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 π whether or not S affects a specified set of state-
ments, given that all conditionals in π are interpreted as
non-deterministic choices.

Only recently, it has been shown that reachability of pro-
gram 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 pro-
cedure 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 syn-
chronization, 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 prob-
lems 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
Mennings of Programs]: Semantics of Programming Lan-
guages—Program analysis; F.4.3 [Mathematical Logic
and Formal Languages]: Formal Languages—classes de-
bined by grammars or automata, decision problems

*The second author was supported by the RTD project
IST-1999-10557 "DAEDALUS" of the European FP5 pro-

Permission to make digital or hard copies of all or part of this work
for personal or classroom use is granted without fee provided that
copies are not made or distributed for profit or commercial advan-
tage and that copies bear this notice and the full citation on the first
page. To copy otherwise, to republish, to post on servers or to
redistribute to lists, requires prior specific permission and/or a fee.

STOC’01, July 6–8, 2001, Hersonissos, Crete, Greece.
Copyright 2001 ACM 1-58113-349-9/01/0007 ...$5.00.

Keywords
Slicing, parallel programs, interprocedural analysis, unde-
cidability, complexity

1. INTRODUCTION
Static program slicing [27] is an established program re-
duction technique that has applications in program under-
standing, debugging, and testing [26]. More recently, it has
also been proposed as a technique for ameliorating the state-
expllosion problem when formally verifying software or hard-
ware [13, 10, 4, 18]. The goal of program slicing is to iden-
tify and remove parts of the program that cannot (potentially)
influence certain value(s) at certain program point(s) of in-
terest. The latter is called the slicing criterion.

There is a vast amount of literature on slicing sequential
languages (see the references in Tipp’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 pro-
posed to use a structure called PDG (Program Dependence
Graph). A PDG captures two kinds of dependences, data de-
pendences and control dependences. Intuitively, a state-
ment 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 pro-
gram 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 lan-
guages was J. Cheng [8]. 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
discussions have in common that slicing is again approached as a
backwards reachability problem but on some extended form
of PDG (called Process Dependence Net [8], Multithreaded
Dependence Graph [28], etc.). These structures model fur-
ther 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 depend-
ence 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 pro-
gram 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 exec-
ution path. Dataflow analysis in general suffers from this
problem and the common remedy is to ignore conditions al-
together when defining feasible paths. In other words, condi-
tional branching is interpreted as non-deterministic branch-
ing, 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 pro-
decreases, PDG-based slicing is efficient and optimal. Optimal-
ity 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 ques-
tion [11]. This analysis can be done in polynomial time [26].
For concurrent languages with procedures and synchroniza-
tion 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 opti-
mal 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 unde-
cidable. 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 aban-
donned. The latter two lower complexity bounds are optimal,
as they match the corresponding upper bounds.

We conclude that all efficient slicing algorithms for con-
current languages are doomed to be sub-optimal (unless P=P-
SPACE). Our results are shown under very weak assump-
tions on the concurrent language. Intuitively, they ex-
pose a weakness of interference dependence only. As no
synchronization properties are exploited, our results point
to a more fundamental limitation for slicing concurrent lan-
guages 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 cer-
tain sense, this comes as a surprise, as only slightly simpler
analysis questions, namely, strong copy constant propaga-
tion 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

\[
\begin{align*}
  a &:= 1 \\
  b &:= a \\
  c &:= b \\
  \text{fork} & \quad \text{join} \\
  b &:= 0 \\
  c &:= 0 \\
  \text{write}(c) & \quad \text{write}(c)
\end{align*}
\]

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

Figure 1: An illustrative example.

slicing parallel programs. Consider the program

\[
a := 1; \{ b := 1; b := 0; c := 0 \} || c := b : \text{write}(c).
\]

In Fig. 1 (a) a control flow graph-like representation of the
program is shown and in (b) the data and interference de-
pendences. 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 conve-
niently; 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 pro-
gram that realizes all dependences in this path and therefore
an optimal slicer must remove \(a := 1\). In order to see this con-
sider that in an execution \(b := 0\) must be executed either
before or after \(c := b\) in the parallel thread. If it is exec-
buted before \(c := b\) then it kills the propagation from \(b := a\)
to \(c := b\). If it is executed after \(c := b\) then the sub-
sequent statement \(c := 0\) kills the propagation from \(c := b\) to
\(\text{write}(c)\). Our undecidability and hardness results exploit
that propagation can be prohibited in this way by means of
re-initializations. Kshirsagar [17] also mentions that ‘inter-
ference 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 interprocedural par-
allel 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 assign-
ments of a very simple form are needed: \(x := k\) where \(k\) is
either a constant or a variable.

A procedure procedure comprises a finite set Proc of
procedure names containing a distinguished name Main.
Each procedure name P is associated with a statement \(s_P\),
the corresponding procedure body constructed according to
the following grammar, in which Q ranges over Proc \{Main\}

\[
\begin{align*}
  x &:= k \\
  x &:= y \\
  x &:= x + y \\
  \text{fork} & \quad \text{join} \\
  x &:= \text{write}(x) \\
  x &:= \text{read}(x)
\end{align*}
\]
and \( x \) over some given finite set of variables:

\[
\begin{align*}
\varepsilon & := c \mid x \\
\pi & := x := e \mid \text{write}(e) \mid \text{skip} \mid Q \mid \pi_1 ; \pi_2 \\
& \mid \pi_1 \mid \pi_2 \mid \pi_1 \cap \pi_2 \mid \text{loop } \pi \text{ end }
\end{align*}
\]

We use the syntax procedure \( P; \pi \text{ end} \) 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 \( \text{skip} \), 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 \( \cap \) parallel composition. The operator \( \cap \) represents non-deterministic branching and loop \( \pi \text{ end} \) stands for a loop that iterates \( \pi \) an indefinite number of times. Such constructs 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; \( \cap \{ \pi_1, \ldots, \pi_m \} \) denotes \( \pi_1 \cap \cdots \cap \pi_m \). The ambiguity inherent in this notation is harmless because \( \cap \) 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 an order in an execution of the program. The program \( x_0 := x \mid x_0 := y \mid y := x \), for example, has the three runs \( x_0 := x, x_0 := y, x_0 := x \), \( x_0 := x, y := x, x := y \), \( y := x, x := 1, x := y \). We denote the set of runs of program \( \pi \) by \( \text{Runs}(\pi) \).

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 the final state \( p_n \) is associated with an instruction \( I(s) \) taken from the following selection:

- \( c_i := c_i + 1; \text{goto } s' \) (increment),
- \( c_i := c_i - 1; \text{goto } s'' \) (test-decrement), or
- \( \text{goto } s' \) (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, \ldots, n, \!:

- \( I(s) = c_i := c_i + 1; \text{goto } s', x'_i = x_i + 1, \! x_{i\! -\! 1} = x_{i\! -\! 1} \).
- \( I(s) = \text{if } c_i = 0 \text{ then goto } s' \text{ else } c_i := c_i - 1; \text{goto } s'', x_1 = x_1 - 1, \! x_{i\! -\! 1} = x_{i\! -\! 1} \).
- \( I(s) = \text{if } c_i = 0 \text{ then goto } s' \text{ else } c_i := c_i - 1; \text{goto } s'', x_1 = x_1 - 1, \! x_{i\! -\! 1} = x_{i\! -\! 1} \).
- \( I(s) = \text{goto } s' \text{ if } x_i = x_i, \! x_{i\! -\! 1} = x_{i\! -\! 1} \).

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 \( \langle p_1, 0, 0 \rangle \rightarrow_{M*} \langle p_n, x_0, x_1 \rangle \) 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
\[ \begin{align*}
    & p := x_k; \text{ KillAllP; } y := p; \text{ } P_{p_0} \mid \\
    & l(p_k) = c_0 := c_0 + 1; \text{ goto } q_k \} \tag{$\star$} \\
    & p := x_k; \text{ KillAllP; } y := p \mid \\
    & l(p_k) = \text{ if } c_0 = 0 \text{ then goto } q_k \text{ else } ... \} \tag{$\star$}
\end{align*} \]
end loop
end procedure $P_{p_0}$;

\begin{figure}[h]
\centering
\begin{align*}
\text{procedure KillAllP; } y := 0; \ldots; y_m := 0; q := 0; x_1 := 0; \ldots; x_n := 0 \end{align*}
\caption{Definition of $P_0$ and $P_{p_0}$.}
\end{figure}

\begin{figure}[h]
\centering
\begin{align*}
\text{procedure Q0; } \\
\text{loop } \begin{align*}
    & q := y_i; \text{ KillAllQ; } x_i := q; \text{ } Q_{q_0} \mid \\
    & l(q_k) = c_1 := c_1 + 1; \text{ goto } p_k \} \tag{$\star$} \\
    & q := y_i; \text{ KillAllQ; } x_i := q \mid \\
    & l(q_k) = \text{ if } c_1 = 0 \text{ then goto } p_k \text{ else } ... \} \tag{$\star$}
\end{align*}
end loop
end procedure Q_{q_0};

\begin{align*}
\text{procedure KillAllQ; } x_1 := 0; \ldots; x_n := 0; p := 0; y_i := 0; \ldots; y_m := 0 \end{align*}
\caption{Definition of $Q_0$ and $Q_{q_0}$.}
\end{figure}

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 $((p_1, 0, 0) \rightarrow (p_2, x_0, x_1) \text{ } \text{ for some } x_0, x_1)$ and $((p_1, 0, 0) \rightarrow (p_2, 0, 0))$ are equivalent. We assume in the following that such loops have been added to the given machine.

\subsection{Constructing a Program}

From a two-counter machine as above we construct a parallel program, $P_M$. For each state $p_k \in P$ the program uses a variable $x_k$ and for each state $q_i \in Q$ a variable $y_i$. 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 $y_i$.

The main procedure of $P_M$ reads as follows:

\begin{align*}
\text{procedure Main;} \\
\text{ procedure Init; } \\
\text{ procedure Write(x_n); } \\
\text{ (P_0 || Q_0) ; write(x_n) } \\
\text{ end end end}
\end{align*}

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.

\[ M \text{ terminates if and only if } x_1 \text{ may hold } 1 \text{ at the write-statement.} \tag{1} \]

The initialization of all variables except $x_1$ by 0 reflects that $p_i$ is the initial state. For each of the two counters the program uses two procedures, $P_0$ and $P_{p_0}$ for counter $c_0$ and $Q_0$ and $Q_{q_0}$ for counter $c_1$. Their definition can be found in Fig. 2 and 3. We describe $P_0$ and $P_{p_0}$ in detail in the following. $Q_0$ and $Q_{q_0}$ are completely analogous.

Intuitively, $P_0$ and $P_{p_0}$ 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 $q_i$ is represented by copying $x_k$ to $y_i$ via an auxiliary variable $p$ and re-initializing $x_k$ by zero as part of KillAllP. 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 $P_{p_0}$ that simulates the transitions induced by the incremented counter. A return from this new instance of $P_{p_0}$ 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 $P_{p_0}$.

- For test-decrement transitions simulated in $P_{p_0}$, 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 $P_{p_0}$ 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 $q_i$, although $x_1$ 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 Q0 and Q{p=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 Q0 and Q{p=0} are of the form q := y or x := q (q is the analog of the auxiliary variable p). Thus, we can safely assign 0 to the y in P0 and P{p=0} as they are not the target of a copy instruction in Q0 or Q{p=0}. After we have done so, we can safely assign 0 to q; a copy instruction q := y, executed by the parallel thread cannot destroy the value 0 as all y contain 0 already. After that we can safely assign 0 to the x, by a similar argument. This explains the definition of KillAll.

4.3 Correctness of the Reduction

From the intuition underlying the definition of πM, the ‘only if’ direction of (1) is rather obvious: If M terminates, i.e., if it has transitions leading from (p1, . . . , pn, 0) to (p1, . . . , pn, 0), we can simulate these transitions by a propagating run of πM. By explaining the definition of KillAll, 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 πM the following assertion is valid:

\[ x_{n+1} = 1 \Rightarrow (p_1, 0, 0) \rightarrow^* (p_n, 0, 0) . \]

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 Bonajjani 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. Bonajjani and Habermehl use the two recursion stacks to simulate two counters (as we do) whose joint operation 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 r1, . . . , rn of regular expressions over some finite alphabet A. The problem is to decide whether \( L(r_1) \cap \cdots \cap L(r_n) \) is non-empty.

**Lemma 1.** The Regular Expression Intersection problem is PSPACE-complete. \( \square \)

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 problem 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 \( \tau_0, \ldots, \tau_n \) after some initialization of the variables used in the program:

```
procedure Main;
    KillXY0; \ldots; KillXYn; \; x_{r_1, i} := 1 ;
    \{\tau_0 \perp \tau_1 \perp \cdots \perp \tau_n\}; write(x_{r_1, i})
end
```

The threads refer to variables \( x_{i, a} \) and \( y_i \) (\( i \in \{0, \ldots, n\} \)). Thread \( \tau_0 \) is defined as follows.

```
\tau_0 = \text{loop} \\
\{ y_0 := x_{r_1, i} ; \text{KillAll} ; x_{0, b} := y_0 \mid a, b \in A \} \\
end
```

The statement KillAll 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 \( \tau_i \) 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.

\[
\begin{align*}
\pi_0(r) &= \text{skip} \\
\pi_i(a) &= y_i := x_{i-1,a}; \text{KillAll} \ ; x_{i,a} := y_i \\
\pi_i(t_1 \cdot t_2) &= \pi_i(t_1) \oplus \pi_i(t_2) \\
\pi_i(t_1 + t_2) &= \pi_i(t_1) \oplus \pi_i(t_2) \\
\pi_i(r^*) &= \text{loop } \pi_i(r) \text{ end}
\end{align*}
\]

The statement \( \text{KillAll} \) re-initializes all variables except \( y_i \). This statement as well as statements \( \text{KillX} \) and \( \text{KillXY} \) on which its definition is laced are defined as follows.

\[
\begin{align*}
\text{KillX} &= x_{i-1, a} := 0; \ldots; x_{j-1, a} := 0 \\
\text{KillY} &= y_i := 0; \text{KillX}, \\
\text{KillAll} &= \text{KillX}; \text{KillXY}_{i+1}; \ldots; \text{KillXY}_{n}; \\
\text{KillXY} &= x_{i,a} := 0; \text{KillXY}_{i-1}
\end{align*}
\]

Again it is not obvious that thread \( \pi_i \) can safely re-initialize the variables because the other threads may arbitrarily interfere. But by exploiting that only copies instructions of the form \( y_i := x_{i-1,a} \) and \( x_{i,a} := y_i \) 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 assignment 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.

\[
\begin{align*}
L(r_1) \cap \ldots \cap L(r_m) \neq \emptyset \text{ if and only if} \\
x_{i,a} := 1 \text{ belongs to the optimal slice.} \quad (2)
\end{align*}
\]

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 \( \pi_i \) \((i = 1, \ldots, n)\) 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 \( x_{n, a} \) if and only if all of them agree on the guessed word and interfere the corresponding runs in a disciplined fashion. Obviously, the latter is possible iff \( L(r_1) \cap \ldots \cap L(r_m) \neq \emptyset \).

Let \( w = c_1 \cdots c_m \) be a word in \( L(r_1) \cap \ldots \cap L(r_k) \) and let \( c_0 = c_1 \), 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 \( x_{i,c_j} \) for each letter \( c_j \) of \( w \). We call this the \textit{propagation game} in the following. At the beginning of the \( j \)-th round, \( j = 1, \ldots, k \), process \( \pi_0 \) proposes the letter \( c_j \) by copying the value 1 from the variable \( x_{m,c_{j-1}} \) 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 \( \pi_i \) \((i = 1, \ldots, n)\) successively copy the value from \( x_{i-1,c_j} \) to \( x_{i,c_j} \) via their ‘local’ variables \( y_i \). From \( x_{m,c_j} \) 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 \( x_{m,c_l} \) to \( x_{0,0} \) 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 \( y_0 := x_{0, a} \); \text{KillAll} \ ; x_{0, b} := y_0 \), we can summarize above discussion by saying that

\[
\begin{align*}
\pi_0(c_1, c_2) \cdots \pi_0(c_{i-1}, c_i) \\
\pi_0(c_{i-1}, c_i) \cdots \pi_0(c_{n-1}, c_n) \\
\vdots
\end{align*}
\]

is a run of \( \pi_0 \| \ldots \| \pi_n \) that witnesses that the initialization of \( x_{n, a} \) belongs to the optimal slice. This implies the ‘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 ‘then’ 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:

\[
\begin{align*}
\pi_0(c_1, c_2) \cdots \pi_0(c_{i-1}, c_i) \\
\pi_0(c_{i-1}, c_i) \cdots \pi_0(c_{n-1}, c_n) \\
\vdots
\end{align*}
\]

Such a run does not propagate the value 1 to the write-instruction as \( \text{KillAll} \) in \( \pi_0(c_{m+1}, c_m) \) re-initializes \( x_{0,a} \).

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, \( y := x_{n-1,a} \) or \( y_0 := x_{n, a} \), respectively, then sets the local variable \( y \) or \( y_0 \) to zero. The reason is that the propagation game ensures that variable \( x_{n-1,a} \) 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 \( \text{KillAll} \) or \( \text{KillAll} \) 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 \( P = \text{NP} \).

**Theorem 3.** Parallel intraprocedural slicing of loop-free programs is \( \text{NP-complete} \).

That the problem is in \( \text{NP} \) is easy to see. For each statement in the optimal slice we can guess a run that witnesses
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 intra-procedural 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 \( e, d \) be two new distinct letters and \( B = A \cup \{e, d\} \). Now \( \pi_i \) is defined as \( \pi_i(e \cdot r_i \cdot d) \) and the initialization and the final write-instruction is moved to thread \( \pi_0 \). More specifically, \( \pi_0 \) is redefined as follows:

\[
\pi_0 = \text{ KillAll} : x_0 \leftarrow 1 ; \\
\text{loop} \\
\{ \{y_0 \leftarrow x_{n, a} : \text{ KillAll} : x_{0, b} \leftarrow y_0 \} | a, b \in B \} \\
\text{end} ; \\
\text{write}(x_{a, d})
\]

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

\[
I(e_1) \cap \ldots \cap I(e_n) \neq \emptyset \text{ if and only if } x_{0, e} \leftarrow 1 \text{ belongs to the optimal slice of } \pi_0 \parallel \ldots \parallel \pi_n.
\]

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 optional program analysis problems, in particular, the detection of truly live 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 live 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 live 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 inappradural 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 interprocedural copy constant detection problem is PSPACE-complete for parallel programs.

3. The interprocedural 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 \( z := 0 \cap \text{skip} \) 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) if the write statement cannot output the value 1. The latter is the case if 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 (intrproce-
dural 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


APPENDIX

A. THE O WICKI/G RIE S-ST YLE P ROOF

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 instruc-
tion 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:

$$x_n = 1 \Rightarrow \langle p_0, 0, 0 \rangle \rightarrow^* \langle p_n, 0, 0 \rangle.$$  (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 notation 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 \( \psi[x/\phi] \), where \( \psi[x/\phi] \) 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]:

\[
\{ A_{n \rightarrow i} \} \{ S_i \mid \cdots \mid S_n \} \{ A_{n \rightarrow i} \} \{ q \}
\]

In this rule \( S^* \) stands for an annotated version of parallel component \( S \), 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 \( \text{pre}(S) \). Then \( \{ \phi \wedge \text{pre}(S) \} \} S \} \{ \phi \} \) must be valid in the usual sense of partial correctness. Intuitively, interference 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 \text{Init} procedure. Furthermore, \( c_0 \) and \( c_1 \) are incremented and decremented at appropriate places in \( P_0 \), \( P_{s0} \), \( Q_0 \), and \( Q_{s0} \). (For the purpose of performing the proof we allow more general expressions in assignment statements.) Specifically, the code pieces of the form

\[
p := x_i \mid \text{KillAll} \; p \; ; y_i := p \mid \; P_{s0}
\]

that represent incrementing transitions in \( P_0 \) and \( P_{s0} \) are replaced by

\[
p := x_i \mid \text{KillAll} \; p \; ; c_0 := c_0 + 1 \; ; y_i := p \mid \; P_{s0}
\]

and the code pieces after the loop in \( P_{s0} \) that represent decrementing transitions are replaced by

\[
p := x_i \mid \text{KillAll} \; p \; ; c_0 := c_0 - 1 \; ; y_i := p .
\]

Analogous modifications are made in \( Q_0 \) and \( Q_{s0} \) 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:

\[
\mathcal{O}(x, s, c_0, c_1) : \iff x = 1 \Rightarrow (p_1, 0, 0) \rightarrow^* (s, c_0, c_1)
\]

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 \( \mathcal{O}(x, s, p_0, 0, 0) \).

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

\[
\begin{align*}
[1] & \quad \{ \text{true} \} \\
[2] & \quad x_1 := 1 \\
[3] & \quad (x_1 = 1) \\
[4] & \quad \text{Init} \\
[5] & \quad (x_1 = 1, c_0 = 0, c_1 = 0, \\
& \quad \mathcal{A}_{n \rightarrow i} \mathcal{O}(x, s, c_0, c_1), \mathcal{A}_{n \rightarrow i} \mathcal{O}(y, s, c_0, c_1)) \\
[6] & \quad (c_0 = 0, c_1 = 0, \\
& \quad \mathcal{A}_{n \rightarrow i} \mathcal{O}(x, s, c_0, c_1), \mathcal{A}_{n \rightarrow i} \mathcal{O}(y, s, c_0, c_1)) \\
[7] & \quad (x_1 = 1, c_0 = 0, c_1 = 0, \\
& \quad \mathcal{A}_{n \rightarrow i} \mathcal{O}(x, s, c_0, c_1), \mathcal{A}_{n \rightarrow i} \mathcal{O}(y, s, c_0, c_1)) \\
[8] & \quad (x_1 = 1, c_0 = 0, c_1 = 0, \\
& \quad \mathcal{A}_{n \rightarrow i} \mathcal{O}(x, s, c_0, c_1), \mathcal{A}_{n \rightarrow i} \mathcal{O}(y, s, c_0, c_1)) \\
[9] & \quad (\mathcal{O}(p_0, s, p_0, 0, 0)) \\
[10] & \quad \text{write}(x_1)
\end{align*}
\]

The obvious proof outline for \text{Init} is omitted. It is easy to see that [5] implies the assertion in line [6] as \( \mathcal{O}(x, s, p_0, 0, 0) \) trivially holds if \( x_1 \) 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:

\[
\begin{align*}
\{ c_0 = 0, \mathcal{A}_{n \rightarrow i} \mathcal{O}(x, s, c_0, c_1) \} & \quad \{ c_1 = 0, \mathcal{A}_{n \rightarrow i} \mathcal{O}(x, s, c_0, c_1) \} \\
\{ (c_0 = 0, \mathcal{A}_{n \rightarrow i} \mathcal{O}(x, s, c_0, c_1) \} & \quad \{ c_1 = 0, \mathcal{A}_{n \rightarrow i} \mathcal{O}(x, s, c_0, c_1) \} \\
\{ c_0 = k, \mathcal{A}_{n \rightarrow i} \mathcal{O}(x, s, c_0, c_1) \} & \quad \{ c_1 = k, \mathcal{A}_{n \rightarrow i} \mathcal{O}(x, s, c_0, c_1) \} \\
\{ (c_0 = k, \mathcal{A}_{n \rightarrow i} \mathcal{O}(x, s, c_0, c_1) \} & \quad \{ c_1 = k, \mathcal{A}_{n \rightarrow i} \mathcal{O}(x, s, c_0, c_1) \}
\end{align*}
\]

Simultaneously, we prove similar specifications for \( P_{s0} \) and \( Q_{s0} \) that are parameterized by a constant \( k > 0 \):

\[
\begin{align*}
\{ c_0 = k, \mathcal{A}_{n \rightarrow i} \mathcal{O}(x, s, c_0, c_1) \} & \quad \{ c_1 = k, \mathcal{A}_{n \rightarrow i} \mathcal{O}(x, s, c_0, c_1) \} \\
\{ (c_0 = k, \mathcal{A}_{n \rightarrow i} \mathcal{O}(x, s, c_0, c_1) \} & \quad \{ c_1 = k, \mathcal{A}_{n \rightarrow i} \mathcal{O}(x, s, c_0, c_1) \} \\
\{ c_0 = k - 1, \mathcal{A}_{n \rightarrow i} \mathcal{O}(x, s, c_0, c_1) \} & \quad \{ c_1 = k - 1, \mathcal{A}_{n \rightarrow i} \mathcal{O}(x, s, c_0, c_1) \}
\end{align*}
\]

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 \( P_{s0} \) in detail; the proofs for \( Q_0 \) and \( Q_{s0} \) 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 \( P_{s0} \).

This is the proof for the paths inclined by skip-transitions in both procedures or test-decrement transitions in \( P_0 \):
A.3 Interference Freedom

Let us now check interference freedom. We look at each type of assignment found in $Q_0$ and $Q_{\mathit{add}}$. 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 $\mathit{OK}(x, p, c_0, c_1)$.
- $c_i := c_i + 1$ and $c_i := c_i - 1$: all conjuncts of the form $\mathit{OK}(p, p_i, c_0, c_1)$ or $\mathit{OK}(x, p, c_0, c_1)$ could potentially be invalidated by these statements. The incrementations and decrements of $c_i$ are however—in analogy to [22] and [33]—guarded by a precondition that ensures that $p$, as well as all variables $x$, hold zero, which make $\mathit{OK}(p, p_i, c_0, c_1)$ or $\mathit{OK}(x, p, c_0, c_1)$ true for trivial reasons.

Note that this argument exploits that the variables are re-initialized in order to avoid ‘cheating’.

- $q := y_i$: 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 $\mathit{OK}(x, p, c_0, c_1)$. In particular this holds in the (omitted) proof outline for $\mathit{KillAll}$, because the variables $y_i$ are re-initialized before $q$. Therefore, $q := y_i$ cannot destroy validity of the assertion.

Note that it is essential for this argument to work, that the re-initializations in $\mathit{KillAll}$ are done in the correct order as discussed in Section 4.2.

- $x_i := q_i$: such a statement could potentially invalidate conjuncts of the form $x_i = 0$ or $\mathit{OK}(x, p, c_0, c_1)$.

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

For conjuncts of the form $\mathit{OK}(x, p, c_0, c_1)$ the argument is more subtle. Similarly to [15], [24], and [33], $x_i := q$ is preceded by an assertion that ensures in particular that $\mathit{OK}(q, q_i, c_0, c_1 + i)$ holds, where $i \in \{-1, 0, 1\}$. By the construction of $\pi_M$, $i = -1$, 1, or 0 if there is a transition from $q_i$ to $p_i$ that increments, decrements, or leaves the counter $c_i$ unchanged, respectively. Now suppose that $x_i$ is assigned the value 1 by $x_i := q$, otherwise $\mathit{OK}(x, p, c_0, c_1)$ holds trivially. Then clearly $q = 1$ which implies $(p_0, 0, 0) \rightarrow^* (q_i, c_0, c_1 + x) \mathit{OK}(q, q_i, c_0, c_1 + x)$. By the transition from $q_i$ to $p_i$, this transition sequence can now be extended to a sequence $(p_1, 0, 0) \rightarrow^* (p_i, c_0, c_1)$. Hence, $\mathit{OK}(x, p_1, c_0, c_1)$ 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.