# On Optimal Slicing of Parallel Programs

Markus Müller-Olm
Universität Dortmund, FB Informatik, LS 5
44221 Dortmund, Germany
mmo@ls5.cs.uni-dortmund.de

Helmut Seidl *
Universität Trier, FB 4-Informatik
54286 Trier, Germany
seidl@uni-trier.de

## ABSTRACT

Optimal program slicing determines for a statement $S$ in a program $\pi$ whether or not $S$ affects a specified set of statements, given that all conditionals in $\pi$ are interpreted as non-deterministic choices.

Only recently, it has been shown that reachability of program points and hence also optimal slicing is undecidable for multi-threaded programs with (parameterless) procedures and synchronization [23]. Here, we sharpen this result by proving that slicing remains undecidable if synchronization is abandoned—although reachability becomes polynomial. Moreover, we show for multi-threaded programs without synchronization, that slicing stays PSPACE-hard when procedure calls are forbidden, and becomes NP-hard for loop-free programs. Since the latter two problems can be solved in PSPACE and NP, respectively, even in presence of synchronization, our new lower bounds are tight.

Finally, we show that the above decidability and lower bound properties equally apply to other simple program analysis problems like copy constant propagation and true liveness of variables. This should be contrasted to the problems of strong copy constant propagation and (ordinary) liveness of variables for which polynomial algorithms have been designed [15, 14, 24].

## Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*concurrent programming structures*; F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Program analysis*; F.4.3 [**Mathematical Logic and Formal Languages**]: Formal Languages—*classes defined by grammars or automata, decision problems*

*The second author was supported by the RTD project IST-1999-20527 "DAEDALUS" of the European FP5 programme.


*STOC'01,* July 6-8, 2001, Hersonissos, Crete, Greece.


## Keywords

Slicing, parallel programs, interprocedural analysis, undecidability, complexity

## 1. INTRODUCTION

Static program slicing [27] is an established program reduction technique that has applications in program understanding, debugging, and testing [26]. More recently, it has also been proposed as a technique for ameliorating the state-explosion problem when formally verifying software or hardware [13, 10, 4, 18]. The goal of program slicing is to identify and remove parts of the program that cannot (potentially) influence certain value(s) at certain program point(s) of interest. The latter is called the *slicing criterion*.

There is a vast amount of literature on slicing sequential languages (see the references in Tip's survey [26]). A crucial idea found in many variations is to perform slicing by means of a backwards reachability analysis on a graph modeling basic dependences between instructions. This approach has been pioneered by Ottenstein and Ottenstein [21] who proposed to use a structure called PDG (Program Dependence Graph). A PDG captures two kinds of dependences, *data dependences* and *control dependences*. Intuitively, a statement $S$ is data dependent on another statement $T$ if $T$ updates a variable that can be referenced by $S$. For example, if $S$ is $x := e$ and $T$ is $y := f$, then $S$ is data dependent on $T$ if $y$ appears in $e$ and there is a path from $T$ to $S$ in the program on which no statement updates $y$. *Control dependence* captures which guards (of branching statements or loops) may determine whether a statement is executed or not. Its formal definition can be found, e.g., in [26].

The first who considered static slicing of *concurrent languages* was J. Cheng [3]. In recent years the interest in this problem has increased due to the proliferation of concurrent languages. There has been work in connection with slicing JAVA-like languages [10, 28], VHDL [13, 4], and PROMELA [18], the input language of the SPIN model checker. All these articles have in common that slicing is again approached as a backwards reachability problem but on some extended form of PDG (called Process Dependence Net [3], Multithreaded Dependence Graph [28], etc.). These structures model further dependences besides data and control dependences that may arise in concurrent programs of the considered kind. One such dependence is *interference dependence* [17, 10]. A statement $S$ is interference dependent on a statement $T$ in another thread if the two threads may run in parallel and there is a variable updated by $T$ and referenced by $S$. This captures the situation that in a parallel execution of the two threads, $S$ may be executed after $T$ in such a way that the shared variable is not overwritten in between. Interference dependence may be interpreted as a kind of data dependence arising from interleaved execution. Other kinds of dependences represent the data flow induced by message passing and the control flow induced by synchronization operations.

A program slicing algorithm must be *sound*: it must not slice away parts of the program that affect the given slicing criterion. Ideally, a slicer should remove as much of the program as possible without sacrificing soundness. Weiser [27] showed already that the problem of determining whether or not a slice is statement-minimal is undecidable [26, p. 7]. The problem is that it is undecidable whether a condition found in the program may be true (or false) on some execution path. Dataflow analysis in general suffers from this problem and the common remedy is to ignore conditions altogether when defining feasible paths. In other words, conditional branching is interpreted as non-deterministic branching, a point of view adopted in this paper. We call a slicer *optimal* if it determines a statement-minimal slice under this abstraction.

In the *sequential*, intraprocedural case (i.e. in single procedures), PDG-based slicing is efficient and optimal. Optimality can also be achieved in the sequential, interprocedural case by solving a context-free reachability problem on the System Dependency Graph (SDG) of the program in question [11]. This analysis can be done in polynomial time [26]. For *concurrent* languages with procedures and synchronization primitives even reachability is undecidable by a recent result of Ramalingam [23]. This implies that also optimal slicing cannot be decidable. In this paper, we consider optimal slicing for concurrent languages but drop the facility of synchronization. As a consequence, reachability as well as reverse reachability become decidable—even polynomial [5, 6, 24]. Our new result is that optimal slicing remains undecidable. We refine this new undecidability result by proving optimal slicing to be PSPACE-hard in case that there are no procedure calls, and still NP-hard if also loops are abandoned. The latter two lower complexity bounds are optimal, as they match the corresponding upper bounds.

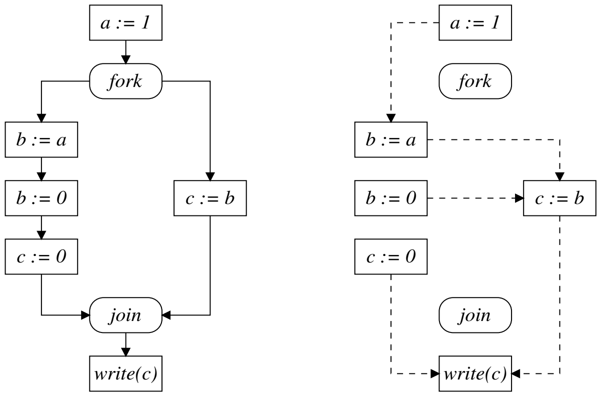
We conclude that all efficient slicing algorithms for concurrent languages are doomed to be sub-optimal (unless P=PSPACE). Our results are shown under very weak assumptions on the concurrent language. Intuitively, they exploit a weakness of interference dependence only. As no synchronization properties are exploited, our results point to a more fundamental limitation for slicing concurrent languages than Ramalingam's and hence are applicable to a much wider range of concurrency scenarios.

Finally, we consider related program analysis problems, copy constant propagation and true liveness of variables, and exhibit similar undecidability and complexity results as for slicing thereby strengthening recent results [20]. In a certain sense, this comes as a surprise, as only slightly simpler analysis questions, namely, *strong* copy constant propagation and (ordinary) liveness of variables can be optimally solved in polynomial time [15, 14, 24].

## 2. A MOTIVATING EXAMPLE

Before we turn to the technical results, let us discuss a small example that illustrates that backwards reachability in the dependence graph can give sub-optimal results when slicing parallel programs. Consider the program

$$a := 1; [(b := 1; b := 0; c := 0) \parallel c := b] \; ; \mathbf{write}(c)\,.$$




(a) CFG-like representation (b) Data and interference dependences

**Figure 1: An illustrative example.**

In Fig. 1 (a) a control flow graph-like representation of the program is shown and in (b) the data and interference dependences. We are interested in slicing w.r.t. variable $c$ at the write instruction. (We always use write instructions in this paper to mark the slicing criterion clearly and conveniently; this is the only purpose of write instructions here). Clearly, the instruction $a := 1$ is backwards reachable in the dependency graph. But there is no execution of the program that realizes all dependences in this path and therefore an optimal slicer must remove $a := 1$. In order to see this consider that in an execution $b := 0$ must be executed either before or after $c := b$ in the parallel thread. If it is executed before $c := b$ then it kills the propagation from $b := a$ to $c := b$. If it is executed after $c := b$ then the subsequent statement $c := 0$ kills the propagation from $c := b$ to $\mathbf{write}(c)$. Our undecidability and hardness results exploit that propagation can be prohibited in this way by means of re-initializations. Krinke [17] also mentions that 'interference dependence is not transitive' and gives an example that is, however, of a less subtle nature than our example. He, too, does not consider synchronization operations and presents an optimal algorithm for the intraprocedural parallel case. His algorithm is worst-case exponential but he gives no hardness proof. Our PSPACE-hardness result explains—by all what we believe about PSPACE-hardness—why he could not find a polynomial algorithm.

## 3. PARALLEL PROGRAMS

We consider a prototypic language with shared memory, atomic assignments and fork/join parallelism. Only assignments of a very simple form are needed: $x := k$ where $k$ is either a constant or a variable.

A *procedural parallel program* comprises a finite set $\mathsf{Proc}$ of *procedure names* containing a distinguished name *Main*. Each procedure name $P$ is associated with a statement $\pi_P$, the corresponding *procedure body*, constructed according to the following grammar, in which $Q$ ranges over $\mathsf{Proc}\setminus\{Main\}$

and $x$ over some given finite set of variables:

$$\begin{array}{lll} e & ::= & c \mid x \\ \pi & ::= & x := e \mid \textbf{write}(e) \mid \textbf{skip} \mid Q \mid \pi_1 \; ; \pi_2 \mid \\ & & \pi_1 \parallel \pi_2 \mid \pi_1 \sqcap \pi_2 \mid \textbf{loop} \; \pi \; \textbf{end}\,. \end{array}$$

We use the syntax **procedure** $P; \pi_P$ **end** to indicate the association of procedure bodies to procedure names. Note that procedures do not have parameters.

The specific nature of constants and the domain in which they are interpreted is immaterial; we only need that 0 and 1 are two constants representing different values, which—by abuse of notation—are denoted by 0 and 1 too. In other words we only need Boolean variables. The atomic statements of the language are assignment statements $x := e$ that assign the current value of $e$ to variable $x$, 'do-nothing' statements **skip**, and write statements. Write statements signify the slicing criterion. A statement of the form $Q$ denotes a call of procedure $Q$. The operator ; denotes sequential composition and $\parallel$ parallel composition. The operator $\sqcap$ represents non-deterministic branching and **loop** $\pi$ **end** stands for a loop that iterates $\pi$ an indefinite number of times. Such construct are chosen in accordance with the common abstraction from conditions mentioned in the introduction. We apply the non-deterministic choice operator also to finite sets of statements; $\sqcap\{\pi_1, \ldots, \pi_n\}$ denotes $\pi_1 \sqcap \cdots \sqcap \pi_n$. The ambiguity inherent in this notation is harmless because $\sqcap$ is commutative, associative, and idempotent semantically.

Note that there are no synchronization operations in the language. The synchronization of start and termination inherent in fork- and join-parallelism is also not essential for our results; see Section 7.

Parallelism is understood in an interleaving fashion; assignments and write statements are assumed to be atomic. A *run* of a program is a maximal sequence of atomic statements that may be executed in this order in an execution of the program. The program $(x := 1 \; ; \; x := y) \parallel y := x$, for example, has the three runs $\langle x := 1, x := y, y := x\rangle$, $\langle x := 1, y := x, x := y\rangle$, and $\langle y := x, x := 1, x := y\rangle$. We denote the set of runs of program $\pi$ by $\mathsf{Runs}(\pi)$.

## 4. INTERPROCEDURAL SLICING

In the remainder of this paper we adopt the following definition of the (optimal) slicing problem as a decision problem. An instance comprises a (non-deterministic, parallel) program $\pi$, a slicing criterion $C$ (given by the write-instructions in the program) and a statement $S$ in $\pi$. The problem is to decide whether $S$ belongs to the optimal slice of $\pi$ with respect to $C$. The slicing problem is parameterized by the class of programs considered.

THEOREM 1. *Parallel interprocedural slicing is undecidable.*

It is well-known that the termination problem for two-counter machines is undecidable [19]. In the remainder of this section, we reduce this problem to an interprocedural slicing problem thereby proving Theorem 1.

### 4.1 Two-Counter Machines

A two-counter machine has two counter variables $c_0$ and $c_1$ that can be incremented, decremented, and tested against zero. It is common to use a combined decrement- and test-instruction in order to avoid complications with decrementing a zero counter. The basic idea of our reduction is to represent the values of the counters by the stack height of two threads of procedures running in parallel. Incrementing a counter is represented by calling another procedure in the corresponding thread, decrementing by returning from the current procedure, and the test against zero by using different procedures at the first and the other stack levels that represent the possible moves for zero and non-zero counters, respectively. It simplifies the argumentation if computation steps involving the two counters alternate. This can always be enforced by adding skip-instructions that do nothing except of transferring control.

Formally, we use the following model. A *two-counter machine* $M$ comprises a finite set of (control) states $S$. $S$ is partitioned into two sets $P = \{p_1, \ldots, p_n\}$ and $Q = \{q_1, \ldots, q_m\}$; moves involving counter $c_0$ start from $P$ and moves involving counter $c_1$ from $Q$. Execution commences at a distinguished *start state* which, w.l.o.G., is $p_1$. There is also a distinguished *final state*, w.l.o.G. $p_n$, at which execution terminates. Each state $s \in S$ except of the final state $p_n$ is associated with an instruction $I(s)$ taken from the following selection:

- $c_i := c_i + 1;$ **goto** $s'$ (increment),
- **if** $c_i = 0$ **then goto** $s'$ **else** $c_i := c_i - 1;$ **goto** $s''$ (test-decrement), or
- **goto** $s'$ (skip),

where $i = 0$ and $s', s'' \in Q$ if $s \in P$, and $i = 1$ and $s', s'' \in P$ if $s \in Q$. Note that this condition captures that moves alternate.

Execution of a two-counter machine $M$ is represented by a transition relation $\rightarrow_M$ on configurations $\langle s, x_0, x_1\rangle$ that consist of a current state $s \in S$ and current values $x_0 \geq 0$ and $x_1 \geq 0$ of the counters. Configurations with $s = p_n$ are called *final configurations*. We have $\langle s, x_0, x_1\rangle \rightarrow_M \langle s', x_0', x_1'\rangle$ if and only if one of the following conditions is valid for $i = 0, 1$:

- $I(s) = c_i := c_i + 1;$ **goto** $s'$, $x_i' = x_i + 1$, and $x_{1-i}' = x_{1-i}$.
- $I(s)$ = **if** $c_i = 0$ **then goto** $s'$ **else** $c_i := c_i - 1;$ **goto** $s''$, $x_i = 0$, $x_i' = x_i$, and $x_{1-i}' = x_{1-i}$.
- $I(s)$ = **if** $c_i = 0$ **then goto** $s''$ **else** $c_i := c_i - 1;$ **goto** $s'$, $x_i \neq 0$, $x_i' = x_i - 1$, and $x_{1-i}' = x_{1-i}$.
- $I(s)$ = **goto** $s'$, $x_i' = x_i$, and $x_{1-i}' = x_{1-i}$.

Thus, each non-final configuration has a unique successor configuration. We denote the reflexive transitive closure of $\rightarrow_M$ by $\rightarrow_M^*$ and omit the subscript $M$ if it is clear from context.

Execution of a two-counter machine commences at the start state with the counters initialized by zero, i.e. in the configuration $\langle p_1, 0, 0\rangle$. The two-counter machine *terminates* if it ever reaches the final state, i.e. if $\langle p_1, 0, 0\rangle \rightarrow^* \langle p_n, x_0, x_1\rangle$ for some $x_0, x_1$. As far as the halting behavior is concerned we can assume without loss of generality that both counters are zero upon termination. This can be ensured by adding two loops at the final state that iteratively

**procedure** $P_0$;
**loop**
  $\sqcap\{p := x_k$ ; $\mathsf{KillAll}_P$ ; $y_l := p$ ; $P_{\neq 0}$ |
    $I(p_k) = c_0 := c_0 + 1;$ **goto** $q_l\}$ $\sqcap$
  $\sqcap\{p := x_k$ ; $\mathsf{KillAll}_P$ ; $y_l := p$ |
    $I(p_k) =$ **if** $c_0 = 0$ **then goto** $q_l$ **else** $\ldots\}$ $\sqcap$
  $\sqcap\{p := x_k$ ; $\mathsf{KillAll}_P$ ; $y_l := p \mid I(p_k) =$ **goto** $q_l\}$
**end**
**end**

**procedure** $P_{\neq 0}$;
**loop**
  $\sqcap\{p := x_k$ ; $\mathsf{KillAll}_P$ ; $y_l := p$ ; $P_{\neq 0}$ |
    $I(p_k) = c_0 := c_0 + 1;$ **goto** $q_l\}$ $\sqcap$
  $\sqcap\{p := x_k$ ; $\mathsf{KillAll}_P$ ; $y_l := p \mid I(p_k) =$ **goto** $q_l\}$
**end** ;
$\sqcap\{p := x_k$ ; $\mathsf{KillAll}_P$ ; $y_l := p$ |
    $I(p_k) =$ **if** $c_0 = 0$ **then** $\ldots$ **else** $\ldots$ **goto** $q_l\}$
**end**

**procedure** $\mathsf{KillAll}_P$;
$y_1 := 0$ ; $\ldots$ ; $y_m := 0$ ; $q := 0$ ; $x_1 := 0$ ; $\ldots$ ; $x_n := 0$
**end**

**Figure 2: Definition of $P_0$ and $P_{\neq 0}$.**

**procedure** $Q_0$;
**loop**
  $\sqcap\{q := y_k$ ; $\mathsf{KillAll}_Q$ ; $x_l := q$ ; $Q_{\neq 0}$ |
    $I(q_k) = c_1 := c_1 + 1;$ **goto** $p_l\}$ $\sqcap$
  $\sqcap\{q := y_k$ ; $\mathsf{KillAll}_Q$ ; $x_l := q$ |
    $I(q_k) =$ **if** $c_1 = 0$ **then goto** $p_l$ **else** $\ldots\}$ $\sqcap$
  $\sqcap\{q := y_k$ ; $\mathsf{KillAll}_Q$ ; $x_l := q \mid I(q_k) =$ **goto** $p_l\}$
**end**
**end**

**procedure** $Q_{\neq 0}$;
**loop**
  $\sqcap\{q := y_k$ ; $\mathsf{KillAll}_Q$ ; $x_l := q$ ; $Q_{\neq 0}$ |
    $I(q_k) = c_1 := c_1 + 1;$ **goto** $p_l\}$ $\sqcap$
  $\sqcap\{q := y_k$ ; $\mathsf{KillAll}_Q$ ; $x_l := q \mid I(q_k) =$ **goto** $p_l\}$
**end** ;
$\sqcap\{q := y_k$ ; $\mathsf{KillAll}_Q$ ; $x_l := q$ |
    $I(q_k) =$ **if** $c_1 = 0$ **then** $\ldots$ **else** $\ldots$ **goto** $p_l\}$
**end**

**procedure** $\mathsf{KillAll}_Q$;
$x_1 := 0$ ; $\ldots$ ; $x_n := 0$ ; $p := 0$ ; $y_1 := 0$ ; $\ldots$ ; $y_m := 0$
**end**

**Figure 3: Definition of $Q_0$ and $Q_{\neq 0}$.**

decrement the counters until they become zero. Obviously, this modification preserves the termination behavior of the two-counter machine. Note that for the modified machine the conditions "$\langle p_1, 0, 0\rangle \to^* \langle p_n, x_0, x_1\rangle$ for some $x_0, x_1$" and "$\langle p_1, 0, 0\rangle \to^* \langle p_n, 0, 0\rangle$" are equivalent. We assume in the following that such loops have been added to the given machine.

## 4.2 Constructing a Program

From a two-counter machine as above we construct a parallel program, $\pi_M$. For each state $p_k \in P$ the program uses a variable $x_k$ and for each state $q_l \in Q$ a variable $y_l$. Intuitively, $x_k$ holds the value 1 in an execution of the program iff this execution corresponds to a run of the two-counter machine reaching state $p_k$, and similarly for the $y_l$.

The main procedure of $\pi_M$ reads as follows:

| | |
|---|---|
| **procedure** *Main*; | **procedure** *Init*; |
| $x_1 := 1$ ; *Init* ; | $x_2 := 0$ ; $\ldots$ ; $x_n := 0$ ; |
| $(P_0 \parallel Q_0)$ ; **write**$(x_n)$ | $y_1 := 0$ ; $\ldots$ ; $y_m := 0$ |
| **end** | **end** |

We will consider slicing with respect to variable $x_n$ at the write-instruction (slicing criterion). The construction is done such that the initialization $x_1 := 1$ belongs to the optimal slice if and only if $M$ terminates. This shows Theorem 1. The goal of the construction can also be reformulated as follows because the initialization $x_1 := 1$ is the only occurrence of the constant 1 in the program and all other assignment statement only copy values or initialize variables by 0.

$$\begin{array}{l} M \text{ terminates if and only if} \\ x_n \text{ may hold 1 at the write-statement.} \end{array} \tag{1}$$

The initialization of all variables except $x_1$ by 0 reflects that $p_1$ is the initial state. For each of the two counters the program uses two procedures, $P_0$ and $P_{\neq 0}$ for counter $c_0$ and $Q_0$ and $Q_{\neq 0}$ for counter $c_1$. Their definition can be found in Fig. 2 and 3. We describe $P_0$ and $P_{\neq 0}$ in detail in the following, $Q_0$ and $Q_{\neq 0}$ are completely analogous.

Intuitively, $P_0$ and $P_{\neq 0}$ mirror transitions of $M$ induced by counter $c_0$ being $=0$ and $\neq 0$, respectively, hence their name. Each procedure non-deterministically guesses the next transition. Such a transition involves two things: first, a state change and, secondly, an effect on the counter value. The state change from some $p_k$ to some $q_l$ is represented by copying $x_k$ to $y_l$ via an auxiliary variable $p$ and re-initializing $x_k$ by zero as part of $\mathsf{KillAll}_P$. The effect on the counter value is represented by how we proceed:

- For transitions that do not change the counter we jump back to the beginning of the procedure such that other transitions with the same counter value can be simulated subsequently. This applies to skip-transitions and test-decrement transitions for a zero counter, i.e. test-decrement transitions simulated in $P_0$.
- For incrementing transitions we call another instance of $P_{\neq 0}$ that simulates the transitions induced by the incremented counter. A return from this new instance of $P_{\neq 0}$ means that the counter is decremented, i.e. has the old value. We therefore jump back to the beginning of the procedure after the return from $P_{\neq 0}$.
- For test-decrement transitions simulated in $P_{\neq 0}$, we leave the current procedure.

This behavior is described in a structured way by means of loops and sequential and non-deterministic composition and is consistent with the representation of the counter value by the number of instances of $P_{\neq 0}$ on the stack.

The problem with achieving (1) is that a procedure may try to 'cheat': it may execute the code representing a transition from $p_i$ to $q_j$ although $x_i$ does not hold the value 1. If this is a decrementing or incrementing transition the coincidence between counter values and stack heights may then be destroyed and the value 1 may subsequently be propagated erroneously. Such cheating may thus invalidate the 'if' direction.

This problem is solved as follows. We ensure by appropriate re-initialization that all variables are set to 0 if a procedure tries to cheat. Thus, such executions cannot contribute to the propagation of the value 1. But re-initializing a set of variables safely is not trivial in a concurrent environment. We have only atomic assignments to single variables available; a variable just set to 0 may well be set to another value by instructions executed by instances of the procedures $Q_0$ and $Q_{\neq 0}$ running in parallel while we are initializing the other variables. Here our assumption that moves involving the counters alternate comes into play. Due to this assumption all copying assignments in $Q_0$ and $Q_{\neq 0}$ are of the form $q := y_i$ or $x_j := q$ ($q$ is the analog of the auxiliary variable $p$). Thus, we can safely assign 0 to the $y_i$ in $P_0$ and $P_{\neq 0}$ as they are not the target of a copy instruction in $Q_0$ or $Q_{\neq 0}$. After we have done so, we can safely assign 0 to $q$; a copy instruction $q := y_i$ executed by the parallel thread cannot destroy the value 0 as all $y_i$ contain 0 already. After that we can safely assign 0 to the $x_i$ by a similar argument. This explains the definition of $\mathsf{KillAll}_P$.

## 4.3 Correctness of the Reduction

From the intuition underlying the definition of $\pi_M$, the 'only if' direction of (1) is rather obvious: If $M$ terminates, i.e., if it has transitions leading from $\langle p_1, 0, 0\rangle$ to $\langle p_n, 0, 0\rangle$, we can simulate these transitions by a propagating run of $\pi_M$. By explaining the definition of $\mathsf{KillAll}_P$, we justified the 'if' direction as well. A formal proof can be given along the lines of the classic Owicki/Gries method for proving partial correctness of parallel programs [22, 8, 1]. Although this method is usually presented for programs without procedures it is sound also for procedural programs. In the Owicki/Gries method, programs are annotated with assertions that represent properties valid for any execution reaching the program point at which the assertion is written down. This annotation is subject to certain rules that guarantee soundness of the method.

Specifically, we prove that just before the write-instruction in $\pi_M$ the following assertion is valid:

$$x_n = 1 \;\Rightarrow\; \langle p_1, 0, 0\rangle \rightarrow^* \langle p_n, 0, 0\rangle\,.$$

Validity of this assertion implies the 'if' direction of (1). The details of this proof are deferred to Appendix A.

Our proof should be compared to undecidability of reachability in presence of synchronization as proved by Ramalingam [23], and undecidability of LTL model-checking for parallel languages (even without synchronization) as proved by Bouajjani and Habermehl [2]. Both proofs employ two sequential threads running in parallel. Ramalingam uses the two recursion stacks of the threads to simulate context-free grammar derivations of two words whose equality is enforced by the synchronization facilities of the programming language. Bouajjani and Habermehl use the two recursion stacks to simulate two counters (as we do) whose joint operation then is synchronized through the LTL formula. Thus, both proofs rely on some kind of "external synchronization" of the two threads – which is not available in our scenario. Instead, our undecidability proof works with "internal synchronization" which is provided implicitly by killing of the circulating value 1 as soon as one thread deviates from the intended synchronous behavior.

# 5. INTRAPROCEDURAL SLICING

The undecidability result just presented means that we cannot expect a program slicer for parallel programs to be optimal. We therefore must lower our expectation. In dataflow analysis one often investigates also *intraprocedural* problems. These can be viewed as problems for programs without procedure calls. Here, we find:

THEOREM 2. *Parallel intraprocedural slicing is PSPACE-complete.*

In a fork/join parallel program without procedures, the number of threads potentially running in parallel is bounded by the size of the program. Therefore, every run of the program can be simulated by a Turing machine using just a polynomial amount of space. We conclude that the intraprocedural optimal parallel slicing problem is in PSPACE.

It remains to show that PSPACE is also a lower bound on the complexity of an optimal intraprocedural parallel slicer, i.e. PSPACE-hardness. This is done by a reduction from the REGULAR EXPRESSION INTERSECTION problem. This problem is chosen in favor of the better known intersection problem for finite automata as we are heading for structured programs and not for flow graphs.

An instance of REGULAR EXPRESSION INTERSECTION is given by a sequence $r_1, \ldots, r_n$ of regular expressions over some finite alphabet $A$. The problem is to decide whether $L(r_1) \cap \ldots \cap L(r_n)$ is non-empty.

LEMMA 1. *The* REGULAR EXPRESSION INTERSECTION *problem is PSPACE-complete.* □

In fact, PSPACE-hardness of the REGULAR EXPRESSION INTERSECTION problem follows by a reduction from the acceptance problem for linear space bounded Turing machines along the same lines as in the corresponding proof for finite automata [16]. The problem remains PSPACE-complete if we consider expressions without $\emptyset$.

Suppose now that $A = \{a_1, \ldots, a_k\}$, and we are given $n$ regular expressions $r_1, \ldots, r_n$. In our reduction we construct a parallel program that starts $n+1$ threads $\pi_0, \ldots, \pi_n$ after some initialization of the variables used in the program:

**procedure** *Main*;
$\mathsf{KillXY}_0$ ; ... ; $\mathsf{KillXY}_n$ ; $x_{n,a_1} := 1$ ;
$[\pi_0 \parallel \pi_1 \parallel \cdots \parallel \pi_n]$ ; **write**$(x_{0,a_1})$
**end**

The threads refer to variables $x_{i,a}$ and $y_i$ ($i \in \{0, \ldots, n\}$, $a \in A$). Thread $\pi_0$ is defined as follows.

$\pi_0 =$ **loop**
$\quad\sqcap\{y_0 := x_{n,a}$ ; $\mathsf{KillAll}_0$ ; $x_{0,b} := y_0 \mid a, b \in A\}$
**end**

The statement $\mathsf{KillAll}_0$ that is defined below ensures that all variables except $y_0$ are re-initialized by 0 irrespective of the behavior of the other threads as shown below.

For $i = 1, \ldots, n$, the thread $\pi_i$ is induced by the regular expression $r_i$. It is given by $\pi_i = \pi_i(r_i)$, where $\pi_i(r)$ is defined by induction on $r$ as follows.

$$\begin{aligned}
\pi_i(\varepsilon) &= \textbf{skip} \\
\pi_i(a) &= y_i := x_{i-1,a} \ ; \mathsf{KillAll}_i \ ; x_{i,a} := y_i \\
\pi_i(r_1 \cdot r_2) &= \pi_i(r_1) \ ; \pi_i(r_2) \\
\pi_i(r_1 + r_2) &= \pi_i(r_1) \sqcap \pi_i(r_2) \\
\pi_i(r^*) &= \textbf{loop } \pi_i(r) \textbf{ end}
\end{aligned}$$

The statement $\mathsf{KillAll}_i$ re-initializes all variables except $y_i$. This statement as well as statements $\mathsf{KillX}_j$ and $\mathsf{KillXY}_j$ on which its definition is based are defined as follows.

$$\begin{aligned}
\mathsf{KillX}_j &= x_{j,a_1} := 0; \dots ; x_{j,a_k} := 0 \\
\mathsf{KillXY}_j &= y_j := 0; \mathsf{KillX}_j \\
\mathsf{KillAll}_i &= \mathsf{KillX}_i; \mathsf{KillXY}_{i+1}; \dots ; \mathsf{KillXY}_n; \\
&\phantom{=} \ \mathsf{KillXY}_0; \dots ; \mathsf{KillXY}_{i-1}
\end{aligned}$$

Again it is not obvious that thread $\pi_i$ can safely re-initialize the variables because the other threads may arbitrarily interleave. But by exploiting that only copy instructions of the form $y_j := x_{j-1,a}$ and $x_{j,a} := y_j$ with $j \neq i$ are present in the other threads this can be done by performing the re-initializations in the order specified above.[1] Two crucial properties are exploited for this. First, whenever $a := b$ is a copying assignments in a parallel thread, variable $b$ is re-initialized before $a$. Therefore, execution of $a := b$ after the re-initialization of $b$ just copies the initialization value 0 from $b$ to $a$ but cannot destroy the initialization of $a$. Secondly, in all constant assignments $a := k$ in parallel threads $k$ equals 0 such that no other values can be generated.

Altogether, the threads are constructed in such a way that the following is valid.

$$\begin{array}{l} L(r_1) \cap \ldots \cap L(r_n) \neq \emptyset \text{ if and only if} \\ x_{n,a_1} := 1 \text{ belongs to the optimal slice.} \end{array} \tag{2}$$

In the following, we describe the intuition underlying the construction and at the same time prove (2).

The threads can be considered to form a ring of processes in which process $\pi_i$ has processes $\pi_{i-1}$ as left neighbor and $\pi_{i+1}$ as right neighbor. Each thread $\pi_i$ ($i = 1, \dots, n$) guesses a word in $L(r_i)$; thread $\pi_0$ guesses some word in $A^*$. The special form of the threads ensures that they can propagate the initialization value 1 for $x_{n,a_1}$ if and only if all of them agree on the guessed word and interleave the corresponding runs in a disciplined fashion. Obviously, the latter is possible iff $L(r_1) \cap \ldots \cap L(r_n) \neq \emptyset$.

Let $w = c_1 \cdot \ldots \cdot c_l$ be a word in $L(r_1) \cap \ldots \cap L(r_n)$ and let $c_0 = a_1$, the first letter in alphabet $A$. In the run induced by $w$ that successfully propagates the value 1, the threads circulate the value 1 around the ring of processes in the variables $x_{i,c_i}$ for each letter $c_i$ of $w$. We call this the *propagation game* in the following. At the beginning of the $j$-th round, $j = 1, \dots, l$, process $\pi_0$ 'proposes' the letter $c_j$ by copying the value 1 from the variable $x_{n,c_{j-1}}$ to $x_{0,c_j}$ in which it was left by the previous round or by the initialization, respectively. For technical reasons this copying is done via the 'local' variable[2] $y_0$. Afterwards the processes $\pi_i$ ($i = 1, \dots, n$) successively copy the value from $x_{i-1,c_j}$ to $x_{i,c_j}$ via their 'local' variables $y_i$. From $x_{n,c_j}$ it is copied by $\pi_0$ in the next round to $x_{0,c_{j+1}}$ and so on. After the last round ($j = l$) $\pi_0$ finally copies the value 1 from $x_{n,c_l}$ to $x_{0,a_1}$ and all processes terminate. Writing—by a little abuse of notation—$\pi_i(a)$ for the single run of $\pi_i(a)$ and $\pi_0(a, b)$ for the single run of $y_0 := x_{n,a}$ ; $\mathsf{KillAll}_0$ ; $x_{0,b} := y_0$, we can summarize above discussion by saying that

$$\begin{array}{l}
\pi_0(a_1, c_1) \cdot \pi_1(c_1) \cdot \ldots \cdot \pi_n(c_1) \cdot \\
\pi_0(c_1, c_2) \cdot \pi_1(c_2) \cdot \ldots \cdot \pi_n(c_2) \cdot \\
\quad \vdots \\
\pi_0(c_{l-1}, c_l) \cdot \pi_1(c_l) \cdot \ldots \cdot \pi_n(c_l) \cdot \\
\pi_0(c_l, a_1)
\end{array}$$

is a run of $\pi_0 \parallel \ldots \parallel \pi_n$ that witnesses that the initialization of $x_{n,a_1}$ belongs to the optimal slice. This implies the 'only if' direction of (2).

Next we show that the construction of the threads ensures that runs that do not follow the propagation game cannot propagate value 1 to the write-instruction. In particular, if $L(r_1) \cap \ldots \cap L(r_n) = \emptyset$, no propagating run exists, which implies the 'if' direction of (2).

Note first that all runs of $\pi_i$ are composed of pieces of the form $\pi_i(a)$ and all runs of $\pi_0$ of pieces of the form $\pi_0(a, b)$ which is easily shown by induction. A run can now deviate from the propagation game in two ways. First, it can follow the rules but terminate in the middle of a round:

$$\begin{array}{l}
\pi_0(a_1, c_1) \cdot \pi_1(c_1) \cdot \ldots \cdot \pi_i(c_1) \cdot \ldots \cdot \pi_n(c_1) \cdot \\
\pi_0(c_1, c_2) \cdot \pi_1(c_2) \cdot \ldots \cdot \pi_i(c_2) \cdot \ldots \cdot \pi_n(c_2) \cdot \\
\quad \vdots \\
\pi_0(c_{m-1}, c_m) \cdot \pi_1(c_m) \cdot \ldots \cdot \pi_i(c_m)
\end{array}$$

Such a run does not propagate the value 1 to the write-instruction as $\mathsf{KillAll}_i$ in $\pi_i(c_m)$ re-initializes $x_{0,a_1}$.

Secondly, a run might cease following the rules of the propagation game after some initial (possibly empty) part. Consider then the first code piece $\pi_i(a)$ or $\pi_0(a, b)$ that is started in negligence of the propagation game rules. It is not hard to see that the first statement in this code piece, $y_i := x_{i-1,a}$ or $y_0 := x_{n,a}$, respectively, then sets the local variable $y_i$ or $y_0$ to zero. The reason is that the propagation game ensures that variable $x_{i-1,a}$ or $x_{n,a}$ holds 0 unless the next statement to be executed according to the rules of the propagation game comes from $\pi_i(a)$ or some $\pi_0(a, b)$, respectively. The subsequent statement $\mathsf{KillAll}_i$ or $\mathsf{KillAll}_0$ then irrevocably re-initializes all the other variables irrespective of the behavior of the other threads as we have shown above. Thus such a run also cannot propagate the value 1 to the write-instruction.

An Owicki/Gries style proof that confirms this fact is contained in the full paper.

# 6. SLICING LOOP-FREE PROGRAMS

We may lower our expectation even more, and ban in addition to procedures also loops from the programs that we expect to slice optimally. But even then, the problem remains intractable, unless P=NP.

THEOREM 3. *Parallel intraprocedural slicing of loop-free programs is NP-complete.*

That the problem is in NP is easy to see. For each statement in the optimal slice we can guess a run that witnesses

[1]Here and in the following, addition and subtraction in subscripts of variables and processes is understood modulo $n+1$.

[2]Variable $y_i$ is not local to $\pi_i$ in a strict sense. But the other threads do not use it as target or source of a copying assignment; they only re-initialize it.

that the statement can affect the slicing criterion. This run can involve each statement in the program at most once as the program is loop-free. Hence its length and consequently the time that is necessary for guessing the run is linear in the size of the given program.

NP-hardness can be proved by specializing the construction from Section 5 to star-free regular expressions. The intersection problem for such expressions is NP-complete.

An alternative reduction from the well-known SAT problem was given in [20]. In contrast to the construction of the current paper, the reduction there relies only on propagation along copying assignments but not on "quasi-synchronization" through well-directed re-initialization of variables. However, this technique does not seem to generalize to the general intraprocedural and the interprocedural case.

# 7. EXTENSIONS

## 7.1 Beyond Fork/Join Parallelism

A weak form of synchronization is inherent in the fork/join parallelism used in this paper as start and termination of threads is synchronized. The hardness results in this paper, however, are not restricted to such settings but can also be shown without assuming synchronous start and termination. Therefore, they also apply to languages like JAVA.

The PSPACE-hardness proof in Section 5, for instance, can be modified as follows. Let $c, d$ be two new distinct letters and $B = A \cup \{c, d\}$. Now $\pi_i$ is defined as $\pi_i(c \cdot r_i \cdot d)$ and the initialization and the final write-instruction is moved to thread $\pi_0$. More specifically, $\pi_0$ is redefined as follows:

$$
\begin{aligned}
\pi_0 = {} & \mathsf{KillAll}_0 \; ; \; x_{0,c} := 1 \; ; \\
& \textbf{loop} \\
& \quad \sqcap \{ y_0 := x_{n,a} \; ; \; \mathsf{KillAll}_0 \; ; \; x_{0,b} := y_0 \mid a, b \in B \} \\
& \textbf{end} \; ; \\
& \textbf{write}(x_{n,d})
\end{aligned}
$$

(Of course the statements $\mathsf{KillX}_i$ have to re-initialize also the new variables $x_{i,c}$ and $x_{i,d}$.) Essentially this modification amounts to requiring that the propagation game is played with a first round for letter $c$—this ensures a quasi-synchronous start of the threads—and a final round for letter $d$—this ensures a quasi-synchronous termination. Thus,

$L(r_1) \cap \ldots \cap L(r_n) \neq \emptyset$ if and only if
$x_{0,c} := 1$ belongs to the optimal slice of $\pi_0 \parallel \ldots \parallel \pi_n$.

Similar modifications work for the reductions in Section 4 and 6.

## 7.2 Further Dataflow Analysis Problems

Our techniques here can be used to obtain similar results also for other optimal program analysis problems, in particular, the detection of *truly life variables* and *copy constants* thereby strengthening recent complexity results for these problems [20].

A variable $x$ is *live* at a program point $p$ if there is a run from $p$ to the end of the program on which $x$ is used before it is overwritten. By referring to [9], Horwitz et. al. [12] define a variable $x$ as *truly live* at a program point $p$ if there is a run from $p$ to the end of the program on which $x$ is used in a truly life context before being defined, where a truly live context means: in a predicate, or in a call to a library routine, or in an expression whose value is assigned to a truly life variable.

Thus, true liveness can be seen as a refinement of the ordinary liveness property. For the programs considered in this paper, the variable initialized in the crucial initialization statement is truly live at that program point if and only if that statement belongs to the optimal slice. Therefore, the lower bounds provided in Theorem 1, 2 and 3 immediately translate to corresponding bounds also for the truly live variable problem. Since the upper bounds PSPACE and NP for intraprocedural and loop-free intraprocedural programs also can be easily verified, we obtain the same complexity characterizations as in Theorem 2 and 3. Indeed, these results are in sharp contrast to the detection of ordinary liveness of a variable at a program point which has been shown to be solvable even in polynomial time [15, 5, 24].

Constant propagation is a standard analysis in compilers. It aims at detecting expressions that are guaranteed to evaluate to the same value in any run of the program, information that can be exploited e.g. for expression simplification or branch elimination. Copy constant detection [7, pp. 660] is a particularly simple variant of this problem in sequential programs. In this problem only assignment statements of the simple forms $x := c$ (constant assignment) and $x := y$ (copying assignment), where $c$ is a constant and $x, y$ are variables, are considered, a restriction obeyed by all programs in this paper. Here, we obtain:

THEOREM 4. *1. The interprocedural copy constant detection problem is undecidable for parallel programs.*

*2. The intraprocedural copy constant detection problem is PSPACE-complete for parallel programs.*

*3. The intraprocedural copy constant detection problem is co-NP-complete for loop-free parallel programs.*

Only a small modification is necessary to apply the reductions in this paper to copy constant detection in parallel programs: the statement $z := 0 \sqcap \textbf{skip}$ must be added just before each write-statement, where $z$ is the written variable. Obviously, this statement prohibits $z$ from being a copy constant of value 1 at the write statement. After this modification $z$ is a copy constant at the write statement (necessarily of value 0) iff the write-statement cannot output the value 1. The latter is the case iff the crucial initialization statement in question does not belong to the optimal slice. This proves the lower bounds in the above theorem. The upper bounds are easily achieved by non-deterministic algorithms that guess paths that witness non-constancy.

Theorem 4 essentially states that optimal detection of copy constants in parallel programs is intractable. This result should be contrasted to the detection problem for *strong* copy constants. Strong copy constants differ from (full) copy constants in that only constant assignments are taken into account by the analysis. In particular, each variable that is a strong copy constant at a program point $p$ is also a copy constant. The detection of strong copy constants turns out to be a much simpler problem as it can be solved in polynomial time [14, 24].

# 8. CONCLUSION

In this paper we have studied the complexity of synchronization-independent program slicing and related dataflow problems for parallel languages. By means of a reduction from the halting problem for two-counter machines, we have shown that the interprocedural problem is undecidable. If we consider programs without procedure calls (intraprocedural problem) the slicing problem becomes decidable but is still intractable. More specifically, we have shown it to be PSPACE-hard by means of a reduction from the intersection problem for regular expressions. Finally, even if we restrict attention to parallel straight-line programs, the problem remains NP-hard. These lower bounds are tight as matching upper bounds are easy to establish.

Previous complexity and undecidability results for data-flow problems for concurrent languages [25, 23] exploit in an essential way synchronization primitives of the considered languages. In contrast our results hold independently of any synchronization. They only exploit interleaving of atomic statements and are thus applicable to a much wider class of concurrent languages.

## 9. REFERENCES


[1] K.-R. Apt and E.-R. Olderog. *Verification of Sequential and Concurrent Programs*. Springer, 1997.

[2] A. Bouajjani and P. Habermehl. Constrained properties, semilinear systems, and Petri nets. In *Concur'96*, LNCS 1119. Springer, 1996.

[3] J. Cheng. Slicing concurrent programs—a graph-theoretical approach. In *Proc. 1st International Workshop on Automated and Algorithmic Debugging*, LNCS 749, pp. 223–240. Springer, 1993.

[4] E. M. Clarke, M. Fujita, S. P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Program slicing for VHDL. In *Charme'99*, Bad Herrenalb, Germany, September 1999.

[5] J. Esparza and J. Knoop. An Automata-theoretic Approach to Interprocedural Data-flow Analysis. In *FOSSACS '99*, LNCS 1578, pp. 14–30. Springer, 1999.

[6] J. Esparza and A. Podelski. Efficient Algorithms for pre* and post* on Interprocedural Parallel Flow Graphs. In *ACM POPL 2000*, pp. 1–11, 2000.

[7] C. Fischer and R. LeBlanc. *Crafting a Compiler*. Benjamin/Cummings Publishing Co., Inc., Menlo Park, CA, 1988.

[8] N. Francez. *Program Verification*. Addison-Wesley, 1992.

[9] R. Giegerich, U. Möncke, and R. Wilhelm. Invariance of approximative semantics with respect to program transformations. In *GI 11. Jahrestagung*, Informatik-Fachberichte 50, pp. 1–10. Springer, 1981.

[10] J. Hatcliff, J. Corbett, M. Dwyer, S. Sokolowski, and H. Zheng. A formal study of slicing for multi-threaded programs with JVM concurrency primitives. In *SAS'99*, LNCS 1694, pp. 1–18. Springer, 1999.

[11] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM TOPLAS*, 12(1):26–61, 1990.

[12] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. Technical Report TR-1283, Computer Sciences Department, University of Wisconsin, Madison, WI, 1995.

[13] M. Iwaihara, M. Nomura, S. Ichinose, and H. Yasuura. Program slicing on VHDL descriptions and its applications. In *Proc. 3rd APCHDL'96*, pp. 132–139, Bangalore, 1996.

[14] J. Knoop. Parallel constant propagation. In *Euro-Par'98*, LNCS 1470, pp. 445–455. Springer, 1998.

[15] J. Knoop, B. Steffen, and J. Vollmer. Parallelism for free: Efficient and optimal bitvector analyses for parallel programs. *ACM TOPLAS*, 18(3):268–299, 1996.

[16] D. Kozen. Lower bounds for natural proof systems. In *IEEE FOCS'77*, pp. 254–266, Long Beach, CA, 1977.

[17] J. Krinke. Static slicing of threaded programs. In *ACM PASTE'98*, pp. 35–42, Montreal, Canada, 1998.

[18] L. I. Millett and T. Teitelbaum. Issues in slicing PROMELA and its applications to model checking, protocol understanding, and simulation. *STTT*, 2(4):343–349, 2000.

[19] M. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, 1967.

[20] M. Müller-Olm. The complexity of copy constant detection in parallel programs. In *STACS 2001*, LNCS 2010, pp. 490–501. Springer, 2001.

[21] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *ACM Software Engineering Symposium on Practical Software Development Environments*, pp. 177–184, 1984.

[22] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6:319–340, 1976.

[23] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM TOPLAS*, 22(2): 416–430, 2000.

[24] H. Seidl and B. Steffen. Constraint-based inter-procedural analysis of parallel programs. In *ESOP'2000*, LNCS 1782, pp. 351–365. Springer, 2000.

[25] R. N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.

[26] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[27] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[28] J. Zhao. Slicing concurrent Java programs. In *7th IEEE International Workshop on Program Comprehension*, pp. 126–133, Pittsburgh, PA, USA, May 1999.


# APPENDIX

## A. THE OWICKI/GRIES-STYLE PROOF

In this appendix we prove the 'if' direction of (1) by means of an Owicki/Gries style program proof [22, 8, 1], a proof that was omitted from the main body of this paper. We assume all notations and definitions of Section 4.

As mentioned, we prove that just before the write instruction in $\pi_M$ the following assertion is valid in the sense of partial correctness, i.e., that any execution reaching this program point satisfies this property:

$$x_n = 1 \;\Rightarrow\; \langle p_0, 0, 0\rangle \to^* \langle p_n, 0, 0\rangle\,. \tag{3}$$

Validity of this assertion corresponds directly to the 'if' direction of (1).

The Owicki/Gries method relies on *proof outlines* which are programs annotated with assertions. Assertions are formulas that represent properties valid for any execution that reaches the program point where the assertion is written down. As usual we write assertions in braces. The annotation is subject to the rules well-known from sequential program proofs. For example if an assignment statement $x := e$ is preceded by an assertion $\{\phi\}$ and followed by an assertion $\{\psi\}$, then $\phi$ must imply $\psi[e/x]$, where $\psi[e/x]$ denotes the assertion obtained by substituting $e$ for $x$ in $\psi$. We assume that the reader is familiar with this style of program proofs (for details see e.g. [22, 8, 1]).

The rule for parallel program looks as follows [1, Rule 19]:

$$\frac{\text{The standard proof outlines } \{p_i\}S_i^*\{q_i\},\ i \in \{1, \dots, n\}, \text{ are interference free}}{\{\bigwedge_{i=1}^n p_i\}[S_1 \parallel \dots \parallel S_n]\{\bigwedge_{i=1}^n q_i\}}$$

In this rule $S_i^*$ stands for an annotated version of parallel component $S_i$ and the requirement that the proof outlines for the component programs are 'standard' means in our context that every atomic statement is surrounded by assertions.

The crucial additional premise for parallel programs is *interference freedom*. The following must be true in an interference free proof outline for a parallel program: Suppose $\{\phi\}$ is an assertion in one parallel component and $S$ is an atomic statement in another parallel component that is preceded by the assertion $pre(S)$. Then $\{\phi \wedge pre(S)\}S\{\phi\}$ must be valid in the usual sense of partial correctness. Intuitively, inference freedom guarantees that validity of an assertion is not destroyed by a thread running in parallel.

## A.1 Enriching the Program

Before we discuss the proof outlines, we enrich the program $\pi_M$ by two variables $c_0$ and $c_1$ that reflect the values of the counters. Initialization statements $c_0 := 0$ and $c_1 := 0$ are added to the *Init* procedure. Furthermore, $c_0$ and $c_1$ are incremented and decremented at appropriate places in $P_0$, $P_{\neq 0}$, $Q_0$, and $Q_{\neq 0}$. (For the purpose of performing the proof we allow more general expressions in assignment statements.) Specifically, the code pieces of the form

$$p := x_k \ ; \ \mathsf{KillAll}_P \ ; \ y_l := p \ ; \ P_{\neq 0}$$

that represent incrementing transitions in $P_0$ and $P_{\neq 0}$ are replaced by

$$p := x_k \ ; \ \mathsf{KillAll}_P \ ; \ c_0 := c_0 + 1 \ ; \ y_l := p \ ; \ P_{\neq 0}$$

and the code pieces after the loop in $P_{\neq 0}$ that represent decrementing transitions are replaced by

$$p := x_k \ ; \ \mathsf{KillAll}_P \ ; \ c_0 := c_0 - 1 \ ; \ y_l := p \, .$$

Analogous modifications are made in $Q_0$ and $Q_{\neq 0}$ for counter $c_1$. It is obvious that Assertion (3) holds in the modified program if and only if it holds in the original program as $c_0$ and $c_1$ are only used in assignments to themselves. ($c_0$ and $c_1$ are *auxiliary variables* in the formal sense of the term used in connection with the Owicki/Gries method. It is well-known that the Owicki/Gries method is incomplete without auxiliary variables [8].)

## A.2 The Proof Outlines

The assertions in the proof ensure that certain configurations are reachable in $M$ if a certain variable in $\pi_M$ holds value 1. We introduce an abbreviation for the formula expressing this fact:

$$\mathsf{OK}(x, s, c_0, c_1) \quad :\Leftrightarrow \quad x = 1 \Rightarrow \langle p_1, 0, 0\rangle \rightarrow^* \langle s, c_0, c_1\rangle$$

Here $x$ is a variable of the constructed program, $s$ is a state of the two-counter machine and $c_0, c_1$ are expressions involving the auxiliary variables from above. Note that Assertion (3) is simply $\mathsf{OK}(x_n, p_n, 0, 0)$.

The global part of the proof outline looks as follows. For clarity, we use a comma to denote conjunction in assertions.

[1] $\{\mathsf{true}\}$
[2] $x_1 := 1$ ;
[3] $\{x_1 = 1\}$
[4] *Init*
[5] $\{x_1 = 1,\ c_0 = 0,\ c_1 = 0,$
$\quad \bigwedge_{i=2}^n x_i = 0,\ \bigwedge_{i=1}^m y_i = 0\}$
[6] $\{c_0 = 0,\ c_1 = 0,$
$\quad \bigwedge_{i=1}^n \mathsf{OK}(x_i, p_i, c_0, c_1),\ \bigwedge_{i=1}^m \mathsf{OK}(y_i, q_i, c_0, c_1)\}$
[7] $(P_0 \parallel Q_0)$ ;
[8] $\{c_0 = 0,\ c_1 = 0,$
$\quad \bigwedge_{i=1}^n \mathsf{OK}(x_i, p_i, c_0, c_1),\ \bigwedge_{i=1}^m \mathsf{OK}(y_i, q_i, c_0, c_1)\}$
[9] $\{\mathsf{OK}(x_n, p_n, 0, 0)\}$
[10] $\mathbf{write}(x_n)$

The obvious proof outline for *Init* is omitted. It is easy to see that [5] implies the assertion in line [6] as $\mathsf{OK}(x, s, 0, 0)$ trivially holds if $x$ holds 0 or if $s$ is $p_1$. It is also obvious that the assertion in line [8] implies the assertion in line [9].

For demonstrating validity of Assertion [8] we prove—by interference free proof outlines—that $P_0$ and $Q_0$ satisfy the following specifications and apply the parallel rule of the Owicki/Gries method:

| | |
|---|---|
| $\{c_0 = 0,$ | $\{c_1 = 0,$ |
| $\bigwedge_{i=1}^n \mathsf{OK}(x_i, p_i, c_0, c_1)\}$ | $\bigwedge_{i=1}^m \mathsf{OK}(y_i, q_i, c_0, c_1)\}$ |
| $P_0$ | $Q_0$ |
| $\{c_0 = 0,$ | $\{c_1 = 0,$ |
| $\bigwedge_{i=1}^n \mathsf{OK}(x_i, p_i, c_0, c_1)\}$ | $\bigwedge_{i=1}^m \mathsf{OK}(y_i, p_i, c_0, c_1)\}$ |

Simultaneously, we prove similar specifications for $P_{\neq 0}$ and $Q_{\neq 0}$ that are parameterized by a constant $k > 0$:

| | |
|---|---|
| $\{c_0 = k,$ | $\{c_1 = k,$ |
| $\bigwedge_{i=1}^n \mathsf{OK}(x_i, p_i, c_0, c_1)\}$ | $\bigwedge_{i=1}^m \mathsf{OK}(y_i, q_i, c_0, c_1)\}$ |
| $P_{\neq 0}$ | $Q_{\neq 0}$ |
| $\{c_0 = k - 1,$ | $\{c_1 = k - 1,$ |
| $\bigwedge_{i=1}^n \mathsf{OK}(x_i, p_i, c_0, c_1)\}$ | $\bigwedge_{i=1}^m \mathsf{OK}(y_i, q_i, c_0, c_1)\}$ |

As we are concerned with partial correctness, it suffices to show that the body of the procedures satisfy these specification, under the assumption that recursive calls do.

In the following we present the proof outlines for $P_0$ and $P_{\neq 0}$ in detail; the proofs for $Q_0$ and $Q_{\neq 0}$ are completely analogous. Afterwards we show interference freedom, a proof that reflects crucial properties of our construction.

The first goal is to show that the precondition of each procedure is an invariant of the loop in the body of that procedure. This amounts to proving that each path through the loop preserves the precondition. Let $k = 0$ for the proof in $P_0$ and $k > 0$ for the proof in $P_{\neq 0}$.

This is the proof for the paths induced by skip-transitions in both procedures or test-decrement transitions in $P_0$ :

[11] $\{c_0 = k,\ \bigwedge_{i=1}^{n} \mathsf{OK}(x_i, p_i, c_0, c_1)\}$
[12] $p := x_k$ ;
[13] $\{c_0 = k,\ \mathsf{OK}(p, p_k, c_0, c_1)\}$
[14] $\mathsf{KillAll}_P$
[15] $\{c_0 = k,\ \mathsf{OK}(p, p_k, c_0, c_1),$
$\bigwedge_{i=1}^{m} y_i = 0,\ q = 0,\ \bigwedge_{i=1}^{n} x_i = 0\}$
[16] $y_l := p$
[17] $\{c_0 = k,\ \bigwedge_{i=1}^{n} \mathsf{OK}(x_i, p_i, c_0, c_1)\}$

Instruction [16] leaves all variables $x_i$ untouched. Hence, it establishes its postcondition [17], because all $x_i$ are ensured to be zero in [15] and $\mathsf{OK}(x_i, p_i, c_0, c_1)$ holds trivially if $x_i = 0$. It may be surprising that the conjunct $\mathsf{OK}(p, p_k, c_0, c_1)$ is not needed in this proof because, intuitively, it captures a crucial property of the construction. The reason is that the proofs of $P_0$ and $P_{\neq 0}$ establish only a property about the $x_i$. The conjunct $\mathsf{OK}(p, p_k, c_0, c_1)$ is, however, important to ensure interference freedom of [16] with the proof outlines for $Q_0$ and $Q_{\neq 0}$ that concern the variables $y_i$.

The specification of $\mathsf{KillAll}_P$, viz. $\{[13]\}\ \mathsf{KillAll}_P\ \{[15]\}$, is again parameterized by a constant $k \geq 0$ and is also used in the proof outlines that follow. It is straightforward to construct a proof outline witnessing this specification: the variables that have already been re-initialized are collected in an increasingly larger conjunction.

The proof outline for the paths through the loop bodies induced by incrementing transitions is similar but has to reflect the change of the counter. It also applies the assumption about recursive calls of $P_{\neq 0}$ (for $k_{\text{new}} := k + 1$):

[18] $\{c_0 = k,\ \bigwedge_{i=1}^{n} \mathsf{OK}(x_i, p_i, c_0, c_1)\}$
[19] $p := x_k$ ;
[20] $\{c_0 = k,\ \mathsf{OK}(p, p_k, c_0, c_1)\}$
[21] $\mathsf{KillAll}_P$
[22] $\{c_0 = k,\ \mathsf{OK}(p, p_k, c_0, c_1),$
$\bigwedge_{i=1}^{m} y_i = 0,\ q = 0,\ \bigwedge_{i=1}^{n} x_i = 0\}$
[23] $c_0 := c_0 + 1$
[24] $\{c_0 = k + 1,\ \mathsf{OK}(p, p_k, c_0 - 1, c_1),$
$\bigwedge_{i=1}^{m} y_i = 0,\ q = 0,\ \bigwedge_{i=1}^{n} x_i = 0\}$
[25] $y_l := p$
[26] $\{c_0 = k + 1,\ \bigwedge_{i=1}^{n} \mathsf{OK}(x_i, p_i, c_0, c_1)\}$
[27] $P_{\neq 0}$
[28] $\{c_0 = k,\ \bigwedge_{i=1}^{n} \mathsf{OK}(x_i, p_i, c_0, c_1)\}$

This completes the proof that the preconditions of $P_0$ and $P_{\neq 0}$ are loop invariants and also finishes the proof outline for $P_0$, as its pre- and postcondition coincide and its body just consists of the loop.

It remains to show that the paths from the loop exit to the procedure exit in $P_{\neq 0}$ induced by decrementing transitions establish the postcondition from the loop invariant, i.e. the precondition of $P_{\neq 0}$:

[29] $\{c_0 = k,\ \bigwedge_{i=1}^{n} \mathsf{OK}(x_i, p_i, c_0, c_1)\}$
[30] $p := x_k$ ;
[31] $\{c_0 = k,\ \mathsf{OK}(p, p_k, c_0, c_1)\}$
[32] $\mathsf{KillAll}_P$
[33] $\{c_0 = k,\ \mathsf{OK}(p, p_k, c_0, c_1),$
$\bigwedge_{i=1}^{m} y_i = 0,\ q = 0,\ \bigwedge_{i=1}^{n} x_i = 0\}$
[34] $c_0 := c_0 - 1$ ;
[35] $\{c_0 = k - 1,\ \mathsf{OK}(p, p_k, c_0 + 1, c_1),$
$\bigwedge_{i=1}^{m} y_i = 0,\ q = 0,\ \bigwedge_{i=1}^{n} x_i = 0\}$
[36] $y_l := p$
[37] $\{c_0 = k - 1,\ \bigwedge_{i=1}^{n} \mathsf{OK}(x_i, p_i, c_0, c_1)\}$

## A.3 Interference Freedom

Let us now check interference freedom. We look at each type of assignment found in $Q_0$ and $Q_{\neq 0}$. It is clear that an assignment to a variable $z$ cannot invalidate conjuncts in assertions that do not mention $z$. Therefore, we only need to consider conjuncts in assertions mentioning the variable to which the statement in question assigns.

- $x_i := 0$, $y_i := 0$, $p := 0$: these re-initializing assignment statements cannot invalidate any assertion in the proof outlines because all conjuncts that mention the left-hand-side variable trivially hold if the variable is zero. This holds in particular for conjuncts of the form $\mathsf{OK}(x, s, c_0, c_1)$.
- $c_1 := c_1 + 1$ and $c_1 := c_1 - 1$: all conjuncts of the form $\mathsf{OK}(p, p_k, c_0, c_1)$ or $\mathsf{OK}(x_i, p_i, c_0, c_1)$ could potentially be invalidated by these statements. The incrementations and decrementations of $c_1$ are however—in analogy to [22] and [33]—guarded by a precondition that ensures that $p$ as well as all variables $x_i$ hold zero, which make $\mathsf{OK}(p, p_k, c_0, c_1)$ or $\mathsf{OK}(x_i, p_i, c_0, c_1)$ true for trivial reasons.

  Note that this argument exploits that the variables are re-initialized in order to avoid 'cheating'.
- $q := y_k$: such a statement could potentially invalidate a conjunct of the form $q = 0$. However, the conjunct $q = 0$ appears in assertions only together with the conjunct $\bigwedge_{i=1}^{m} y_i = 0$. In particular this holds in the (omitted) proof outline for $\mathsf{KillAll}_p$ because the variables $y_i$ are re-initialized before $q$. Therefore, $q := y_k$ cannot destroy validity of the assertion.

  Note that it is essential for this argument to work, that the re-initializations in $\mathsf{KillAll}_P$ are done in the correct order as discussed in Section 4.2.
- $x_l := q$: such a statement could potentially invalidate conjuncts of the form $x_l = 0$ or $\mathsf{OK}(x_l, p_l, c_0, c_1)$.

  All assertions that contain $x_l = 0$ also contain a conjunct $q = 0$. Thus we can argue as for instructions of the form $q := y_k$.

  For conjuncts of the form $\mathsf{OK}(x_l, p_l, c_0, c_1)$ the argument is more subtle. Similarly to [15], [24], and [35], $x_l := q$ is preceded by an assertion that ensures in particular that $\mathsf{OK}(q, q_k, c_0, c_1 + \iota)$ holds, where $\iota \in \{-1, 0, 1\}$. By the construction of $\pi_M$, $\iota = -1$, 1, or 0 iff there is a transition from $q_k$ to $p_l$ that increments, decrements, or leaves the counter $c_1$ unchanged, respectively. Now suppose that $x_l$ is assigned the value 1 by $x_l := q$, otherwise $\mathsf{OK}(x_l, p_l, c_0, c_1)$ holds trivially. Then clearly $q = 1$ which implies $\langle p_1, 0, 0\rangle \to^* \langle q_k, c_0, c_1 + x\rangle$ by $\mathsf{OK}(q, q_k, c_0, c_1 + x)$. By the transition from $q_k$ to $p_l$, this transition sequence can now be extended to a sequence $\langle p_1, 0, 0\rangle \to^* \langle p_l, c_0, c_1\rangle$. Hence, $\mathsf{OK}(x_l, p_l, c_0, c_1)$ holds.

It is interesting to observe that the crucial properties of the construction are reflected in the interference freedom proof rather than the local proofs. Note, however, that the interference freedom proof massively relies on the preconditions of the interleaving statements that are established by the local proofs.