e_1 \dots e_n$. Due to the distributivity of $L$, its abstraction can be written as $f_{w^R}(\bot_L) = \bigsqcup_{1 \le i \le n} (\mathsf{gen}_{e_i} \sqcap A_i)$ with $A_i := \mathsf{kill}_{e_1} \sqcap \dots \sqcap \mathsf{kill}_{e_{i-1}}$.

We show for each edge $e_k$ that the value $\mathsf{gen}_{e_k} \sqcap A_k$ is below $\alpha^{\mathsf{B}}(\mathsf{L}^{\mathsf{op}}_{\subseteq}[u])$. For this, we distinguish whether transition $e_k$ was executed in the initial thread (from stack $[u]r$) or in some spawned thread. To cover the case of a transition $e_k$ of the initial thread, we consider the subpath $w' \in \mathsf{L}^{\mathsf{op}}_{\subseteq}[u]$ of $w$ that makes no transitions of spawned threads at all. We can obtain $w'$ by discarding some transitions from $w$. Moreover, $w'$ also contains the transition $e_k$. If we write $f_{w'^R}(\bot_L)$ in a similar form as above, it contains a term $\mathsf{gen}_{e_k} \sqcap A'$, and because we discarded some transitions, we have $A' \sqsupseteq A_k$, and hence $f_{w'^R}(\bot_L) \sqsupseteq \mathsf{gen}_{e_k} \sqcap A' \sqsupseteq \mathsf{gen}_{e_k} \sqcap A_k$.

To cover the case of a transition $e_j$ of a spawned thread, we consider the subpath $w'' \in \mathsf{L}^{\mathsf{op}}_{\subseteq}[u]$ of $w$ that, besides directly leaving ones, only contains transitions of the considered thread. Because $e_j$ occurs as early as possible in $w''$, the prefix of $w''$ up to $e_j$ can be derived from the prefix of $w$ up to $e_j$ by discarding some transitions, and again we get $f_{w''^R}(\bot_L) \sqsupseteq \mathsf{gen}_{e_j} \sqcap A_j$. □

We can characterize $\mathsf{L}^{\mathsf{op}}_{\subseteq}$ by the following constraint system:

| | | |
|---|---|---|
| [LS.init] | $\mathsf{LS}[u] \supseteq \{\varepsilon\}$ | |
| [LS.init2] | $\mathsf{LS}[r_p] \supseteq \{[\mathsf{ret}]\}$ | for $p \in P$ |
| [LS.base] | $\mathsf{LS}[u] \supseteq e; \mathsf{LS}[v]$ | for $e = (u, \mathsf{base}\ \_, v) \in E$ |
| [LS.call1] | $\mathsf{LS}[u] \supseteq e; \mathsf{LS}[\mathsf{e}_p]$ | for $(u, \mathsf{call}\ p, v) \in E$ |
| [LS.call2] | $\mathsf{LS}[u] \supseteq e; \mathsf{S_B}[\mathsf{e}_p]; \mathsf{ret}; \mathsf{LS}[v]$ | for $(u, \mathsf{call}\ p, v) \in E$ |
| [LS.spawn] | $\mathsf{LS}[u] \supseteq e; \mathsf{LS}[\mathsf{e}_p]; \mathsf{LS}[v]$ | for $(u, \mathsf{spawn}\ p, v) \in E$ |
| [SB.init] | $\mathsf{S_B}[\mathsf{r}_p] \supseteq \{\varepsilon\}$ | |
| [SB.base] | $\mathsf{S_B}[u] \supseteq e; \mathsf{S_B}[v]$ | for $e = (u, \mathsf{base}\ \_, v) \in E$ |
| [SB.call] | $\mathsf{S_B}[u] \supseteq e; \mathsf{S_B}[\mathsf{e}_p]; \mathsf{ret}; \mathsf{S_B}[v]$ | for $(u, \mathsf{call}\ p, v) \in E$ |
| [SB.spawn] | $\mathsf{S_B}[u] \supseteq e; \mathsf{LS}[\mathsf{e}_p]; \mathsf{S_B}[v]$ | for $(u, \mathsf{spawn}\ p, v) \in E$ |
| [L.leave1] | $\mathsf{L}_{\subseteq}[u] \supseteq \mathsf{S_B}[u]; \mathsf{L}_{\subseteq}[\mathsf{r}_p]$ | for $u \in N_p$ if $u$ reachable |
| [L.leave2] | $\mathsf{L}_{\subseteq}[u] \supseteq \mathsf{LS}[u]$ | if $u$ reachable |
| [L.ret] | $\mathsf{L}_{\subseteq}[\mathsf{r}_p] \supseteq \mathsf{ret}; \mathsf{L}_{\subseteq}[v]$ | for $(\_, \mathsf{call}\ p, v) \in E$ and $p$ can terminate |

The $\mathsf{LS}$ part of the constraint system characterizes paths from a single control node: $\mathsf{LS}^{\mathsf{op}}[u] := \{w \mid \{[u]\} \overset{w}{\Longrightarrow} \_\}$. The $\mathsf{S_B}$-part characterizes same-level paths from a control node to the return node of the corresponding procedure: $\mathsf{S}^{\mathsf{op}}_{\mathsf{B}}[u] := \{w \mid \exists c' : \{[u^\bullet]\} \overset{w}{\Longrightarrow}_{\mathsf{m}} \{[\mathsf{r}^\bullet_p]\} \uplus c'\}$. It is straightforward to prove $\mathsf{lfp}(\mathsf{LS}) = \mathsf{LS}^{\mathsf{op}}$, $\mathsf{lfp}(\mathsf{S_B}) = \mathsf{S}^{\mathsf{op}}_{\mathsf{B}}$ and $\mathsf{lfp}(\mathsf{L}_\subseteq) = \mathsf{L}^{\mathsf{op}}_\subseteq$. Using abstract interpretation one gets constraint systems $\mathsf{L}_\subseteq{}^\#$ over $(L, \subseteq)$ and $\mathsf{LS}^\#, \mathsf{S_B}^\#$ over $(L \overset{\mathsf{mon}}{\to} L, \sqsubseteq)$ with $\mathsf{lfp}(\mathsf{L}_\subseteq{}^\#) = \alpha^{\mathsf{B}}(\mathsf{lfp}(\mathsf{L}_\subseteq))$.

## 6 Possible Interference

In order to be able to compute the forward and backward $\mathsf{MOP}$-solution, it remains to describe a constraint system based characterization of the possible interference suitable for abstract interpretation. We use the following constraints:

| | | |
|---|---|---|
| [SP.edge] | $\mathsf{SP}[v] \supseteq \mathsf{SP}[u]$ | for $(u, \mathsf{base}\ \_, v) \in E$ or $(u, \mathsf{spawn}\ \_, v) \in E$ |
| [SP.call] | $\mathsf{SP}[v] \supseteq \mathsf{SP}[u] \cup \mathsf{SP}[\mathsf{r}_q]$ | for $(u, \mathsf{call}\ q, v) \in E$ if $q$ can terminate |
| [SP.spawnt] | $\mathsf{SP}[v] \supseteq \alpha^{\mathsf{E}}(\mathsf{LS}^{\mathsf{op}}[\mathsf{e}_q])$ | for $(u, \mathsf{spawn}\ q, v) \in E$ |
| [PI.reach] | $\mathsf{PI}[u] \supseteq \mathsf{PI}[\mathsf{e}_p] \cup \mathsf{SP}[u]$ | for $u \in N_p$ and $u$ reachable |
| [PI.trans1] | $\mathsf{PI}[\mathsf{e}_q] \supseteq \mathsf{PI}[u]$ | for $(u, \mathsf{call}\ q, \_) \in E$ |
| [PI.trans2] | $\mathsf{PI}[\mathsf{e}_q] \supseteq \mathsf{PI}[u]$ | for $(u, \mathsf{spawn}\ q, \_) \in E$ |
| [PI.trans3] | $\mathsf{PI}[\mathsf{e}_q] \supseteq \alpha^{\mathsf{E}}(\mathsf{L}^{\mathsf{op}}_\subseteq[v])$ | for $(u, \mathsf{spawn}\ q, v) \in E$ |

Here, $\alpha^{\mathsf{E}} : \mathcal{P}(\mathcal{L}^*) \to \mathcal{P}(\mathcal{L})$ with $\alpha^{\mathsf{E}}(W) = \{e \mid \exists w, e, w' : w[e]w' \in W\}$ abstracts sets of paths to the sets of transitions contained in the paths. The constraint system $\mathsf{PI}$ follows the same-level pattern: $\mathsf{SP}$ characterizes the interfering transitions that are enabled by same-level paths. It is straightforward to show $\mathsf{lfp}(\mathsf{SP}) = \mathsf{SP}^{\mathsf{op}}$, with $\mathsf{SP}^{\mathsf{op}}[u] := \{e \mid \exists c, w : \{[\mathsf{e}^\bullet_p]\} \overset{*}{\longrightarrow}_{\mathsf{m}} \{[u^\bullet]\} \uplus c \overset{w[e]}{\longrightarrow}_{\mathsf{i}} \_\}$. The constraint [PI.reach] captures that the possible interference at a reachable node $u$ is greater than the possible interference at the beginning of $u$'s procedure and the interference created by same-level paths to $u$. The [PI.trans1]- and [PI.trans2]-constraints describe that the interference at the entry point of a called or spawned procedure is greater than the interference at the start node of the call resp. spawn edge. The [PI.trans3]-constraint accounts for the interference generated in the spawned thread by the creator thread continuing its execution. Because the creator thread may be inside a procedure, we have to account not only for edges inside the current procedure, but also for edges of procedures the creator thread may return to. These edges are captured by $\alpha^{\mathsf{E}}(\mathsf{L}^{\mathsf{op}}[v]) = \alpha^{\mathsf{E}}(\mathsf{L}^{\mathsf{op}}_\subseteq[v])$.

With the ideas described above, it is straightforward to show $\mathsf{lfp}(\mathsf{PI}) = \mathsf{PI}^{\mathsf{op}}$. Abstraction of the $\mathsf{PI}$- and $\mathsf{SP}$-systems is especially simple in this case, as the constraint systems only contain variables and constants. For the abstract versions $\mathsf{SP}^\#$ and $\mathsf{PI}^\#$, we have $\mathsf{lfp}(\mathsf{SP}^\#) = \alpha^{\mathsf{PI}}(\mathsf{lfp}(\mathsf{SP}))$ and $\mathsf{lfp}(\mathsf{PI}^\#) = \alpha^{\mathsf{PI}}(\mathsf{lfp}(\mathsf{PI}))$.

Now, we have all pieces to compute the forward and backward $\mathsf{MOP}$-solutions: Combining Theorems 2, 4 and Lemma 5 with the statements about the abstract constraint systems we get

$$\mathsf{MOP}^{\mathsf{F}}[u] = \mathsf{lfp}(\mathsf{R}^{\#})[u] \sqcup \mathsf{lfp}(\mathsf{PI}^{\#})[u] \text{ and } \mathsf{MOP}^{\mathsf{B}}[u] = \mathsf{lfp}(\mathsf{L}_{\subseteq}{}^{\#})[u] \sqcup \mathsf{lfp}(\mathsf{PI}^{\#})[u]\,.$$

The right hand sides are efficiently computable, e.g. by a worklist algorithm [11].

## 7 Parallel Calls

In this section we discuss the extension to parallel procedure calls. Two procedures that are called in parallel are executed concurrently, but the call does not return until both procedures have terminated.

*Flowgraphs.* In the flowgraph definition, we replace the $\mathsf{call}\ p$ annotation by the $\mathsf{pcall}\ p_1 \parallel p_2$ annotation, where $p_1, p_2 \in P$ are the procedures called in parallel. Note that there is no loss of expressiveness by assuming that all procedure calls are parallel calls, because instead of calling a procedure $p$ alone, one can call it in parallel with a procedure $q_0$, where $q_0$ has only a single node $\mathsf{e}_{q_0} = \mathsf{r}_{q_0}$ with no outgoing edges. To describe a configuration, the notion of a stack is extended from a linear list to a binary tree. While in the case without parallel calls, the topmost node of the stack can make transitions and the other nodes are the stored return addresses, now the leafs of the tree can make transitions and the inner nodes are the stored return addresses. We write $u$ for the tree consisting just of node $u$, and $u(t, t')$ for the tree with root $u$ with the two successor trees $t$ and $t'$. The notation $r[t]$ denotes a tree consisting of a subtree $t$ in some context $r$. The position of $t$ in $r$ is assumed to be fixed, such that writing $r[t]$ and $r[t']$ in the same expression means that $t$ and $t'$ are at the same position in $r$.

The rule templates for $\overset{\cdot}{\longrightarrow}_{\mathsf{m}}$, $\overset{\cdot}{\longrightarrow}_{\mathsf{i}}$ and $\overset{\cdot}{\longrightarrow}_{\mathsf{p}}$ are refined as follows:

$$\begin{array}{lll}
[\text{base}] & (\{r[u^m]\} \uplus c) \xrightarrow{e}_x (\{r[v^m]\} \uplus c) & e = (u, \mathsf{base}\ a, v) \in E \\
[\text{pcall}] & (\{r[u^m]\} \uplus c) \xrightarrow{e}_x (\{r[v^m(\mathsf{e}_p^\circ, \mathsf{e}_q^\circ)]\} \uplus c) & e = (u, \mathsf{pcall}\ p \parallel q, v) \in E \\
[\text{ret}] & (\{r[v^m(\mathsf{r}_p^\circ, \mathsf{r}_q^\circ)]\} \uplus c) \xrightarrow{\mathsf{ret}}_x (\{r[v^m]\} \uplus c) & p, q \in P \\
[\text{spawn}] & (\{r[u^m]\} \uplus c) \xrightarrow{e}_x (\{r[v^m]\} \uplus \{\mathsf{e}_q^\circ\} \uplus c) & e = (u, \mathsf{spawn}\ q, v) \in E \\
 & & \\
[\text{c.pushl}] & (\{r[u^\bullet]\} \uplus c) \xrightarrow{e}_{\mathsf{p}} (\{r[v^\circ(\mathsf{e}_p^\bullet, \mathsf{e}_q^\circ)]\} \uplus c) & e = (u, \mathsf{pcall}\ p \parallel q, v) \in E \\
[\text{c.pushr}] & (\{r[u^\bullet]\} \uplus c) \xrightarrow{e}_{\mathsf{p}} (\{r[v^\circ(\mathsf{e}_p^\circ, \mathsf{e}_q^\bullet)]\} \uplus c) & e = (u, \mathsf{pcall}\ p \parallel q, v) \in E \\
[\text{sp.push}] & (\{r[u^\bullet]\} \uplus c) \xrightarrow{e}_{\mathsf{p}} (\{r[v^\circ]\} \uplus \{\mathsf{e}_q^\bullet\} \uplus c) & e = (u, \mathsf{spawn}\ q, v) \in E
\end{array}$$

For the $\overset{\cdot}{\longrightarrow}_{\mathsf{m}}$-relation, we require the position of the processed node $u^m$ resp. subtree $v^m(\mathsf{r}_p^\circ, \mathsf{r}_q^\circ)$ in $r$ to be below a marked node. For the $\overset{\cdot}{\longrightarrow}_{\mathsf{i}}$-relation the position must not be below a marked node. The reference semantics $\overset{\cdot}{\longrightarrow}$ on unmarked configurations is defined by analogous rules.

*Forward Analysis.* Again we can use the relation $\overset{\cdot}{\longrightarrow}_{\mathsf{mp}}$ to push an initial marker to any reachable node. Although there is some form of synchronization at procedure return now, the $\overset{\cdot}{\longrightarrow}_{\mathsf{m}}$ and $\overset{\cdot}{\longrightarrow}_{\mathsf{i}}$-relations are defined in such a way that the interfering transitions can again be moved to the end of a reaching path and in analogy to Lemma 1 we get:

**Lemma 6.** *Given a reaching path* $\{\mathsf{e}_{\mathsf{main}}\}\xrightarrow{w}\{r[u]\}\uplus c$*, there exists paths* $w_1, w_2$ *with* $w \in w_1 \otimes w_2$ *such that* $\exists \hat{c}, \hat{r} : \{\mathsf{e}^\bullet_{\mathsf{main}}\}\xrightarrow{w_1}_{\mathsf{mp}}\{\hat{r}[u^\bullet]\} \uplus \hat{c}\xrightarrow{w_2}_{\mathsf{i}}\{r[u^\bullet]\} \uplus c$.

Note that the interfering transitions may work not only on the threads in $\hat{c}$, but also on the nodes of $\hat{r}$ that are no predecessors of $u^\bullet$, i.e. on procedures called in parallel that have not yet terminated.

We redefine the directly reaching paths $\mathsf{R}^{\mathsf{op}}$ and possible interference $\mathsf{PI}^{\mathsf{op}}$ accordingly, and get the same characterization of $\mathsf{MOP}^{\mathsf{F}}$ as in the case without parallel calls (Theorem 2). In $\mathsf{S}$ and $\mathsf{R}$, we replace the constraints for call edges with the following constraints for parallel procedure calls:

| | | |
|---|---|---|
| [call] | $\mathsf{S}[v] \supseteq \mathsf{S}[u]; e; (\mathsf{S}[\mathsf{r}_p] \otimes \mathsf{S}[\mathsf{r}_q]); \mathsf{ret}$ | for $e = (u, \mathsf{pcall}\ p \parallel q, v) \in E$ |
| [callp1] | $\mathsf{R}[\mathsf{e}_p] \supseteq \mathsf{R}[u]; e$ | for $(u, \mathsf{pcall}\ p \parallel q, \_) \in E$ |
| [callp2] | $\mathsf{R}[\mathsf{e}_q] \supseteq \mathsf{R}[u]; e$ | for $(u, \mathsf{pcall}\ p \parallel q, \_) \in E$ |

The [call]-constraint accounts for the paths through a parallel call, that are all interleavings of same-level paths through the two procedures called in parallel. For the abstract interpretation of this constraint, we lift the operator $\otimes^\# : (L \overset{\mathsf{mon}}{\to} L) \times (L \overset{\mathsf{mon}}{\to} L) \to (L \overset{\mathsf{mon}}{\to} L)$ defined by $f \otimes^\# g = f \circ g \sqcup g \circ f$ to sets and use it as precise [13] abstract interleaving operator. The redefined $\mathsf{PI}$ constraint system will be described after the backward analysis.

*Backward Analysis* For backward analysis, the concept of directly leaving paths has to be generalized: In the case without parallel calls, a directly leaving path is a path from some reachable stack. It is complementary to the possible interference, i.e. the transitions on leaving paths are either interfering or directly leaving transitions. In the case of parallel calls, interference is not only caused by parallel threads, but also by procedures called in parallel. Hence it is not sufficient to just distinguish the thread that reached the node from the other threads, but we have to look inside this thread and distinguish between the procedure reaching the node and the procedures executed in parallel.

For instance, consider the tree $s = v(\mathsf{e}_{p_1}, v'(u^\bullet, \mathsf{e}_{p_2}))$ that is visualized in Fig. 2 (the $^\circ$-annotations at the unmarked nodes are omitted for better readability). This tree may have been created by a directly reaching path to $u$. The nodes $\mathsf{e}_{p_1}$ and $\mathsf{e}_{p_2}$ may execute interfering transitions. The other transitions, that are exactly the directly leaving ones, may either be executed from the node $u^\bullet$, or from the nodes $v'$ and $v$, if $p_1$ and $p_2$ have terminated. To describe the directly leaving transitions separately, we define the function $\mathsf{above}$, that transforms a tree with a marked node $u^\bullet$ by pruning all nodes that are not predecessors of $u^\bullet$ and adding dummy nodes to make the tree binary again. For a subtree that may potentially terminate, i.e. be transformed to a single return node by some transition sequence, a dummy return node $u_r$ is added. If the pruned subtree cannot terminate, a dummy non-return node $u_n$ is added. Both $u_r$ and $u_n$ have no outgoing edges. Assuming, for in-

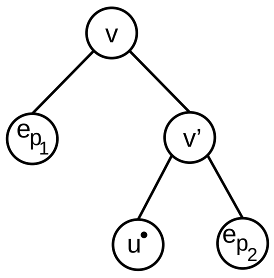

**Fig. 2:** Sample tree with marked node u.

stance, that procedure $p_1$ cannot terminate and $p_2$ can terminate, we would have $\mathsf{above}(s) = v(u_n, v'(u, u_r))$. For technical reasons, $\mathsf{above}$ deletes the marker.

With the help of the $\mathsf{above}$-function, we define the directly leaving paths by

$$\mathsf{L^{op}}[u] := \{w \mid \exists r, c : \{\mathsf{e}^\bullet_{\mathsf{main}}\} \xrightarrow{*}_{\mathsf{mp}} \{r[u^\bullet]\} \uplus c \wedge \{\mathsf{above}(r[u^\bullet])\} \xrightarrow{w} \_\}$$

and show similar to Theorem 4:

**Theorem 7.** $\mathsf{MOP^B}[u] = \alpha^{\mathsf{B}}(\mathsf{L^{op}}[u]) \sqcup \alpha^{\mathsf{PI}}(\mathsf{PI^{op}}[u])$.

The proof formalizes the ideas discussed above. Due to lack of space it is deferred to the appendix of [7].

As in the case without parallel calls, we can characterize the directly leaving paths by a complex constraint system whose abstract version is less complex. So we again perform part of the abstraction on the path level. As in the case without parallel calls, we define the $\overset{\cdot}{\Longrightarrow}$ transition relation, to describe those paths that execute transitions of spawned threads only immediately after the corresponding spawn transition. Transitions executed in parallel due to parallel calls, however, may be executed in any order. Formally, the definition of $\overset{\cdot}{\Longrightarrow}$ looks the same as without parallel calls (we just use trees instead of lists). The definition of $\mathsf{L^{op}_{\subseteq}}$ changes to: $\mathsf{L^{op}_{\subseteq}}[u] := \{w \mid \exists r, c : \{\mathsf{e}^\bullet_{\mathsf{main}}\} \xrightarrow{*}_{\mathsf{mp}} \{r[u^\bullet]\} \uplus c \wedge \{\mathsf{above}(r[u^\bullet])\} \overset{\cdot}{\underset{}{\xRightarrow{w}}} \_\}$. The proof of $\alpha^{\mathsf{B}}(\mathsf{L^{op}}[u]) = \alpha^{\mathsf{B}}(\mathsf{L^{op}_{\subseteq}}[u])$ (Lemma 5) does not change significantly.

However, we do not know any simple constraint system that characterizes $\mathsf{L^{op}_{\subseteq}}[u]$. The reason is that there must be constraints that relate the leaving paths before a parallel call to the paths into or through the called procedures. We cannot use sequential composition here, because $\mathsf{L^{op}_{\subseteq}}$ contains interleavings of procedures called in parallel. But we cannot use interleaving either, because transitions of one parallel procedure might get interleaved arbitrarily with transitions of a thread spawned by the other parallel procedure, which is prohibited by $\overset{\cdot}{\Longrightarrow}$. While it is possible to avoid this problem by working with a more complex definition of $\overset{\cdot}{\Longrightarrow}$, there is a simpler way out. We observe that we need not characterize $\mathsf{L^{op}_{\subseteq}}[u]$ exactly, but only some set $\mathsf{lfp}(\mathsf{L}_{\subseteq})[u]$ *between* $\mathsf{L^{op}_{\subseteq}}[u]$ and $\mathsf{L^{op}}[u]$, i.e. $\mathsf{L^{op}}[u] \supseteq \mathsf{lfp}(\mathsf{L}_{\subseteq})[u] \supseteq \mathsf{L^{op}_{\subseteq}}[u]$. From these inclusions, it follows by monotonicity of the abstraction function $\alpha^{\mathsf{B}}$, that $\alpha^{\mathsf{B}}(\mathsf{L^{op}}[u]) \sqsupseteq \alpha^{\mathsf{B}}(\mathsf{lfp}(\mathsf{L}_{\subseteq})[u]) \sqsupseteq \alpha^{\mathsf{B}}(\mathsf{L^{op}_{\subseteq}}[u]) = \alpha^{\mathsf{B}}(\mathsf{L^{op}}[u])$, and thus $\alpha^{\mathsf{B}}(\mathsf{lfp}(\mathsf{L}_{\subseteq})[u]) = \alpha^{\mathsf{B}}(\mathsf{L^{op}}[u])$.

In order to obtain appropriate constraint systems, we replace in the constraint systems $\mathsf{L}_{\subseteq}$, $\mathsf{LS}$ and $\mathsf{S_B}$ the constraints related to call edges as follows:

[LS.init2] dropped
[LS.call1] $\mathsf{LS}[u] \supseteq e; (\mathsf{LS}[\mathsf{e}_{p_1}] \otimes \mathsf{LS}[\mathsf{e}_{p_2}])$ for $(u, \mathsf{pcall}\ p_1 \parallel p_2, v) \in E$
[LS.call2] $\mathsf{LS}[u] \supseteq e; (\mathsf{S_B}[\mathsf{e}_{p_1}] \otimes \mathsf{S_B}[\mathsf{e}_{p_2}]); \mathsf{ret}; \mathsf{LS}[v]$ for $(u, \mathsf{pcall}\ p_1 \parallel p_2, v) \in E$

[SB.call] $\mathsf{S_B}[u] \supseteq e; (\mathsf{S_B}[\mathsf{e}_{p_1}] \otimes \mathsf{S_B}[\mathsf{e}_{p_2}]); \mathsf{ret}; \mathsf{S_B}[v]$ for $(u, \mathsf{pcall}\ p_1 \parallel p_2, v) \in E$
[L.ret] $\mathsf{L}_{\subseteq}[\mathsf{r}_{p_i}] \supseteq \mathsf{ret}; \mathsf{L}_{\subseteq}[v]$ for $(\_, \mathsf{pcall}\ p_1 \parallel p_2, v) \in E$
$p_1, p_2$ can terminate, $i = 1, 2$

We have to drop the constraint [LS.init2], because in our generalization to parallel calls, the procedure at the root of the tree can never return, while in the model without parallel calls, the procedure at the bottom of the stack may return. The constraints [LS.call1], [LS.call2] and [SB.call] account for any interleaving between the paths into resp. through two procedures called in parallel, even when thereby breaking the atomicity of the transitions of some spawned thread. With the ideas above, it is straightforward to prove the required inclusions $\mathsf{L}^{\mathsf{op}}[u] \supseteq \mathsf{lfp}(\mathsf{L}_{\subseteq}[u]) \supseteq \mathsf{L}^{\mathsf{op}}_{\subseteq}[u]$. As these constraint systems do not contain any new operators, abstract versions can be obtained as usual.

*Possible Interference.* It remains to modify the constraint system for $\mathsf{PI}$. This is done by replacing the constraints for call edges with the following ones:

| | | |
|---|---|---|
| [SP.call] | $\mathsf{SP}[v] \supseteq \mathsf{SP}[\mathsf{r}_{p_1}] \cup \mathsf{SP}[\mathsf{r}_{p_2}] \cup \mathsf{SP}[u]$ | $(u, \mathsf{pcall}\ p_1 \parallel p_2, v) \in E$ if $p_1, p_2$ can terminate |
| [PI.trans1] | $\mathsf{PI}[\mathsf{e}_{p_i}] \supseteq \mathsf{PI}[u]$ | $(u, \mathsf{pcall}\ p_1 \parallel p_2, v) \in E, i = 1, 2$ |
| [PI.callmi] | $\mathsf{PI}[\mathsf{e}_{p_i}] \supseteq \alpha^{\mathsf{E}}(\mathsf{LS}^{\mathsf{op}}[\mathsf{e}_{p_{3-i}}])$ | $(u, \mathsf{pcall}\ p_1 \parallel p_2, v) \in E, i = 1, 2$ |

The [SP.call]-constraint now accounts for the interference generated by both procedures called in parallel and [PI.trans1] forwards the interference to both procedures. The [PI.callmi]-constraint has no correspondent in the original $\mathsf{PI}$-system. It accounts for the fact that a procedure $p_{3-i}$ generates interference for $p_i$ in a parallel call $\mathsf{pcall}\ p_1 \parallel p_2$. Again, the necessary soundness and precision proofs as well as abstraction are straightforward.

## 8 Conclusion

From the results in this paper, we can construct an algorithm for precise kill/gen-analysis of interprocedural flowgraphs with thread creation and parallel procedure calls:

1. Generate the abstract versions of the constraint systems $\mathsf{R}$,$\mathsf{PI}$, $\mathsf{L}_{\subseteq}$ and all dependent constraint systems from the flowgraph.
2. Compute their least solutions.
3. Return the approximation of the $\mathsf{MOP}^{\mathsf{F}}$- and $\mathsf{MOP}^{\mathsf{B}}$-solution respectively, as indicated by Theorem 2, Theorem 4 and Lemma 5.

Let us briefly estimate the complexity of this algorithm: We generate $O(|E| + |P|)$ constraints over $O(|N|)$ variables. If the height of $(L, \sqsubseteq)$ is bounded by $h(L)$ and a lattice operation (join, compare, assign) needs time $O(op)$, the algorithm needs time $O((|E| * h(L) + |N|) * op)$ if a worklist algorithm [11] is used in Step 2. A prototype implementation of our algorithm for forward problems has been constructed in [6]. The algorithm may be extended to treat local variables using well-known techniques; see e.g. [13].

Compared to related work, our contributions are the following: Generalizing [13], we treat thread creation in addition to parallel procedure calls and handle backward analysis completely. Compared to [1], our analysis computes information for all program points in linear time, while the automata based algorithm of [1] needs at least linear time *per* program point. Moreover, representing powersets by bitvectors as usual, we can exploit efficient bitvector operations, while the algorithm of [1] needs to be iterated for each bit.

Like other related work [5, 13, 3, 4, 9, 1], we do not handle synchronization such as message passing. In presence of such synchronization, we still get a correct (but weak) approximation. There are limiting undecidability results [12], but further research has to be done to increase approximation quality in presence of synchronization. Also extensions to more complex domains, e.g. analysis of transitive dependences as studied in [9] for parallel calls, have to be investigated.

*Acknowledgment.* We thank Helmut Seidl and Bernhard Steffen for interesting discussions on analysis of parallel programs and the anonymous reviewers for their helpful comments.

## References

1. A. Bouajjani, M. Müller-Olm, and T. Touili. Regular symbolic analysis of dynamic networks of pushdown systems. In *Proc. of CONCUR'05*. Springer, 2005.
2. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'77*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York.
3. J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In *Proc. of FoSSaCS'99*, pages 14–30. Springer, 1999.
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. J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *TOPLAS*, 18(3):268–299, May 1996.
6. P. Lammich. Fixpunkt-basierte optimale Analyse von Programmen mit Thread-Erzeugung. Master's thesis, University of Dortmund, May 2006.
7. P. Lammich and M. Müller-Olm. Precise fixpoint-based analysis of programs with thread-creation. Available from `http://cs.uni-muenster.de/u/mmo/pubs/`. Version with appendix.
8. P. Lammich and M. Müller-Olm. Precise fixed point based analysis of programs with thread-creation. In *Proc. of MEMICS 2006*, pages 91–98. Faculty of Information Technology, Brno University of Technology, 2006.
9. M. Müller-Olm. Precise interprocedural dependence analysis of parallel programs. *Theor. Comput. Sci.*, 311(1-3):325–388, 2004.
10. M. Müller-Olm and H. Seidl. On optimal slicing of parallel programs. In *Proc. of STOC'01*, pages 647–656, New York, NY, USA, 2001. ACM Press.
11. F. Nielson, H. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 1999.
12. G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *TOPLAS*, 22(2):416–430, 2000.
13. H. Seidl and B. Steffen. Constraint-Based Inter-Procedural Analysis of Parallel Programs. *Nordic Journal of Computing (NJC)*, 7(4):375–400, 2000.